Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis

Design and Implementation of highly-efficient Data Structures for Parallel Complex Event Processing Framework on Multicore Shared Memory Architecture

Jane Shirley Joseph Victor Raj

Course of Study: INFOTECH

Examiner: Prof. Dr. rer. nat. Dr. Kurt Rothermel

Supervisor: Ahmad Slo, M.Sc.

Commenced: 2017-09-01

Completed: 2018-02-28

CR-Classification: C.1.4, C.2.4, C.4, D.1.3, E.1

Abstract

Social networks, financial applications, Internet of Things (IoT) technology, etc. are systems that produce a continuous stream of data. Due to the increase in the amount of data generated, there is a need for systems that facilitate the real-time interpretation of the generated data. Complex Event Processing (CEP) is one such system. A Distributed Complex Event Processing (DCEP) system helps to infer complex occurrences from the received sensor readings. In order to increase the throughput of operators, data parallel DCEP systems split incoming event streams into windows with overlapping events that can be processed independent of each other. However, as consumption policies call for consumed events of a window to be excluded from dependent windows, data parallelization between windows could be affected. This problem was solved in the existing work by means of speculation. The SPECTRE system proposes maintaining two different versions of a window for each event that could be consumed – one which assumes that a detected event will be consumed and one which assumes that the event will not be consumed. Several data structures are used to represent the relationship between the window versions and this is information is stored in the shared memory. As all components need to access the shared memory, it could cause a bottleneck on the system performance. Also, once the outcome of an event is known (whether it is consumed or not), the memory allocated for the irrelevant window versions should be reclaimed. This poses a load on the garbage collector. Hence, the goal of this thesis is to design data structures to improve memory allocation and utilization and ease the load on the garbage collector.

Contents

1 Introduction **11**

2 Complex Event Processing **13**
2.1 CEP Systems 13
2.2 Parallelization 14
2.2.1 Data parallel DCEP systems 15

3 Existing System **19**
3.1 The SPECTRE system 19
3.1.1 Consumption group 20
3.1.2 Window Versions 21
3.1.3 Dependency Tree 22
3.1.4 Maintenance of Dependency Tree 23
3.1.5 Markov process 26
3.1.6 Select top k Window Versions 26
3.1.7 Process top k window versions 28
3.2 Problem analysis 29

4 Solving Approach **31**
4.1 Proposed changes 31
4.1.1 Threshold probability calculation 32
4.1.2 Maintenance of dependency tree 33
4.1.3 Parallel processing of window versions 38
4.2 Comparison of dependency tree with and without lazy loading 40

5 Evaluations **43**
5.1 Experimental setup 43
5.2 Evaluation approach 45
5.3 Performance evaluation 46

6 Conclusion **55**

Bibliography **57**

List of Figures

2.1 Event Processing System 13
2.2 Complex Event Processing System 14
2.3 Distributed Complex Event Processing System 15
2.4 Data parallel DCEP System 15
2.5 Splitting of windows 16
2.6 Detected complex events 17
2.7 Working of Query Q 17

3.1 Working of SPECTRE system 19
3.2 Consumption Groups 20
3.3 Window Versions 21
3.4 Dependency Tree 23
3.5 Dependency tree (a) before and (b) after addition of window W_k 24
3.6 Dependency tree (a) before and (b) after addition of consumption group CG_3 . 24
3.7 Dependency tree (a) before and (b) after completion of CG_1. (c) After completion of CG_2 and CG_3 25
3.8 (a) Markov model and (b) Markov matrix 26

4.1 Dependency tree (a) before and after (b) addition of window W_j 34
4.2 Dependency tree (a) before and (b) after creation of CG_2. (c) Window lists . . . 36
4.3 Dependency tree (a) before and (b) after addition of WV_5 37
4.4 Dependency tree (a) before and (b) after completion of CG_1 39
4.5 Dependency tree (a) without and (b) with lazy loading 40

5.1 C1: Node count for the SPECTRE system (a) without and (b) with lazy loading 47
5.2 C1: Memory consumption for the SPECTRE system (a) without and (b) with lazy loading 48
5.3 C1: Tree size for the SPECTRE system (a) without and (b) with lazy loading . 49
5.4 C1: Throughput for the SPECTRE system (a) without and (b) with lazy loading 49
5.5 Throughput of the SPECTRE system with lazy loading for different values of M 50
5.6 Tree size for different pattern sizes and threshold multipliers 51
5.7 C2: Node count for the SPECTRE system (a) without and (b) with lazy loading 52
5.8 C2: Throughput for SPECTRE system with lazy loading 52

List of Algorithms

1 Top-k window version selection 27
2 Scheduling algorithm 28

3 Threshold probability calculation 33
4 Creation of a new dependent window 35
5 Creation of a new consumption group 36
6 Addition of a window version from window list to tree 37
7 Consumption group completed 38
8 Consumption group abandoned 38
9 Consistency check 40
10 Processing of events 41

1 Introduction

The Internet of Things(IoT) is a system of objects that are connected to the Internet and communicate with one another to create a ”smart environment”. The objects that make up a smart environment are called smart objects. Smart objects may range from electronics or everyday household appliances equipped with sensors or network connectivity to special sensors that monitor the environment for the occurrence of a special event. Smart objects generate sensor data that can be processed to control these objects remotely or they can be connected to automation systems that generate an action when certain ”events” occur. Complex Event Processing (CEP) is one such automation system that generates an action when a complex event is detected.

Since complex event processing systems process events from multiple streams [15], using a single operator running on a server is limited by the capabilities of that server. To overcome this, a distributed network of operators - called operator graph - is deployed. These operators process events independent of each other and the results from each operator are aggregated to give the final result [10].

In order to increase the operator throughput in a DCEP system, the incoming event stream is split into blocks of data (called windows) to be processed by multiple instances of the same operator. This is called data parallelization. In data parallel DCEP systems, the splitting of the event stream can be key-based [3] or batch-based [2]. The split windows may have events overlapping between them i.e. they may be dependent.

However, in the related works, the windows are assumed to be independent of each other and hence, they can be processed independently by operators. However, consumption policies do not allow for events already consumed in one window to be processed in another window [5]. This could pose a threat to parallelization as operators should wait for the output of a window that it depends on so that consumed events can be skipped. This problem was solved in the preceding work [14] using a speculative approach.

Here, for every event that could be consumed, different window versions are created for each window that contains this event. One version assumes that the event will be consumed and another assumes that the event will not be consumed. Based on the outcome, the corresponding window version will be kept and the other will be dropped. Since there are now multiple versions of a window based on events being consumed or abandoned, a tree data structure is used to keep track of the dependency between the various window versions.

Since all operators access a shared memory for this dependency information, the access to memory could cause a bottleneck on the system performance. In addition, several window versions are created for each event that is a part of a consumption group and half of the created versions are deleted once the outcome of a consumption group is known. This puts a load on the garbage collector.

In this work, we design data structures to keep the tree size minimal and reduce the number of window versions created unnecessarily (as they have to be garbage collected later). Hence, memory utilization and allocation is improved while easing the load on the garbage collector.

All of the content discussed above are detailed in the coming chapters. The document is structured as follows: the second chapter deals with general background work about the thesis and explains CEP, DCEP and data parallelization in detail. In Chapter 3, we explain the work that this thesis is based on in detail. We define the framework, its working and the problems in the system that we tackle in this work. In the fourth chapter, we describe the changes that are introduced to the existing system and how it solves the problems of the existing system. Chapter 5 deals with the evaluation of our work and how it compares to the existing model. Finally, we outline our findings and results in the final chapter.

2 Complex Event Processing

In this chapter, we first provide some background information about Complex Event Processing systems and its basic components. Then, we talk about distributed complex event processing systems and parallelization in DCEP systems. Finally, we discuss data parallel DCEP systems in detail.

2.1 CEP Systems

With the growth of ”Internet of Things” (IoT) and smart objects, the amount of sensor data being generated has become very high. The data generated by these sensors are called events and are usually sent to an event processor for further actions. Event processors process and analyze these events to detect a certain condition. So, an event processing system basically consists of three main components: an event source, an event processor and an event consumer [9].

The basic components in an event processing system are shown in figure 2.1 The event source produces events into the event processing system. An event source could be smart objects, Global Positioning System (GPS) sensors or Radio Frequency Identification (RFID) sensors. The event processing system could perform event transformation (like adding timestamps), filter events or check for a certain condition. The event consumer consumes the event and performs actions accordingly.

Figure 2.1: Event Processing System

In Complex Event Processing (CEP), a concept introduced by D. Luckham in [11], events from multiple event sources are processed to detect interesting or predefined patterns and respond to them as soon as possible. The pattern to be detected is called query and is specified in an event specification language. The pattern could be used to detect sequence of events, relationship between events or causality of events.

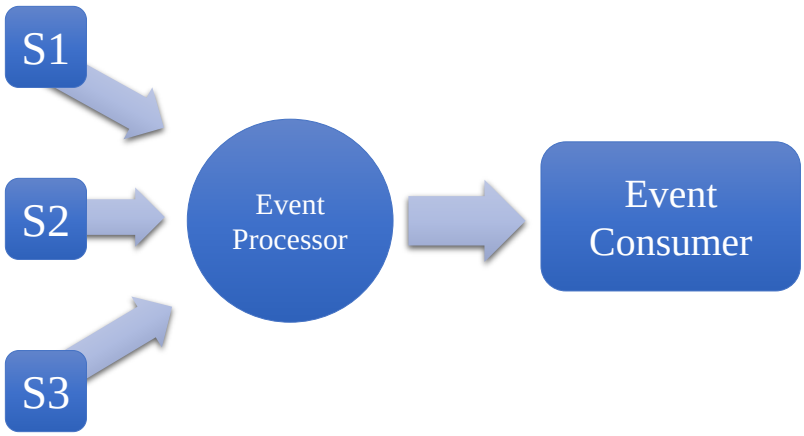


Figure 2.2: Complex Event Processing System

For example, if there is a temperature sensor monitoring an environment which needs to have a constant temperature, then any change detected by the sensor should trigger an action to adjust the temperature so that it is maintained at the desired value. In order to do so, the sensor measures the temperature at specific intervals and this data is sent in the form of a continuous stream of events. Here, the pattern to be detected would be the change in the value of the temperature received.

As the consumers do not know about the number of sources and vice versa, CEP systems are extremely scalable [4]. However, handling data from multiple sources increases the load on the event processor and that may interfere with real time processing of data. Hence, Distributed Complex Event Processing (DCEP) systems were introduced.

DCEP systems uses a distributed network of operators (event processors) in order to handle higher workloads in real-time and scale system performance. Each operator processes the input event stream and detects a portion of the event pattern. Upon detection, a new event is emitted either to the consumer or the next operator. The entire DCEP system is deployed as an operator graph as shown in fig. 2.3, where each component is connected by an event stream i.e. operators are nodes connected by event stream edges.

2.2 Parallelization

Operator graphs (called stream graphs in [8]) can be parallelized in three different ways - pipeline, task and data [8]. In pipeline parallel systems, the source, operator and consumer work concurrently to achieve parallelism. In task parallel systems, different operators that are not in a pipeline, perform their tasks concurrently. Data parallel systems concurrently run multiple instances of the same operator and each instance works on different portions of the data.

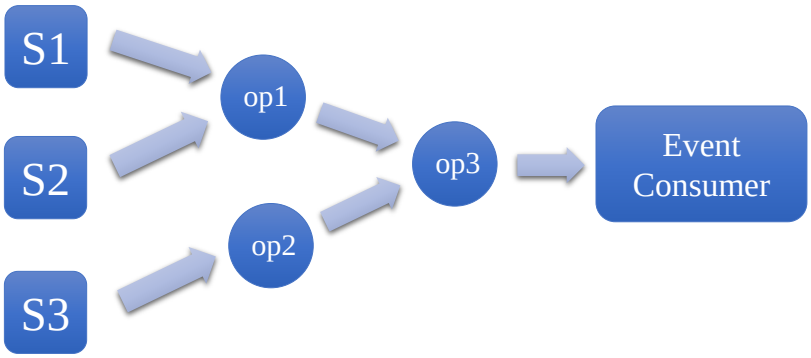


Figure 2.3: Distributed Complex Event Processing System

2.2.1 Data parallel DCEP systems

A data parallel DCEP system has a split-process-merge architecture [2, 12, 13] as shown in fig. 2.4. Here, the input stream is first received by a splitter that splits the incoming event stream into blocks called windows. Each event is an attribute-value pair as follows (meta-data, payload). The meta-data contains information such as time-stamp, event type, sequence number, etc. and the payload contains the actual information such as sensor readings. The meta-data is useful for determining the global ordering of the events.

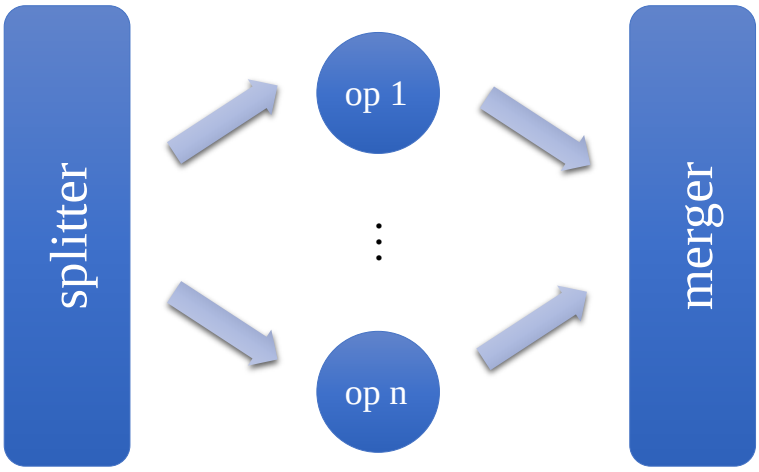


Figure 2.4: Data parallel DCEP System

The splitting of events can be based on time (a window is opened every t seconds from the opening of the previous window and a opened window is closed after m seconds) [12], number of events (a window is opened every t seconds from the opening of the previous window and closed after n number of events) or a custom predicate [12]. We call these predicates *PStart* and *PClose*. For each event, the splitter checks if that event opens or closes a window. Once

a window is opened, it is scheduled for processing using the underlying scheduling algorithm. The windows usually have an overlap between them. In fig. 2.5, the splitting is done using a custom *PStart* predicate. Here, a new window is opened for every event of type A. The meta-data information is used to determine the type of event.

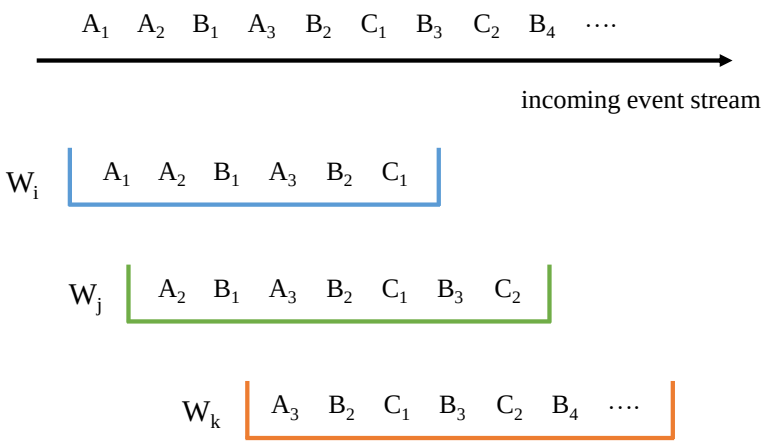


Figure 2.5: Splitting of windows

Each window is processed independently and in parallel by the operators. The operator processes the events based on their global ordering. If a pattern is detected, an event is emitted by the operator to its successor in the operator graph.

The event pattern to be detected is specified in an event specification language like Tesla [6], Snoop [5] or Amit [1]. These languages contain operators to specify selection, negation, aggregation, sequences, etc in order to detect patterns in the incoming stream. Specifying what qualifies a pattern is called selection policy.

For example, consider the query Q in Tesla [6],

```
define Influence(Factor)
from A() and each B() within 3min from A and
last C() within 1min from A
where Factor = C: change / B: change / A: change
consuming B
```

Here, the *selection policy* is "first A, each B, last C". Hence, from the event stream and windows shown in fig. 2.5, the following complex events are detected (see fig. 2.6): $(A_1B_1C_1)$, $(A_1B_2C_1)$, $(A_2B_1C_2)$, $(A_2B_2C_2)$, $(A_2B_3C_2)$.

In addition, event specification languages also specify which events in the detected event pattern to consume. This is called consumption policy [1, 5, 6, 18]. Sometimes, no event, all events

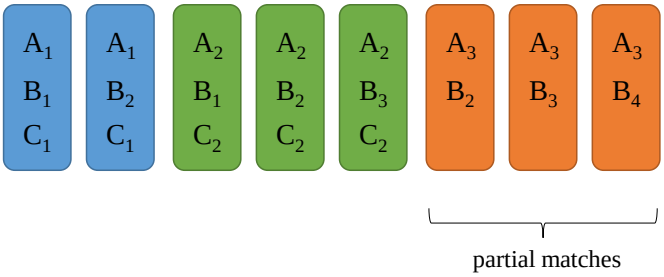


Figure 2.6: Detected complex events

or some events in the detected pattern may be consumed. When a complex event is detected, all events in the pattern are checked against the consumption policy and all events specified in the consumption policy are consumed. Therefore, events are consumed only when the entire pattern is matched and a complex event is produced and not for a partial match. In Query Q, the consumption policy is ”consume B”.

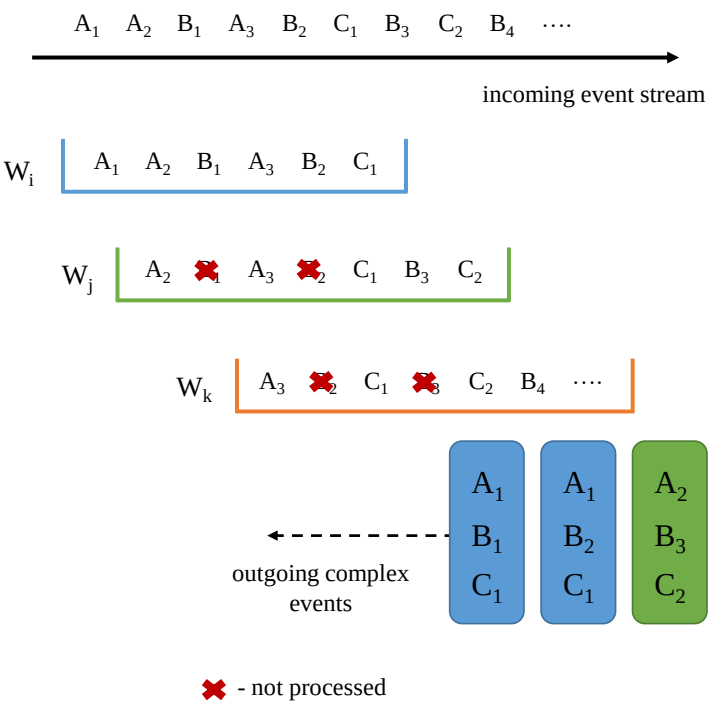


Figure 2.7: Working of Query Q

If the consumption policy were ”none” in Query Q, then all detected events would have been sent to the output stream. However, since the consumption policy is ”consume B”, only $(A_1B_1C_1)$, $(A_1B_2C_1)$ and $(A_2B_3C_2)$ are sent to the output stream. Note that, in window W_k,

the complex event *(A_3B_4)* is the only valid partial match. If C_1 or C_2 happen to be the ”last C” in window W_k, then no complex event will be detected in W_k.

Figure 2.7 shows the working of Query Q when applied to windows from fig. 2.5. It is evident that consumption policies introduce a dependency between windows as consumed events can only be part of the window that consumes them i.e consumed events cannot be processed in other windows. Due to this, windows may need to wait for the outcome of a partial match to know if an event will be consumed. For example, the successive window of W_k (say W_{k+1}) will need to wait until *(A_3B_4)* becomes a complete match to know if event B_4 will be consumed or not (since events are only consumed when the entire pattern is matched). Hence, W_{k+1} will have to wait until W_k has finished processing to know if it should process or omit event B_4. Due to this, the processing of windows becomes almost sequential.

This problem was solved in [14] by creating multiple versions for a window. As our work is built upon this system, we explain the framework and its working in detail in the following chapter.

3 Existing System

The existing system architecture and its working are discussed in this chapter. First the various data structures used in the system and how they solve the problems stated in the previous chapter are discussed. Finally, we describe the working of the system followed by the problems in the system that we hope to solve with our work.

3.1 The SPECTRE system

The basic working of the SPECTRE (SPECulaTive Runtime Environment) [14] system is shown in fig. 3.1

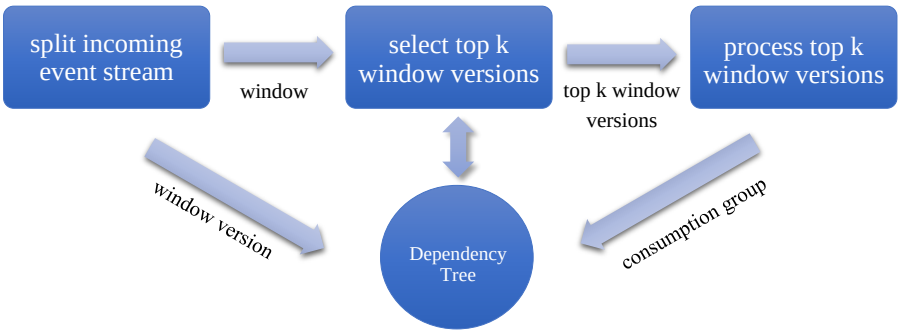


Figure 3.1: Working of SPECTRE system

As discussed earlier, the incoming event stream is split into windows and windows may have a few events in common between them i.e. windows may overlap. For each operator, the windows scheduled to be processed at that operator are ordered according to their start events. For example, a window w_j, occurs before window w_k iff the start event of w_j, occurs before the start event of w_k in the global event stream. Windows may also have a consumption dependency between them. A consumption dependency is defined as follows: A window w_k depends on window w_j, if consumption of events in w_j, affects processing of w_k. Finally, a two windows are said to have a dependency *if w_j occurs before w_k and they have an overlap of events*. A window which does not have such a dependency is called an independent window. From fig. 2.5, window W_j depends on W_i, window W_k depends on W_i and W_j and window W_i is an independent window.

3.1.1 Consumption group

When an operator processes a window and a partial match occurs, the operator creates a consumption group based on the consumption policy. A consumption group is created for each partial match and consists all events of that particular window that need to be consumed if the partial match becomes a complete match. The events in a consumption group are consumed only if a complete match occurs and will be discarded otherwise. Hence, a consumption group is said to be completed when the last event that completes the pattern is processed and abandoned if there is no possibility for a complete match. Several consumption groups could be created while processing a window and all of them will either be completed or abandoned.

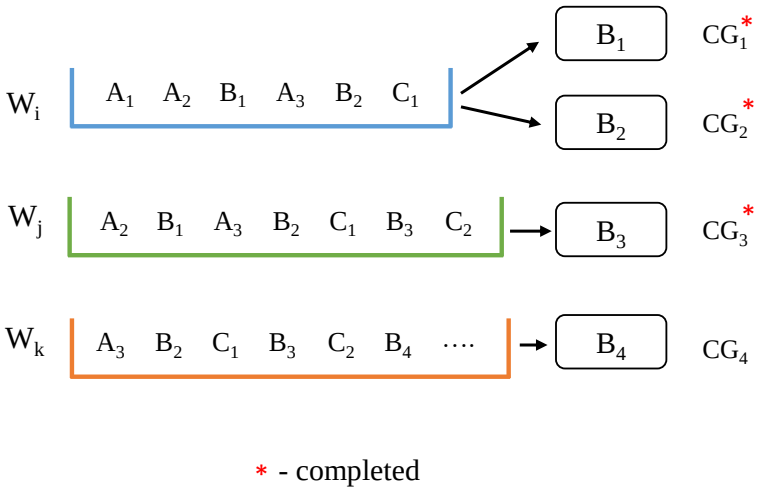


Figure 3.2: Consumption Groups

Considering the same example as before, for selection policy ”first A, each B, last C” and consumption policy ”consume B”, it is seen from fig. 3.2 that each window creates one or more consumption groups based on their partial matches. Consumption groups CG_1, CG_2 and CG_3 are considered ”completed”, as a complete match has occurred i.e. the ”last C” has occurred. Consumption group CG_4 however could either be completed or abandoned. If an event of type C occurs before the window is closed, then a complete match would occur and CG_4 will be completed. If however, the window closes without the occurrence of an event of type C, then CG_4 will be abandoned.

How SPECTRE handles consumption of events and what happens when a consumed event was already processed in another window will be discussed later.

3.1.2 Window Versions

Since a consumption group associated with a window *w* can either be abandoned or completed, this affects all windows that depend on window *w*. All events consumed in window *w* should not be processed in windows that depend on *w*. Because of this consumption dependency, the dependent windows will have to wait till window *w* has finished processing so that the outcome of the consumption group is known (abandoned or completed). This interferes with the parallel processing of windows.

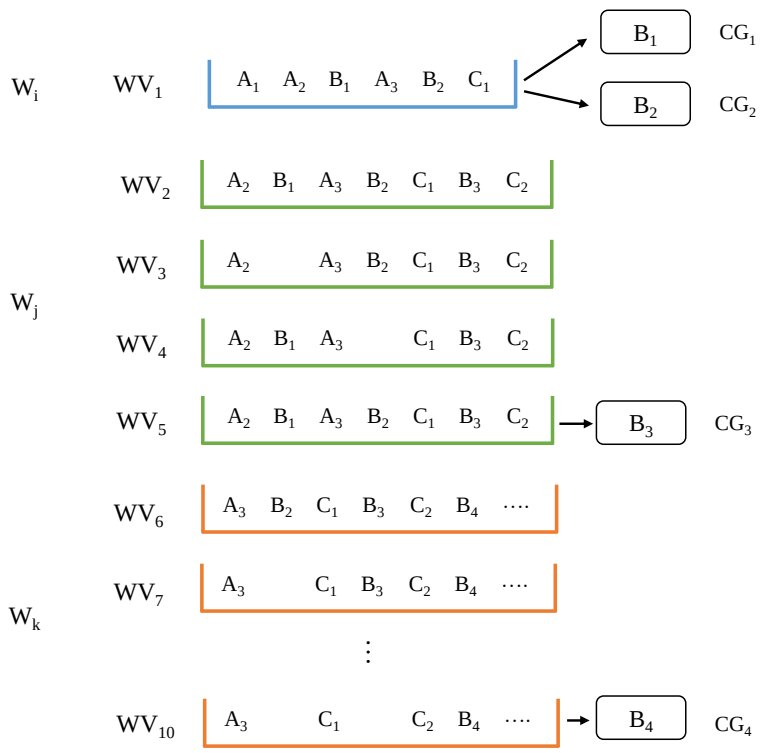


Figure 3.3: Window Versions

In order to maintain the parallelization between dependent windows, the SPECTRE system maintains different versions of a window. Whenever a new consumption group is created for window *w*, two window versions are created for every window that depends on *w*. One window version assumes that the consumption group will be completed and the other assumes that the consumption group will be abandoned. Both window versions are executed in parallel along with *w* and depending on the whether the consumption group is abandoned or completed the correct window version is processed further and the other one is dropped. Therefore, for every consumption group created, there are two window versions - one assuming the consumption group will be completed and one assuming that the consumption group will be abandoned.

In this way, even though there is dependency between windows, they can be processed in parallel.

For example, in fig. 3.3, when a new consumption group CG_1 is associated with window W_i, two versions of window W_j are created. Window version WV_2 assumes that CG_1 will be abandoned and so event B_1 is processed. Window version WV_3 assumes that CG_1 will be completed and B_1 is not processed. Since window W_i produces two consumption groups, four different window versions are created for W_j. WV_3 and WV_4 assume that CG_1 and CG_2 will be completed respectively. WV_2 assumes that neither CG_1 nor CG_2 will be completed and WV_5assumes that both CG_1 and CG_2 will be completed.

Once it is know that CG_1 is completed, WV_2 and WV_4 will be dropped. Likewise, when CG_2 is completed, WV_3 will be dropped and when CG_3 is completed, WV_7 will be dropped. Note that if window versions WV_2, WV_3, WV_4, .. are processed, they would also create consumption groups and corresponding window versions. However, to keep things simple, we do not go into such details.

3.1.3 Dependency Tree

As seen above, numerous window versions are created and it is difficult to keep track of which window versions correspond to a particular consumption group. In order to denote the relationship between consumption groups and window versions, a dependency tree is used. Consumption groups and window versions are depicted as vertices and edges represent the dependency between them. Therefore, the root of the dependency tree is the only independent window and for every independent window, there exists a dependency tree. Figure 3.4 shows a dependency tree where $v(WV_1)$ is the only independent window.

The vertex representing a window version WV, denoted by v(WV) can have one or no children. The child of a v(WV) can be another window version which is dependent on WV or a consumption group associated with WV. All window versions in the child sub tree of v(WV) are dependent on WV.

The vertex v(CG) of a consumption group CG, has two children. Each child represent the two possible outcomes - abandon and completion. The abandon edge links v(CG) to the window version that corresponds to the consumption group being abandoned and the completion edge links v(CG) to the window version that assumes that the consumption group will be consumed. All window versions in the completion sub tree of a v(CG) does not contain events in the consumption group where as the window versions in the abandon sub tree do. Based on whether the consumption group completes or not, the corresponding sub tree will be removed from the dependency tree.

From fig. 3.4, once it is known that consumption group CG_1 will be completed, the abandon sub tree (left sub tree) will be cut i.e the vertices $v(CG_2)$, $v(WV_2)$, $v(WV_4)$, $v(WV_6)$ and $v(WV_7)$

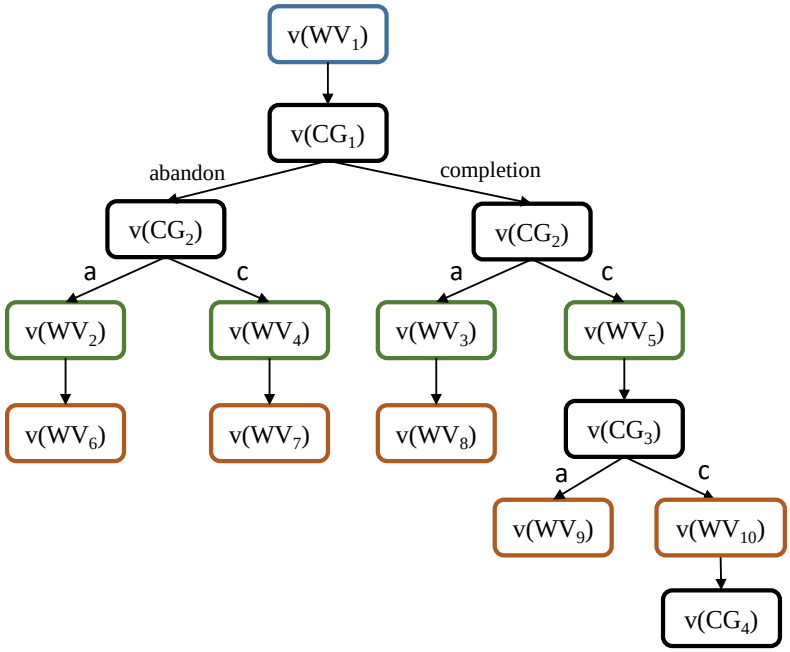


Figure 3.4: Dependency Tree

will be removed from the tree. When CG_2 is completed, $v(WV_3)$ and $v(WV_8)$ will be removed and the completion of CG_2 removes $v(WV_9)$ from the tree.

3.1.4 Maintenance of Dependency Tree

Changes to the dependency tree may be due to

(1) the opening of a new dependent window

(2) creation of a new consumption group

(3) when a consumption group is either completed or abandoned

New dependent window created:

Whenever a new dependent window is opened, it is added as the child of every leaf vertex rooted at the vertex that the new window is dependent on. If the leaf vertex is a window version, the new window is added as its child. If it is a consumption group, then the original window version is added at the abandon edge and a modified version of the window is created that does not include events in the consumption group. The modified window version is added at the completion edge.

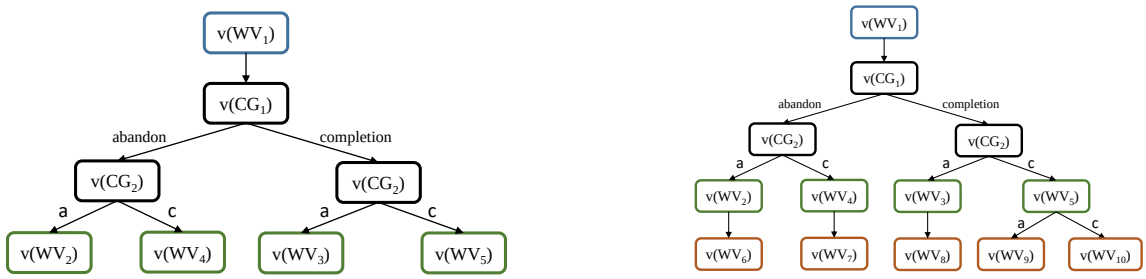


Figure 3.5: Dependency tree (a) before and (b) after addition of window W_k

Figure 3.5 shows the changes to the dependency tree when window W_k is created. At this point only window versions WV_6 - WV_9 are created as consumption group CG_3 is not yet created. The corresponding vertices of the window versions are added to every leaf vertex in the dependency tree.

New consumption group created:

When a new consumption group is created by a window version, a vertex of the consumption group is added as the child of the window version that created it. The sub tree that was previously the child of the window version is added at the abandon edge of the new consumption group vertex and a modified copy of the window versions in the original sub tree are added at the completion edge. In the modified sub tree, all events part of the consumption group are suppressed i.e not processed.

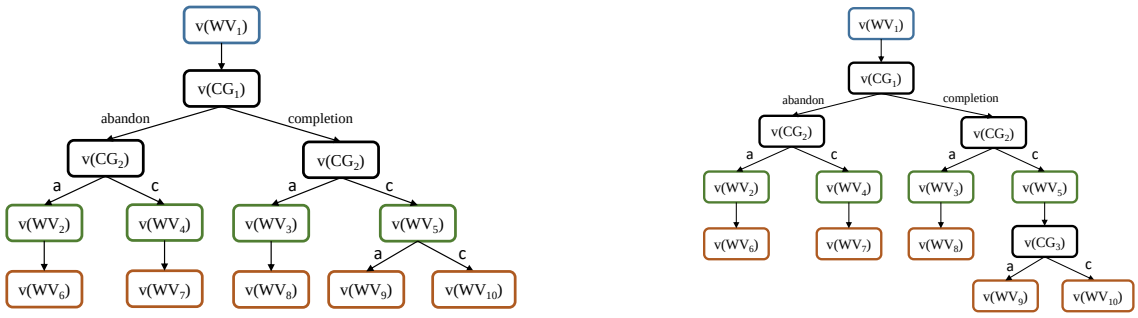


Figure 3.6: Dependency tree (a) before and (b) after addition of consumption group CG_3

For example, from fig. 3.6, when WV_5 creates the consumption group CG_3, the existing child of $v(WV_5)$ becomes the left child of $v(CG_3)$ and a new version of WV_9 suppressing the events in CG_3 is created. Hence, WV_{10} is a copy of WV_9 omitting event B_3.

In the case of WV_1, when CG_2 is created, two copies of the consumption group are added at each edge of CG_1 and the existing children should be added at the abandon edge of the new consumption as usual. Hence, the sub tree previously at the abandon edge of $v(CG_1)$ is added to the copy of the consumption group at the abandon edge and the sub tree previously at the completion edge of $v(CG_1)$ is added to the copy of the consumption group at the completion edge. Therefore, $v(WV_2)$ is added to the abandon edge of $v(CG_2)$ at the abandon edge of $v(CG_1)$ and $v(WV_3)$ is added to the abandon edge of $v(CG_2)$ at the completion edge of $v(CG_1)$. Modified versions of WV_2 and WV_3 (WV_4 and WV_5 resp.) are added to the corresponding completion edges (refer fig. 3.5a).

Consumption group completed or abandoned:

When the complete pattern has been detected in a window, then the corresponding consumption group is considered complete and if there is no possibility for a complete match, then the consumption group is abandoned. Whenever the outcome of a consumption group is known, the sub tree attached to the contradicting edge in the dependency tree is removed. If the consumption group is abandoned, the completion edge of the consumption group vertex is removed and vice versa. In addition, the consumption group is also removed from the tree and a complex event is sent to the outgoing stream. The remaining sub tree becomes the child of the window version that created the consumption group.

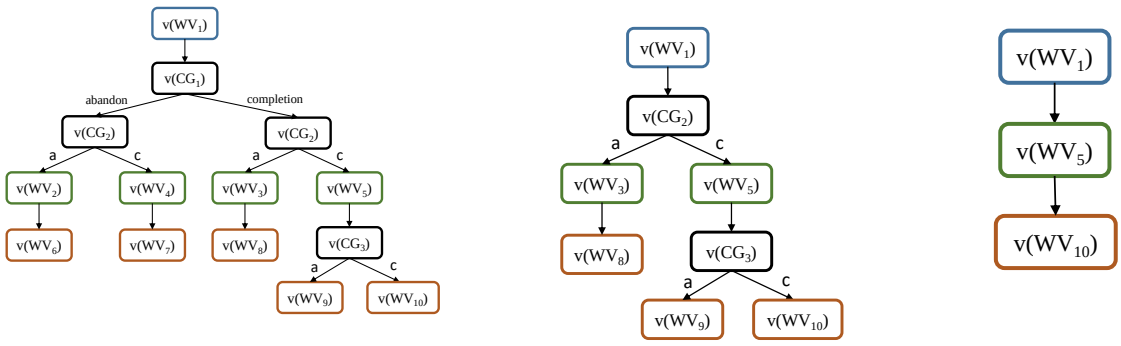


Figure 3.7: Dependency tree (a) before and (b) after completion of CG_1. (c) After completion of CG_2 and CG_3

For example, in fig. 4.4, once consumption group CG_1 is completed, the abandon sub tree i.e. the sub tree rooted at $v(CG_2)$ is discarded and the sub tree rooted at the completion sub tree becomes the child of $v(WV_1)$. Figure 3.7c, shows the dependency tree after the completion of consumption groups CG_2 and CG_3. When CG_2 is completed, $v(WV_5)$ becomes the child of $v(WV_1)$ and the subtree rooted at $v(WV_3)$ is removed. Likewise, when CG_3 is completed, $v(WV_{10})$ becomes the child of $v(WV_5)$ and $v(WV_9)$ is deleted.

3.1.5 Markov process

A process which satisfies the Markov property is called a Markov process. The Markov property is defined as follows: a stochastic process where the future only depends on the present and is independent of the past. A discrete time Markov process is called a Markov Chain [16].

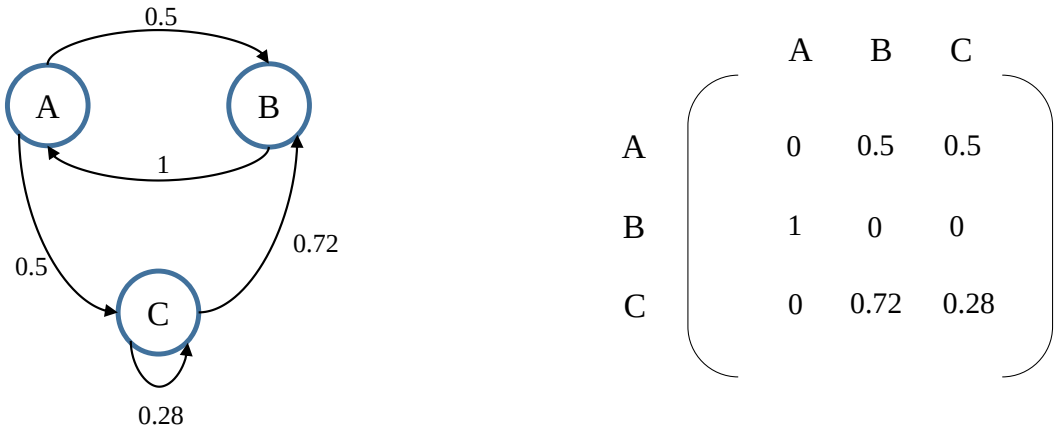


Figure 3.8: (a) Markov model and (b) Markov matrix

A Markov process can be represented as a state machine in which the edges are labeled with the probability of transition from one state to another. For example, from fig. 3.8a, if the Markov process is in state A, then there is an equal probability of changing to either state B or C and if it is in state C, then the probability that it changes to state B is 0.72 while the probability that it stays in the same state C is 0.28. Based on this state machine, a Markov matrix can be built as shown in fig. 3.8b.

3.1.6 Select top k Window Versions

If there are only k operator instances, then only k window versions can be processed in parallel. The scheduler chooses the best k window versions and assigns them to the operator instances for processing. The window versions are chosen based on the probability that they will survive. The survival probability of a window version depends on the consumption groups present in the path from the root to that particular window version. For example, from fig. 3.4, the survival probability of WV_8 depends on consumption groups CG_1, CG_2 and CG_3.

Since a consumption group has two edges - abandon and complete - the outcome of the consumption group invalidates one of these edges and the survival probability of all window versions in that sub tree drops to 0. If a consumption group is abandoned, then the abandon edge becomes valid and vice versa. Hence, a window version only survives if all edges (abandon

or complete) in its path to the root become valid. Even if one edge becomes invalid, then the window version is discarded. Therefore, it can be seen that the survival probability of a window version depends on the completion probability of all consumption groups on the path to the root.

Algorithm 1 Top-k window version selection

```
1:  procedure FINDTOPKWINDOWVERSIONS(dependencyTree, k)
2:      topK ← {}
3:      candidates ← {dependencyTree.root}
4:      for i ← 1...k do
5:          temp ← candidates.pop()
6:          topK.append(temp)
7:          for each T ← temp.child do
8:              candidates.add(T)
9:          end for
10:     end for
11:     return topK
12: end procedure
```

Probability calculation

Let SP(WV) be the survival probability of the window version WV and P(CG) be the probability that a consumption group will be completed.

$$SP(WV) = \prod_{c \in CG_c} P(c) \times \prod_{a \in CG_a} (1 - P(a))$$

CG_c and CG_a are the set of consumption groups that contribute complete and abandon edges to WV's root path, respectively.

The completion probability of a consumption group depends on 1. the inverse degree of completion i.e., how many events are at least required for the pattern to complete denoted by δ and 2. the expected number of events left in the window denoted by n. If is δ low and many events are still expected to occur (n is high), then completion probability is high. And if δ is low and very few events are remaining in the window (n is low), then the completion probability is low.

The process of pattern completion while events are being process is modeled as discrete-time Markov process. The Markov process has 0 to δ states where δ is the number of events required to finish the pattern. Therefore, if two more events are required to complete the pattern, then the Markov process has three states - "0", "1" and "2". States "1" and "2" represent that 1 more and 2 more event(s) are required to complete the pattern respectively. State "0" represents pattern

completion. Based on these states, a matrix T is built as shown in fig. 3.8b. The matrix T is a $\delta \times \delta$ matrix and contains the transition probabilities between each state. The matrix is updated when each event is being processed.

Scheduling

The top k window versions with the highest survival probabilities are selected and scheduled periodically. This is made easy as, the dependency tree is maintained in such a way that the survival probability of window versions decreases as the tree depth increases i.e. probability decreases in the root to leaf direction. Hence, it is easy to select the top k window versions for scheduling at any given time. Also, note that each window version is scheduled for processing only once. The algorithm for selecting the top k window versions is shown in algorithm 1. The top k window versions are selected using two data structures - a priority queue and a list to store the top k window versions. The window versions are added to the priority queue starting from the root until the top k window versions are found. The priority queue sorts window versions, highest probability first.

Algorithm 2 Scheduling algorithm

```
1:  procedure SCHEDULE(topKWindowVersions)
2:      List ⟨OperatorInstance⟩opInst
3:      List ⟨WindowVersion⟩toBeScheduled
4:      List ⟨OperatorInstance⟩freeOpInst ← operatorInstances
5:      for each WV in topKWindowVersions do
6:          if not WV.isScheduled() then
7:              toBeScheduled.push(WV)
8:          else
9:              freeOpInst.remove(WV.getOperatorInstance())
10:         end if
11:     end for
12:     for each WV in toBeScheduled do
13:         OperatorInstanceop ← freeOpInst.pop()
14:         op.schedule(WV)
15:     end for
16: end procedure
```

3.1.7 Process top k window versions

Once the top k window versions are scheduled for execution, they are processed in parallel. The way that a window version is processed depends on all the consumption groups in the path to the root, especially the completion edge. If there are completion edges on the path to the root,

then all events in the consumption group are suppressed and all complex events emitted are kept buffered until the outcome of the consumption group is known. Since all window versions are being processed in parallel, there might be situations where an update to a consumption group could be propagated too late and cause inconsistencies. For example, an event could have been processed in a window version only to be later updated as a part of the consumption group that the window version is at the completion edge of. In order to avoid such situations, a periodic consistency check is done and if it is found that an event that should have been suppressed has been processed instead, then the window version is reprocessed from the start.

3.2 Problem analysis

In the SPECTRE system, all data are stored in the shared memory and as all components access the same shared memory, memory access becomes a bottleneck and hinders the system with respect to scaling. Also, due to the huge amount of data, the garbage collector becomes overloaded and slows down the system.

As shown in fig. 3.1, all components of the system access a shared memory. As the dependency information is maintained in a tree data structure, as the depth of the tree increases, the number of vertices could expand exponentially whenever a new consumption group is created. And whenever a consumption group is either completed or abandoned, the corresponding branch is removed. This puts a load on the garbage collector and could interfere with the processing of events. Hence, to keep the tree data structure in check and reduce the load on the garbage collector, lazy loading of the tree is proposed. Here, a list of window versions are stored without adding them as vertices to the tree. Vertices are created and added to the tree only when needed (determined using a threshold probability).

4 Solving Approach

In this chapter, we discuss our work in detail. First, we talk about the changes that were introduced to overcome the problems in the existing system. Then, the new data structures that were introduced, the changes in the working of the system due to the new data structures and how it solves the existing problems are discussed. Finally, we compare the tree data structure in particular with and without changes made by our work.

4.1 Proposed changes

The following changes are made to the existing system:

- A new kind of data structure that stores a list of windows (WL) is introduced.
- A threshold probability (P_{th}) is calculated to determine when a window from WL is to be added to the tree.

As seen in the previous chapter, too many vertices are created in the tree only to be removed later. The memory can be utilized better if this was avoided. Hence, we store window versions in a new data structure and do not create vertices for them until their probabilities are high enough to survive. In order to determine if the probabilities are high enough, we calculate a threshold probability from the probabilities of the top *n* window versions in the dependency tree.

Window lists

As the name suggests, a window list contains a list of windows. The windows in a window list are ordered according to the global order of events i.e. if a window w_i occurs before a window w_j in the window list, then the start event of w_i occurs before the start event of w_j in the global event stream. A window list is created in two cases

(1) when a new consumption group is added to the dependency tree and

(2) when the probability of the parent in the tree is lower than the calculated threshold probability.

In terms of dependency tree, the vertex of a window list is always a leaf vertex i.e. it does not have any children.

Threshold probability

In order to decide which survival probabilities are high enough to allow creation of window versions, we need a threshold value. Hence, we calculate a threshold probability that allows to decide when window versions can be created for a particular window or window version. This threshold probability is calculated using the survival probabilities of the top *n* window versions in the dependency tree. The calculated threshold probability is used to determine when a window list is to be created for a vertex of a window version and when a window version should be created for a window in the window list.

4.1.1 Threshold probability calculation

As discussed in chapter 3, window versions are scheduled for processing based on their survival probabilities. For calculating the threshold probability, we use the calculated survival probabilities of the window versions to determine which window versions have a higher probability of being scheduled next. In doing so, we can avoid multiple versions of the window being created and buffer the window in a window list until the probability crosses the calculated threshold. In order to calculate the threshold probability, we use a threshold multiplier M.

First, the survival probabilities of $k \times M$ (referred to as *n* in the previous sections) window versions are calculated, where, *k* is the number of operator instances. Once the probabilities are calculated, the probability of the $(k \times M)^{th}$ window version is chosen as the threshold probability. The threshold probability calculation is done while selecting the top-k window versions in order to avoid probabilities being calculated multiple times and wasting system resources.

Algorithm 3 shows the modified algorithm for selecting top k window versions and calculating threshold probability. The algorithm works in the same way as the *findTopKWindowVersions* algorithm (ref: Algorithm 1), but has been modified so that it calculates the top $k \times M$ window versions instead of top k window versions (line 5). The actual scheduling takes place once top k window versions are calculated (lines 12 - 14). Note that window versions are scheduled as soon as top-k versions are calculated in order to better utilize system resources (line 14). Once the top k window versions are scheduled, the algorithm continues until $k \times M$ window versions are selected and the threshold probability is calculated.

Algorithm 3 Threshold probability calculation

```
1:  procedure CALCULATETHRESHOLDPROBABILITY(dependencyTree, k, M)
2:      List ⟨WindowVersion⟩topK
3:      wList ← {}
4:      candidates ← {dependencyTree.root}
5:      size ← k × M
6:      for i ← 1...size do
7:          temp ← candidates.pop()
8:          wList.append(temp)
9:          for each T ← temp.child do
10:             candidates.add(T)
11:         end for
12:         if i = k then
13:             topK ← wList
14:             schedule(topK)
15:         end if
16:     end for
17:     P_th ← wList[size].probability
18:     return P_th
19: end procedure
```

4.1.2 Maintenance of dependency tree

The maintenance of the dependency tree was already discussed in Chapter 3. However, as a new kind of vertex (for a window list) now needs to be added to the tree, there are changes to the way the dependency tree is maintained. The vertex of a window list WL is denoted as v(WL). The changes in the maintenance algorithms introduced by the new vertex are discussed below.

New dependent window created

As discussed earlier, when a new dependent window is created, window versions are created based on the consumption groups present on the path from root to leaf and their corresponding vertices are added to every leaf vertex in the tree. The only change is that now, the addition of a vertex for a window version depends on the probability of the leaf vertex. For the vertex of a window version, a new vertex is to the leaf vertex only if the probability of the leaf vertex is greater than the threshold probability. Otherwise, a window list is created with the window version and the vertex of the window list is added to the leaf vertex. For the vertex of a consumption group, the addition of the window version to the tree depends on the probability of the outcome of the consumption group. If the consumption group has a higher probability to complete, then the window version is added to the completion edge and a window list is added

at the abandon edge. And if the consumption group has a higher probability to be abandoned, then the window version is added at the abandon edge and a window list is added at the completion edge. In addition, the leaf vertex could be that of a window list, if so, the window version is added to it.

Figure 4.1: Dependency tree (a) before and after (b) addition of window W_j

Figure 4.1 shows the dependency tree before and after the creation of window W_j. Here, since the leaf vertex is a consumption group, we check the probability of the consumption group being completed for the right child and the probability of the consumption group being abandoned for the left child. Since the consumption group has a higher probability for completion, a window version is added at the completion edge and a window list containing the corresponding window version is added at the abandon edge.

Algorithm 4 shows the procedure to be followed when a new dependent window is created. Lines 3 - 10 deal with a leaf vertex containing a window version and lines 11 - 25 deal with a leaf vertex containing a consumption group. Lines 26 - 28 deal with a leaf vertex containing a window list.

If the leaf vertex is a window version, then a vertex for the new window version is added to the tree if the probability of the leaf vertex is higher than the threshold probability (lines 4, 5). If the probability of the leaf vertex is lesser than the threshold probability, then a new window list created and the window version is added to it (lines 7, 8). The vertex of the window list is then added to the tree (line 9). If the leaf vertex is a consumption group, the abandon and completion probabilities are calculated to determine if the window version should be added to the tree or if it should be added to a window list. Line 12 checks completion probability to determine if the left child should be a window version or a window list and line 19 checks the abandon probability to determine the right child. If the leaf vertex is already a window list, then the window version is appended to the window list.

Algorithm 4 Creation of a new dependent window

```
1:  procedure DEPENDENTWINDOWCREATED
2:      for each leafVertex ∈ dependencyTree do
3:          if leafVertex is a window version then
4:              if leafVertex.probability > P_th then
5:                  leafVertex.child ← new v(WV)
6:              else
7:                  WL ← {}
8:                  WL.append(WV)
9:                  leafVertex.child ← v(WL)
10:             end if
11:         else if leafVertex is a consumption group then
12:             if leafVertex.completionProbability > P_th then
13:                 leafVertex.completionChild ← new v(WV)
14:             else
15:                 WL ← {}
16:                 WL.append(WV)
17:                 leafVertex.compeletionChild ← v(WL)
18:             end if
19:             if leafVertex.abandonProbability > P_th then
20:                 leafVertex.abandonChild ← new v(WV)
21:             else
22:                 WL ← {}
23:                 WL.append(WV)
24:                 leafVertex.abandonChild ← v(WL)
25:             end if
26:         else                                          ▷ it is a window list
27:             leafVertex.windowList.append(WV)
28:         end if
29:     end for
30: end procedure
```

New consumption group created

As discussed earlier, when a new consumption group is created, a modified copy of the existing sub tree will be added at the completion edge and the original sub tree is added at the abandon edge. In our approach, when a new consumption group is created by a window version, copies of the window versions in the child sub tree are added to a window list and the vertex of the window list is added at the completion edge while the original sub tree is added at the abandon edge. If there was a window list in the sub tree at the abandon edge, then, the window versions from that list are appended to the window list at the completion edge.

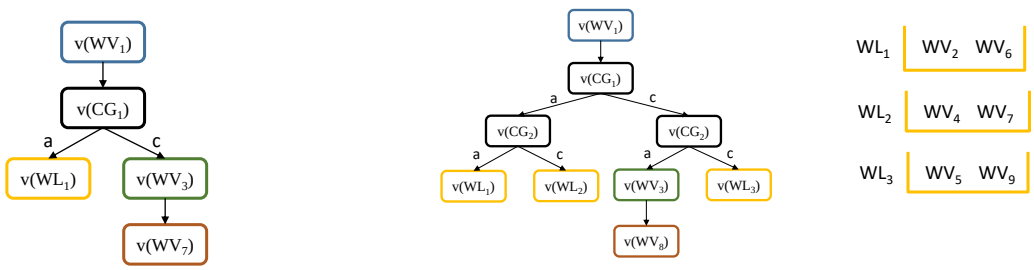


Figure 4.2: Dependency tree (a) before and (b) after creation of CG_2. (c) Window lists

As seen in fig. 4.2, when CG_2 is created, the existing children of $v(CG_1)$ are added at the abandon edge of the two copies of $v(CG_2)$. As for the completion edge, window lists are created that contain the modified window versions of the window versions in the respective abandon edges. Window list WL_2 contains window versions in window list WL_1 excluding the event B_2 and window list WL_3 contains versions of the window versions WV_3 and WV_8 excluding the event B_2.

Algorithm 5 Creation of a new consumption group

1: **procedure** CONSUMPTIONGROUPCREATED(ConsumptionGroup CG, WinVersion WV)
2: $WL \leftarrow \{\}$
3: **for each** $WV \in v(WV).childSubTree$ **do**
4: $WL.append$(modified WV)
5: **end for**
6: $v(CG).completionEdge \leftarrow v(WL)$
7: $v(CG).abandonEdge \leftarrow v(WV).childSubTree$
8: $v(WV).child \leftarrow v(CG)$
9: **end procedure**

As can be seen from Algorithm 5, when a new consumption group CG is created by window version WV, all window versions in the child sub tree of v(WV) are modified and added to a window list (lines 3 - 5). Then the vertex of the window list is added at the completion edge of

the newly created consumption group and the child sub tree of the window version WV is added at the abandon edge of the consumption group (lines 6, 7). Finally, the created consumption group is added as the child of the window version that created it (line 8).

Addition of a window version from a window list to the dependency tree

Sometimes a window version from a window list might need to be added to the dependency tree. This occurs when the probabilities of the the window versions in the window list are high enough that they need to be processed next.

Algorithm 6 Addition of a window version from window list to tree

1: **procedure** ADDWINDOWVERSION(WindowList WL)
2: $parent \leftarrow v(WL).parent$
3: $WV \leftarrow$ first window version in WL
4: $v(WV).child \leftarrow v(WL)$
5: $parent.child \leftarrow v(WV)$
6: **end procedure**

Algorithm 6 shows the procedure for adding a window version from the window list to the tree. The first window version is removed from the list (line 2) and its vertex is added as the child of the window list's parent (line 5). The vertex of the window list is added as the child of the newly created vertex (line 4).

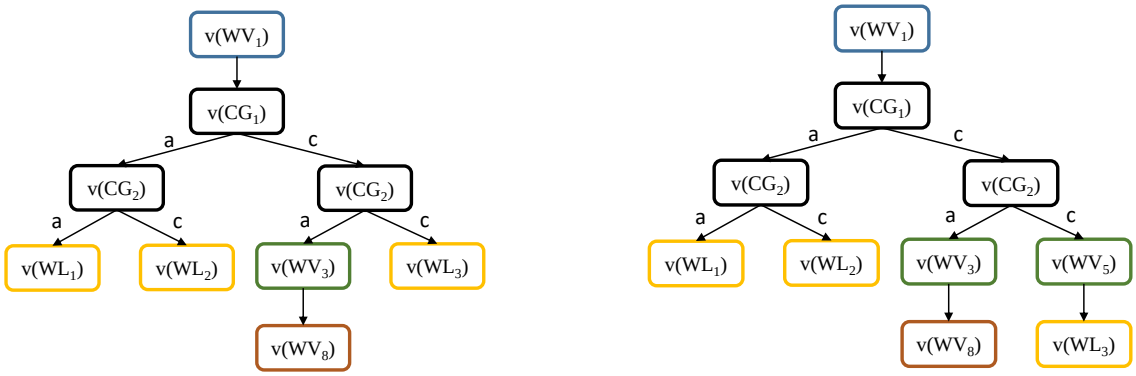


Figure 4.3: Dependency tree (a) before and (b) after addition of WV_5

From fig. 4.3, we can see that once the completion probability of CG_2 becomes high, window version from the window list at the completion edge are added to the tree. Since the consumption group CG_1 has a higher probability of being completed, a window version is added only to the right sub tree. The window list WL_3 now only contains WV_9.

Consumption group completed or abandoned

The outcome of a consumption group is dealt with in the same way as in the existing system. When a consumption group is completed or abandoned, the contradicting edge in the dependency tree is removed along with all vertices in that subtree. Algorithms 7 and 8 show the procedure for when a consumption group is completed and abandoned respectively.

Algorithm 7 Consumption group completed

1: **procedure** CONSUMPTIONGROUPCOMPLETED(ConsumptionGroup CG)
2: $v(CG).abandonEdge \leftarrow null$
3: $parent \leftarrow v(CG).parent$
4: $parent.child \leftarrow v(CG).completionEdge$
5: **end procedure**

If the consumption group is completed, the abandon edge is set to null (Algorithm 7, line 2) and the sub tree at the completion edge becomes the child of the window version that created the completed consumption group. i.e the parent of the consumption group in the tree (Algorithm 7, lines 3, 3). All vertices in the abandon sub tree will later be claimed by the garbage collector.

Algorithm 8 Consumption group abandoned

1: **procedure** CONSUMPTIONGROUPABANDONED(ConsumptionGroup CG)
2: $v(CG).completionEdge \leftarrow null$
3: $parent \leftarrow v(CG).parent$
4: $parent.child \leftarrow v(CG).abandonEdge$
5: **end procedure**

If the consumption group is abandoned, the completion edge is set to null (Algorithm 8, line 2) and the sub tree at the abandon edge becomes the child of the window version that created the completed consumption group. i.e the parent of the consumption group in the tree (Algorithm 8, lines 3, 3). The vertices removed from the tree will be garbage collected.

Figure 4.4 shows the dependency tree before and after the completion of consumption group CG_1. As discussed above, the abandon edge of $v(CG_1)$ is removed and the completion child of $v(CG_1)$ becomes the child of $v(WV_1)$. Note that once WV_9 is added to the dependency tree, the vertex of the window list will not be removed from the tree. The window list will be reused for future window versions. Therefore, window lists are removed from the tree only if the sub tree containing that window list is removed from the tree as in the case shown in Figure 4.4.

4.1.3 Parallel processing of window versions

As seen earlier, the selected top k window versions are processed in parallel. The scheduling of the selected window versions remains the same as in the existing system (ref. Algorithm 2). The

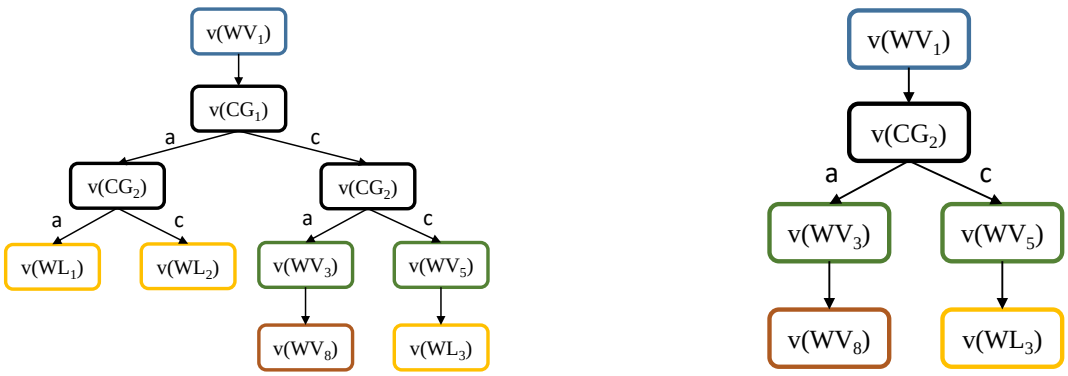


Figure 4.4: Dependency tree (a) before and (b) after completion of CG_1

processing of events in an operator is shown in Algorithm 10. Lines 5 - 39 show one processing cycle for an operator. As each event is being processed by the operator, there are several possible outcomes of the processing. The processed event could complete a partial match (lines 14 - 19), void a partial match (lines 20 - 23) or become part of a partial match (lines 24 - 35). When the processed event completes a partial match, then the associated consumption group is complete. Hence, the dependency tree is updated using the *consumptionGroupCompleted* function (algorithm 7) and the emitted complex events are buffered. If the processed event voids a partial match (the partial match has no possibility of becoming a complete match), then the associated consumption groups should be abandoned. This is done by calling the function *consumptionGroupAbandoned* (Algorithm 8). If the processed event becomes part of a partial match, then the associated consumption groups are updated and if it has no associated consumption groups, then a new consumption group is created and the dependency tree is updated using the function *consumptionGroupCreated* (Algorithm 5).

As the window versions are being processed in parallel and there is no synchronization between the operator instances regarding the progress of the window versions, there may be inconsistencies. For example, an event added to a consumption group might have already been processed by a window version in the completion sub tree of the consumption group. In order to avoid such inconsistencies, a consistency check is done at frequent intervals (Algorithm 10, line 38) and if an inconsistency is found, the window is processed from the beginning.

Algorithm 9 shows the procedure for performing a consistency check. We use a consistency check frequency (line 2) so that a consistency check is not done for every processing cycle. First, we check if the consumption group has been updated since the last consistency check (line 5) and then check for any inconsistencies. If it is found that the window version has processed events in the consumption group that should have been suppressed (line 38), then it is rolled back (line 13).

Algorithm 9 Consistency check

```
procedure PERFORMCONSISTENCYCHECK(processingCounter, currentWV)
    if (processingCounter mod consistencyCheckFreq) == 0 then
        boolean inconsistent ← false
        for each CG ∈ currentWV.suppressedCGs do
            if CG.version ≠ CG.lastCheckedVersion then
                if (currentWV.usedEvents ∩ CG.events) ≠ ∅ then
                    inconsistent ← true
                end if
            end if
            CG.lastCheckedVersion ← CG.version
        end for
        if inconsistent then
            currentWV.rollback
        end if
    end if
end procedure
```

4.2 Comparison of dependency tree with and without lazy loading

The changes made to the maintenance of the dependency tree was discussed in the previous section. Since our work mainly focuses on reducing the number of vertices that are created unnecessarily in the tree and keeping the tree size in check, we compare the tree data structure with and without lazy loading.

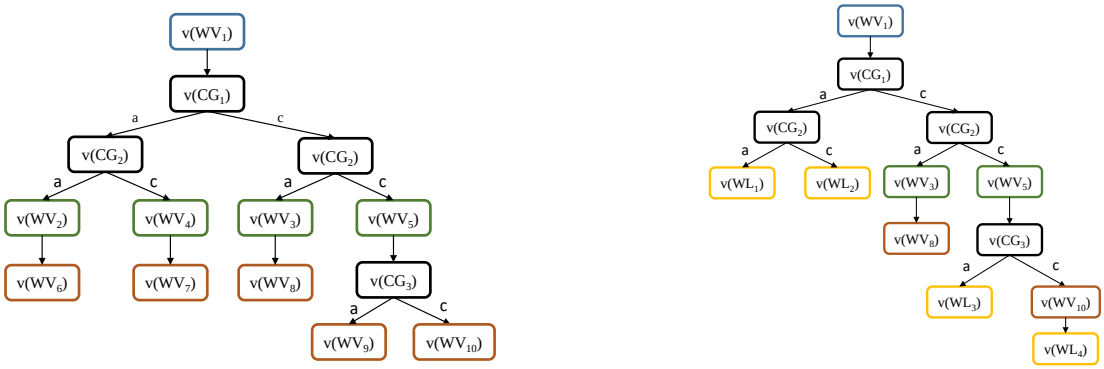


Figure 4.5: Dependency tree (a) without and (b) with lazy loading

From the figures, it is evident that the number of vertices added to the tree has been reduced

Algorithm 10 Processing of events

```
1:  procedure EVENTPROCESSING
2:      WindowVersion currentWV
3:      WindowVersion scheduledWV
4:      int processingCounter ← 0
5:      while true do
6:          processingCounter ← processingCounter + 1
7:          if scheduledWV ≠ currentWV then
8:              currentWV ← scheduledWV
9:          end if
10:         Event event ← currentWV.getNextEvent()
11:         if event ∉ currentWV.suppressedCGs then
12:             Outcome eventOutcome ← process(event)
13:             switch eventOutcome do
14:                 case completes partial match:
15:                     E ← emitted complex event
16:                     CG ← consumption group associated with partial match
17:                     bufferE
18:                     consumptionGroupCompleted(CG)
19:                 end case
20:                 case voids partial match:
21:                     CG ← consumption group associated with partial match
22:                     consumptionGroupAbandoned(CG)
23:                 end case
24:                 case added to partial match:
25:                     if event satisfies consumption policy then
26:                         CG ← consumption group associated with partial match
27:                         if CG = ∅ then
28:                             CG ← {}
29:                             CG.add(event)
30:                             consumptionGroupCreated(CG, currentWV)
31:                         else
32:                             CG.add(event)
33:                         end if
34:                     end if
35:                 end case
36:             end switch
37:         end if
38:         performConsistencyCheck(processingCounter, currentWV)
39:     end while
40: end procedure
```

which in turn translates to the number of vertices that need to be removed once the outcome of a consumption group is known. For consumption group CG_1, once the outcome is known, instead of removing five vertices from the tree, only three need to be removed. Note that, the window versions for any other dependent windows created before CG_1 completes will be kept buffered in the window lists. Also note that, the vertex of an empty window list is not removed from the tree (like in the case of $v(WL_4)$. Instead, it is reused. This avoids a vertex to be removed only to be created again later.

The comparison of the entire SPECTRE system with and without lazy loading for the dependency tree is discussed in the following chapter.

5 Evaluations

In this chapter, we evaluate the entire SPECTRE system with lazy loading for the dependency tree. In the first section, we describe the platform and setup of our evaluation along with the data sets used for evaluation. Then, we describe the system and setup used for evaluation. Finally, we compare the results from the evaluation of our work with the existing work.

5.1 Experimental setup

In this section, we discuss the implementation of the SPECTRE system and the platform that it is run on. In addition, the various queries used for the evaluations and the configurations used for running these queries.

Implementation

The implementation of SPECTRE is in C++. It uses algorithms and data structures that utilize shared memory architecture. SPECTRE is a generic CEP framework where the queries are implemented as user-defined functions (UDF). In order to split incoming event stream into windows, SPECTRE provides interfaces for window open (to specify when to open a window) and close predicates (specifies when to close a window). Since the queries are UDF, the selection and consumption policies are also defined by the user. The data used by the queries are stored in a file. The events are read from this file by a client and sent to SPECTRE using a TCP connection.

Platform

SPECTRE is run on a shared memory multi-core machine with 2x10 CPU cores (Intel Xeon E5-2687WV3 3.1 GHz). The total memory of the machine is 128 GB, the operating system is CentOS and it supports hyper-threading.

Datasets

We use the same datasets and queries that were used to evaluate the original SPECTRE system [14] in order to make the comparison with our work easier. Both datasets are based on the algorithmic trading scenario.

Dataset 1:

The first dataset is based on real world stock trading data from the New York Stock Exchange (NYSE). We refer to this dataset as NYSE Stock Quotes and it is represented as NYSE. NYSE contains real intra day quotes of around 3000 stock symbols from the New York Stock Exchange. This dataset was collected over two months from Google Finance[7] and represents real data for stock market pattern analytics. NYSE contains more than 24 million stock quotes and for each stock symbol, the quotes have a resolution of 1 quote per minute.

Dataset 2:

The second dataset consists of 300 different stock symbols with the probability of each stock symbol equally distributed in a sequence. We generate a random sequence of 3 million events. This dataset is referred to as Random Stock Quotes and denoted as RAND.

Queries

We use two different queries for our evaluations. The queries are written in MATCH-RECOGNIZE notation [17]. The consumption policy and predicate for splitting windows (window size and window start predicate) are specified in TESLA [6]. The CONSUME construct is used to specify the consumption policy and the WITHIN .. FROM construct is used to specify the window start condition and window size.

Query 1:

In query Q1, a complex event is emitted when first q rising (denoted as RE) or falling (denoted as FE) stock quote of stock symbols are detected within ws minutes from the rising or falling stock of a leading stock symbol (denoted as MLE). The leading stock symbol consists of 16 technology blue chip companies. Hence, here the selection policy is ”first MLE, first q RE” i.e. the first q rising stock quote of stock symbols withing ws minutes from the occurrence of a rising quote of stock symbol MLE. The pattern length is always of a fixed size q and as events match the pattern, the completion probability increases. Once a complex event is detected, all events in the pattern are consumed. Therefore, the consumption policy for query Q1 is ”all”.

```
PATTERN (MLE RE1 RE2 ... REq)
    DEFINE
            MLE AS (MLE.closePrice
            > MLE.openPrice)
        RE1 AS (RE1.closePrice
            > RE1.openPrice)
        RE2 AS (RE2.closePrice
            > RE2.openPrice)
        ....
        REq AS (REq.closePrice
            > REq.openPrice)
        WITHIN ws events FROM MLE
        CONSUME (MLE RE1 RE2 ... REq)
```

As mentioned before, the WITHIN .. FROM construct is used to specify the window size and start condition. Hence, for query Q1, a window is opened (created) when a rising or falling stock quote of stock symbol from any of the 16 technology blue chip companies occurs. The window size is defined by *ws* and can be varied.

Query 2:

In query Q2, a complex event is detected when a set of *n* stock symbols following the occurrence of stock symbol A within ws minutes. Here, the *n* stock symbols following A can be in any order. Therefore, the selection policy is ”first A, first *n* X” where X is any stock symbol and the pattern size is *n*. The consumption policy of query Q2 is also ”all” i.e. all events from the detected pattern are consumed.

```
PATTERN (A SET(X1 ... Xn))
    WITHIN ws events
            FROM every s events
    CONSUME(A SET(X1 ... Xn))
```

For query Q2, a window is created for every *s* minutes, where *s* is the sliding size and the window size is defined using *ws*. Both *s* and *ws* can be varied. Note that the value of *s* has an impact on how many events overlap between two consecutive windows and in turn the dependency between them.

5.2 Evaluation approach

As discussed earlier, the goal of this thesis is to decrease the number of vertices created in the dependency tree so that the memory utilization is improved and in turn the load on the garbage

collector is eased. In order to check if the lazy loading technique has had an impact on the number of vertices created, we employ a counter to calculate the number of vertices that are created in the tree. We also employ the same counter for the SPECTRE system without using lazy loading so as to compare the number of vertices that are created in the dependency tree with and without lazy loading.

In addition to this, since the vertices in the tree are lazily loaded i.e. not added to the tree until necessary, we want to check that this does not affect the processing of windows. Since vertices are only added to the tree when necessary, there might be cases where the scheduler might have to wait until the vertex is added to the tree so that it can be scheduled next. Hence, we also calculate and compare the throughput of the system with and without lazy loading.

Further, as the number of vertices created in the dependency tree directly translates to the memory consumption of the system and the size of the dependency tree, we also calculate the memory consumption of the system and tree size and use them for comparison with the original SPECTRE system.

5.3 Performance evaluation

We use the following settings for both versions of the SPECTRE system (using lazy loading and without lazy loading) and for all the evaluations unless specified otherwise. Each window version creates only one consumption group and the threshold multiplier $M = 1$. The markov model is run with $\alpha = 0.7$ and $\ell = 10$.

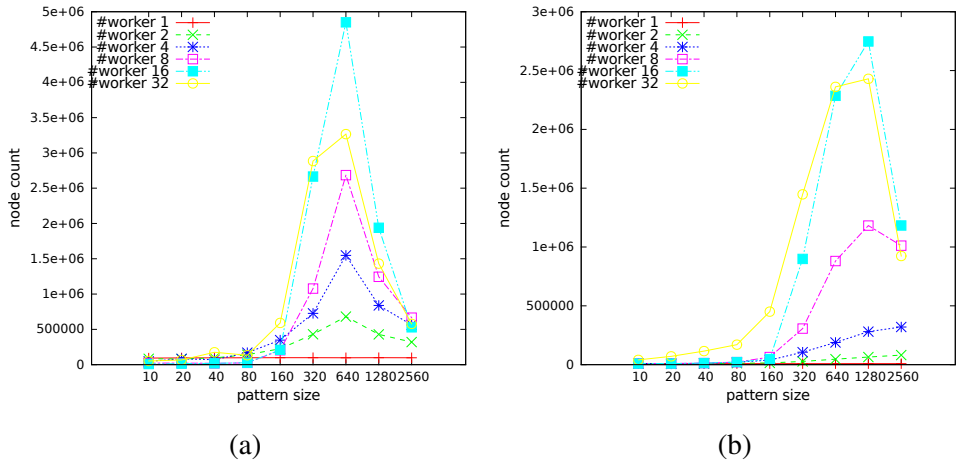
Each experiment was run 10 times and the average of the values for throughput, tree size and node count (in this section, we refer to vertices as nodes) are used. The throughput is the number of events processes per second. The node count refers to the total number of nodes created during run time and tree size refers to the maximum depth of the dependency tree at a given time during the entire run.

For memory consumption, we only take into account the maximum memory consumption over the course of experiment for each run. Finally, an average of the maximum memory consumption is calculated from the 10 runs.

Configuration 1

In this configuration, we use query Q1 with varying pattern sizes i.e. the size of q. The window size *ws* is 8000 events and q = 10, 20, 40, 80, 160, 320, 640, 1280 and 2460. We call this configuration C1.

Figure 5.1 shows the comparison of node count when query Q1 was run with the configuration C1. Figure 5.1a shows the node count values for varying pattern sizes without lazy loading and



(a) (b)

Figure 5.1: C1: Node count for the SPECTRE system (a) without and (b) with lazy loading

fig. 5.1b shows the node count values with lazy loading. It is evident from the two figures that the number of nodes created is significantly reduced when using lazy loading.

In a window of 8000 events, a smaller pattern size has a higher probability of being found. Hence, for smaller pattern sizes the completion probability (probability that the entire pattern will be found) is higher and as the pattern size increases, the completion probability decreases. This is due to the way the window versions are scheduled for execution.

When the completion probability is high (smaller pattern size), the window versions will be scheduled for execution from the right sub tree i.e. windows are scheduled depth wise. This means that window versions of different windows are processed.

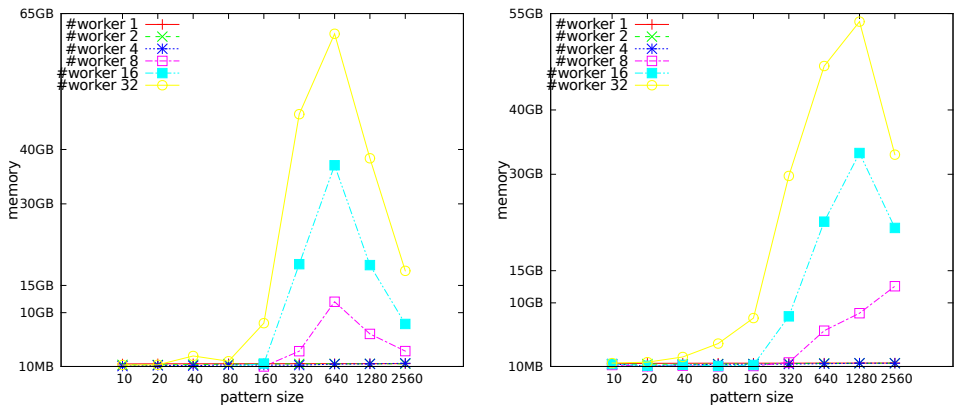
Comparing with the original system, the number of nodes for smaller probabilities are significantly reduced because, in our case, the left sub tree will always be a window list (since completion probability is high, window versions will be created at the completion edge). In addition, since we always create a window list when the sub tree is cloned to be added at the completion edge, even if there is a delay in knowing the probabilities, window versions would not have been erroneously created.

However, for higher pattern sizes, the completion probability is low. In this case, the scheduling is once again done depth wise but window versions from right sub tree are scheduled for processing. From fig. 5.1b, the node count values for pattern sizes 1280 and 2560 are slightly higher because, as mentioned earlier, since we always create a window list in the right sub tree when the left sub tree is cloned, window versions should be added to the tree when they need to be processed and rarely, if a pattern is found, then all nodes in the left sub tree would already have been created and the window lists in the right sub tree would also add to the total node count.

From figs. 5.1a and 5.1b, the interesting points to note are the node count values for pattern size 640. Here the completion probability is 50% [14]. The created consumption groups have equal

probability of being completed or abandoned and the scheduling is done breadth wise (window versions of the same window are scheduled for execution). However, only one window version for a particular window can survive. Therefore the number of nodes created are very high without lazy loading.

Figure 5.2 shows the memory consumption for the SPECTRE system. Since, the number of nodes created translates directly to the amount of memory consumed, it can be seen that the memory consumption is lower when using lazy loading. As seen previously in the case of node count, for worker thread/pattern size 16/640, the node count value without lazy loading is close to 5 million and the memory consumption is close to 40GB. The same values for worker thread/pattern size 16/640 with lazy loading are close to 2.5 million (half of the value with lazy loading) and the memory is close to 20GB.

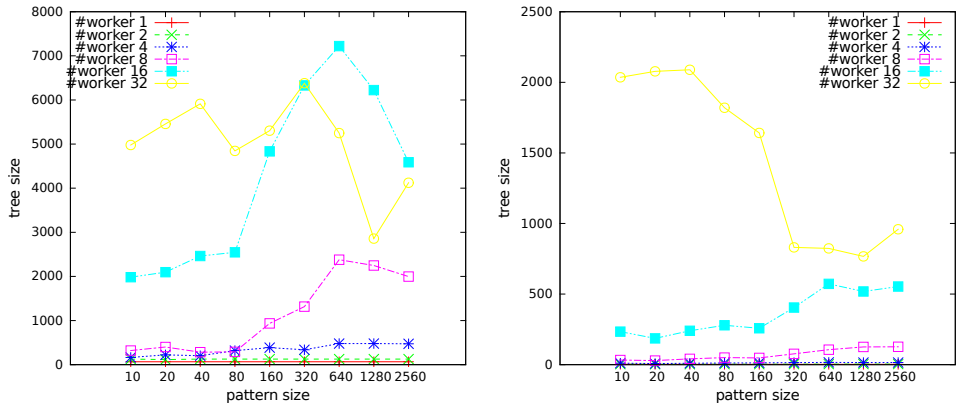


(a) (b)

Figure 5.2: C1: Memory consumption for the SPECTRE system (a) without and (b) with lazy loading

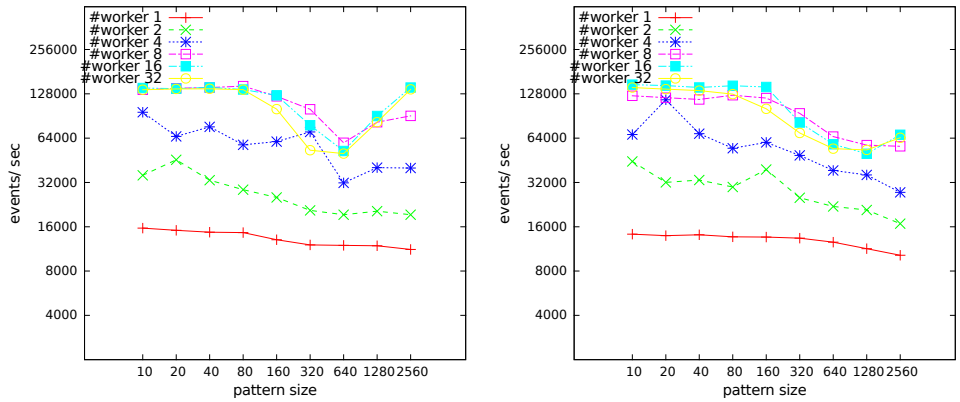
Figure 5.3 shows a comparison of the maximum tree size with and without lazy loading. From fig. 5.3, the tree size when the dependency tree is lazy loaded is lesser than when lazy loading is not used. As discussed earlier, a window list is created whenever the left sub tree is cloned. For smaller pattern sizes, since the completion probability is high, window versions from the list will always be added to the tree and as the window list will not be removed (to be reused later), it contributes to the increase of the tree size. In the case of larger pattern sizes, this is not a problem as the left sub tree will survive and window versions from the list in the right sub tree will never be added to the tree.

Figure 5.4 shows the throughput i.e. number of events processed per second when the tree is lazy loaded and when it is not. It can be seen that the throughput remains almost the same as when the tree is not lazy loaded. Therefore, the throughput is not affected even though the vertices are added to the tree only when needed to be processed.



(a) (b)

Figure 5.3: C1: Tree size for the SPECTRE system (a) without and (b) with lazy loading



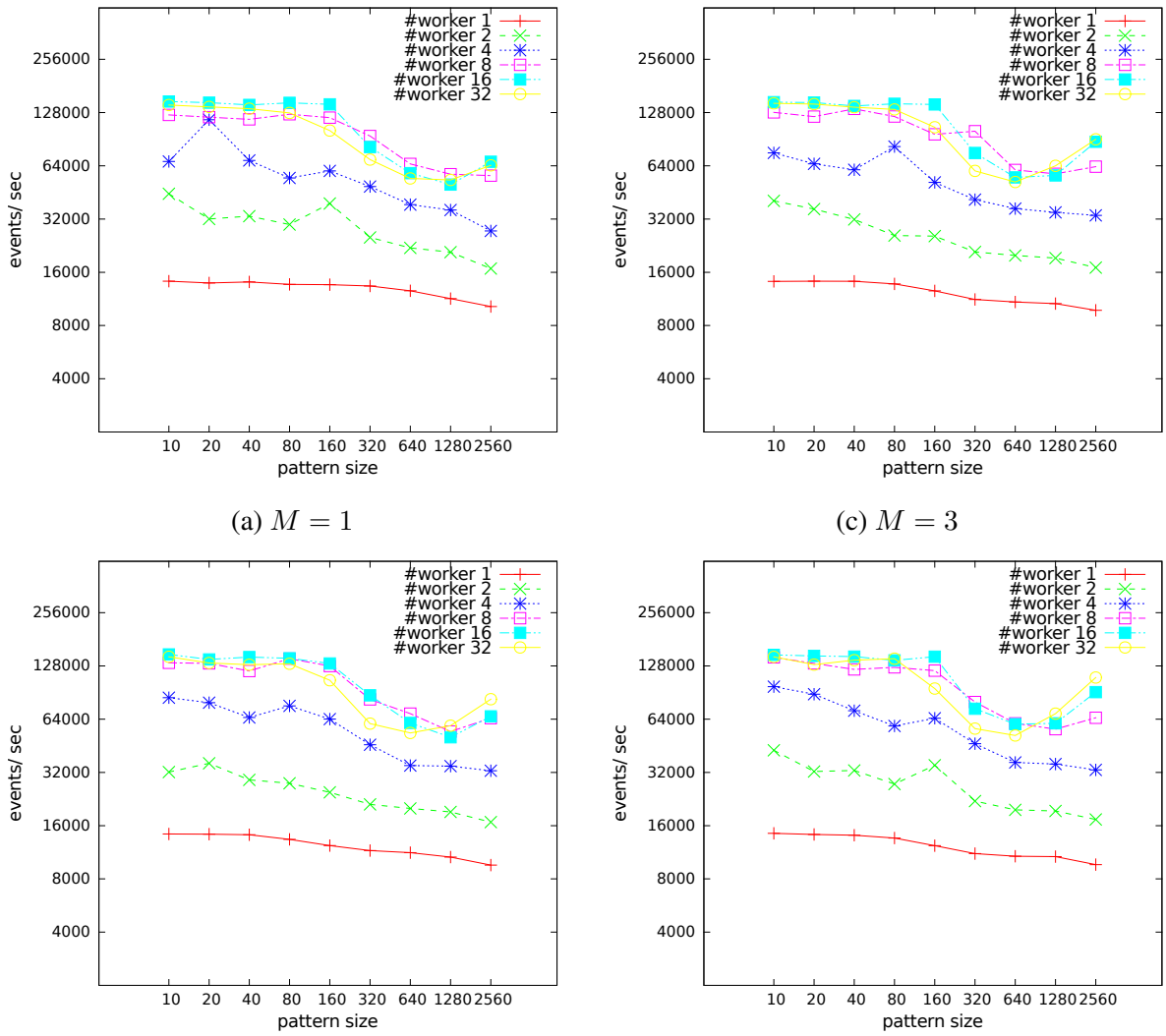
(a) (b)

Figure 5.4: C1: Throughput for the SPECTRE system (a) without and (b) with lazy loading

Effect of threshold multiplier

In order to analyze the effect of threshold multiplier on tree size and throughput, we ran the above experiment for different values of $M = 2, 3$ and 4.

The threshold multiplier has a very minimal effect on the throughput. As discussed earlier, the throughput remains almost the same for the SPECTRE system with or without lazy loading. This is because, even though the load on the garbage collector has been decreased, the scheduler might have to wait until a window version has been added to the tree in order to begin processing of the window. Hence, from fig. 5.5, we can see that the throughput remains almost the same in all four cases with only slight deviations.



(a) $M = 1$

(c) $M = 3$

(b) $M = 2$

(d) $M = 4$

Figure 5.5: Throughput of the SPECTRE system with lazy loading for different values of M

From the graphs in fig. 5.5, we can see that the throughput gets better for worker thread/pattern size 32/2560 for higher threshold multiplier.This is because since the probabilities of the top n window versions are calculated, they are already added in the tree. Since for higher pattern sizes, the abandon probability (probability that the consumption group will be abandoned) is more and the nodes already in the tree are the ones that will be a part of the abandon sub tree, lazy loading gives a better throughput for higher threshold multiplier when the pattern size is large.

In order to analyze the effect of the threshold multiplier on the tree size, we consider the tree size for different pattern sizes for 32 worker threads. Figure 5.6 shows the different tree size

values for different threshold multiplier values. We can see from the graph that the tree size increases as the threshold multiplier increases. As we choose our threshold probability to be the probability of the n_{th} window version, where $n = k \times M$, a higher value of M would mean that the probability of the n_{th} window version would be very low. Hence, nodes from the window list can be added to the tree for a lesser probability value.

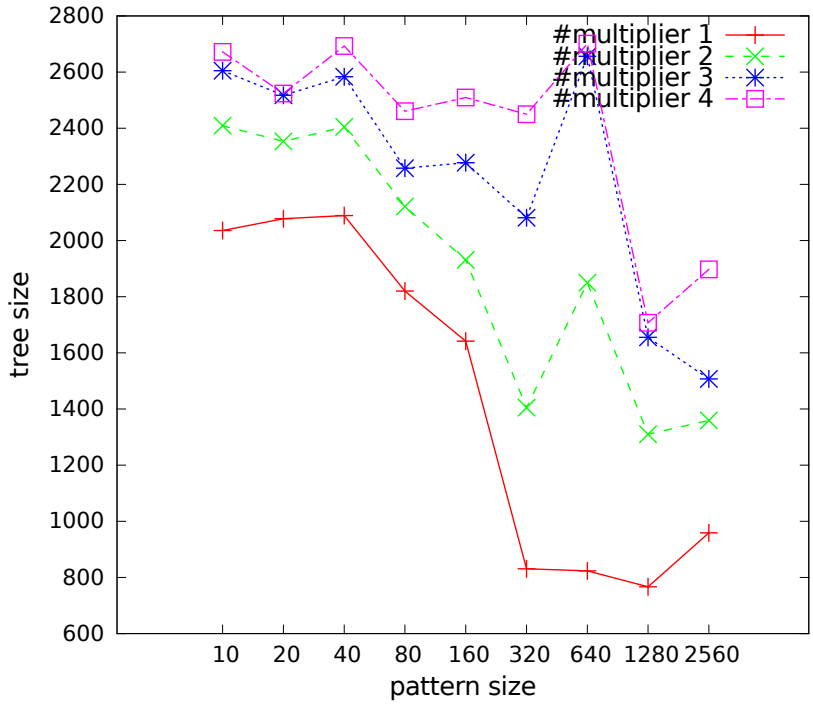


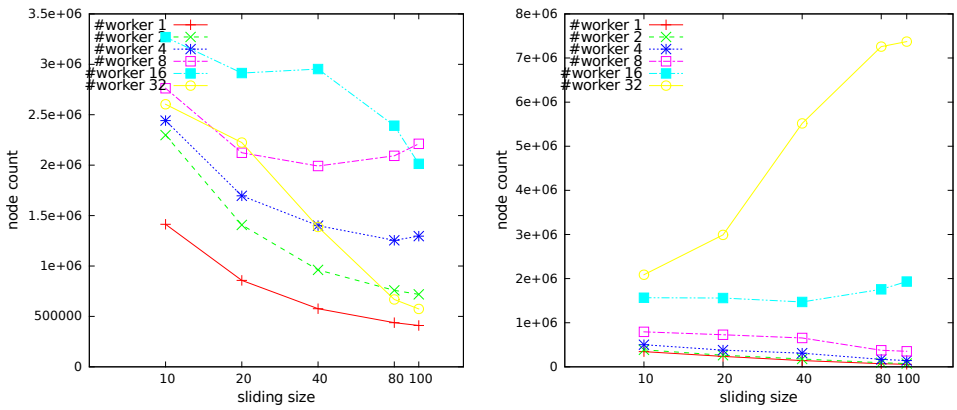
Figure 5.6: Tree size for different pattern sizes and threshold multipliers

Here, we concentrate more on pattern size 640, since consumption groups have equal probability of being completed or abandoned and SPECTRE without lazy loading does performs not perform very well for pattern size 640 in terms of node count and tree size. From fig. 5.6, we can see that for pattern size 640, the peak in tree size for threshold multipliers $M = 2, 3, 4$ is similar to that of the SPECTRE system running without lazy loading though not as high. Hence, a threshold multiplier of 1 is best for obtaining optimal tree size.

Configuration 2

In this configuration C2, we use Q2 with a pattern size of $q = 2$, window size of 1000 events $ws = 1000$ and varying sliding size $s = 10, 20, 40, 80, 100$. As the sliding size increases, the number of windows created decreases and so does the dependency between windows. For example, for a sliding size of 10, two consecutive windows all events in common between them

except for 10 events and a window is created for every 10 events. We performed this experiment to determine how the system would handle a large number of windows being created and if they have a huge dependency between them.



(a) (b)

Figure 5.7: C2: Node count for the SPECTRE system (a) without and (b) with lazy loading

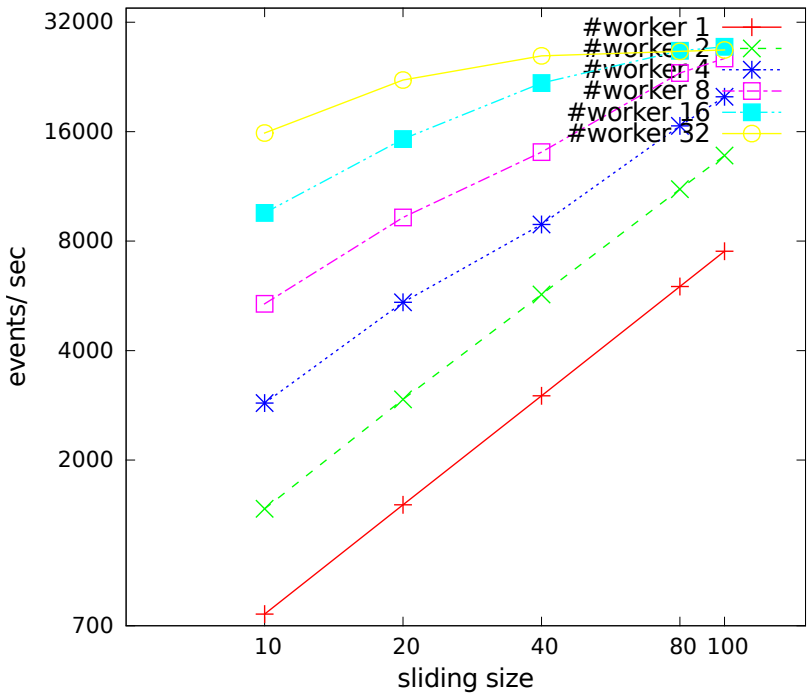


Figure 5.8: C2: Throughput for SPECTRE system with lazy loading

From fig. 5.7, we can that the number of nodes created were considerably low when compared to the system not using lazy loading. From previous experiments, we have seen that lazy

loading performs especially well in the case of small pattern sizes, hence, the sliding size didn't seem to have a very big effect on the number of nodes being created. The system without lazy loading however seemed to create a large number of nodes. In fig. 5.7b, the node count decreases as the sliding size increases. Therefore, it can be seen that for a higher overlap of events, the number of nodes created increases. Since large number of windows are created and they have a lot of overlap between them (window versions have a lot of consumption groups in common and in turn leads to the creation of more window versions), the number of nodes created are high for small sliding sizes and decreases as the sliding size increases. Worker thread of size 32 seems to be an anomaly in both cases.

In addition to the node count, we also check the throughput of the SPECTRE system with lazy loading for this configuration. From fig. 5.8, we can see that the system scales well in terms of throughput. In the beginning, the throughput is low as there are many windows being created and as the number of windows decreases i.e. sliding size increases, the throughput keeps increasing and is the highest for sliding size 100 and lowest of sliding size 10 for all worker thread configurations.

6 Conclusion

With this work, we optimize the memory allocation and utilization of the existing SPECTRE system. First, we discussed the existing SPECTRE framework and its working in detail and discussed the need for optimal use of memory. Since all the components access the shared memory for information about the dependency tree and if this data structure is huge, this would cause a bottleneck in memory. Therefore, we focused on reducing the tree size. From the working of the SPECTRE system, it was seen that the main problem was that too many nodes were created in the tree only to be removed later. Hence, we wanted to reduce the number of nodes that were unnecessarily created. Hence, lazy loading of the tree was proposed. Here, the windows are kept buffered in a list instead of adding all of them to the tree directly. Since the number of nodes that needed to be created and the number of nodes that are destroyed in the tree depends directly on the completion probability of the consumption groups in the tree, the same probability was used to determine when a node can be added in the dependency tree. For this, we calculate a threshold value for the probability - called threshold probability - which is considered as a threshold for the probabilities of the windows. Hence, we reduce the number of nodes that are created in the tree by buffering them and adding them to the tree only when necessary.

Another issue was the load on the garbage collector. Since many nodes were created, once they are known to be invalid, they have to be garbage collected. This posed a load on the garbage collector and in turn utilized valuable system resources for garbage collection. Hence, solving the node creation problem would in turn ease the load on the garbage collector.

For the analysis of our work, we directly calculate the number of nodes created in the dependency tree and the tree size for the original SPECTRE framework (without lazy loading) and our work (SPECTRE with lazy loading). From the evaluations, we can see that the SPECTRE system with lazy loading creates significantly lesser number of nodes in the dependency tree as opposed to without lazy loading. The tree size is also smaller in the case of lazy loading. In addition, the memory consumption is also significantly reduced which in turn translates to easing the load on the garbage collector. It is also seen that lazily loading the tree does not affect the system throughput.

We also found that the SPECTRE system with lazy loading performed very well where the original system was lacking - when the outcome of the completion probability is not known (when completion probability is 50%). In this case, a large number of nodes were created by the original system and erroneously processed only to later be removed from the tree. Here, the SPECTRE system with lazy loading showed the highest optimization as it depends on the

completion probability to determine which sub tree should be processed (instead of processing windows breadth first). It was also seen that the lazy loading could handle large number of windows being created and would optimize memory usage.

For our future work, we would like to focus on getting better results with lazy loading for large pattern sizes. Since the system now creates a window list whenever the left sub tree is cloned, the vertex of the window list adds to the node count resulting in a larger number than expected. From the evaluations, it is also seen that while choosing the value of the threshold probability had inverse effects on the tree size, it showed better results for larger pattern sizes. Hence, we would like to investigate more into the choosing of the threshold probability so that a better system performance can be achieved overall and not just for smaller to medium pattern sizes.

Bibliography

[1] Asaf Adi and Opher Etzion. "Amit - the situation manager". In: *The VLDB Journal The International Journal on Very Large Data Bases* 13.2 (2004), 177–203. DOI: 10.1007/s00778-003-0108-y.

[2] Cagri Balkesen et al. "RIP:Run-based Intra-query Parallelism for scalable complex event processing". In: *Proceedings of the 7th ACM international conference on Distributed event-based systems - DEBS 13* (2013). DOI: 10.1145/2488222.2488257.

[3] Pramod Bhatotia et al. "Slider: Incremental Sliding Window Analytics". In: *Proceedings of the 15th International Middleware Conference*. Middleware '14. Bordeaux, France: ACM, 2014, pp. 61–72. ISBN: 978-1-4503-2785-5. DOI: 10.1145/2663165.2663334. URL: http://doi.acm.org/10.1145/2663165.2663334.

[4] Alejandro Buchmann and Boris Koldehofe. "Complex Event Processing". In: *it - Information Technology* 51.5 (2009). DOI: 10.1524/itit.2009.9058.

[5] S. Chakravarthy and D. Mishra. "Snoop: An expressive event specification language for active databases". In: *Data & Knowledge Engineering* 14.1 (1994), 1–26. DOI: 10.1016/0169-023x(94)90006-x.

[6] Gianpaolo Cugola and Alessandro Margara. "Tesla: A Formally Defined Event Specification Language." In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems - DEBS 10* (2010). DOI: 10.1145/1827418.1827427.

[7] *Google Finance*. URL: https://www.google.com/finance.

[8] Martin Hirzel et al. "A catalog of stream processing optimizations". In: *ACM Computing Surveys* 46.4 (2014), 1–34. DOI: 10.1145/2528412.

[9] IBM Knowledge Center. *What is event processing?* URL: https://www.ibm.com/support/knowledgecenter/en/SSGMCP_4.2.0/com.ibm.cics.ts.eventprocessing.doc/concepts/dfhep_definition.html.

[10] Sachini Jayasekara et al. "Wihidum: Distributed complex event processing". In: *Journal of Parallel and Distributed Computing* 79-80 (2015), 42–51. DOI: 10.1016/j.jpdc.2015.03.002.

[11] David Luckham. *Power of events: an introduction to complex event processing in distributed enterprise*. Addison-Wesley Educationa, 2013.

[12] Ruben Mayer, Boris Koldehofe, and Kurt Rothermel. "Predictable Low-Latency Event Detection With Parallel Complex Event Processing". In: *IEEE Internet of Things Journal* 2.4 (2015), 274–286. DOI: 10.1109/jiot.2015.2397316.

[13] Ruben Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. "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* (2017). DOI: 10.1145/3093742.3093914.

[14] Ruben Mayer et al. "SPECTRE: Supporting Consumption Policies in Window-Based Parallel Complex Event Processing". In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference on - Middleware 17* (2017). DOI: 10.1145/3135974.3135983.

[15] WallStreet & Technology. *Deciphering the myths around complex event processing*. URL: http://www.wallstreetandtech.com/latency/deciphering-the-myths-around-complex-event-processing/d/d-id/1259489?.

[16] Wikipedia. *Markov chain*. URL: https://en.wikipedia.org/wiki/Markov_chain.

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

[18] D. Zimmer and R. Unland. "On the semantics of complex events in active database management systems". In: *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337)* (1999). DOI: 10.1109/icde.1999.754955.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature