Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Window Drop Load Shedding in Complex Event Processing

Albert Flaig

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	M.Sc. Ahmad Slo
Commenced:	April 26, 2018
Completed:	October 26, 2018

Abstract

In Complex Event Processing (CEP) huge input event streams are processed to interpret specific situations in real time. In the field of parallel CEP the event stream is split into different windows which can be processed in parallel by several operator instances. This paradigm reduces the load imposed on a single operator instance and allows for horizontal scalability. However, in case of high load, the operator instances may not be able to process the incoming events in time; the unprocessed events are queued up resulting in a higher processing latency. In such situations it can be desirable to impose a latency bound on the system. One way to satisfy the given latency bound, is to do load shedding by dropping incoming windows.

This problem is not trivial, as it is unclear when, which and how many windows need to be dropped. To answer these questions we try to find out the most promising windows, e.g. windows which may yield a high amount of complex events, yet impose a low impact on processing time. However, this adds additional challenges, as the processing latency of a window depends on many unknown variables, such as the size of the window, the types of incoming events, the position of these events relative to each other and even on the processing latency of other overlapping windows. Furthermore, even if the quality and processing latency of a window is determined, there are several open questions regarding the timing and frequency of load shedding in order to not violate the latency bound, but still keep enough windows active.

In the scope of this thesis, we introduce a latency and quality model to estimate the processing latency a window induces. Based on this model, we propose an algorithm which decides the windows to drop in case of high system load to satisfy a given latency bound while minimizing the loss of quality.

Kurzfassung

Beim Complex Event Processing werden große Datenströme verarbeitet um daraus bestimmte Situationen in Echtzeit herzuleiten - sogenannte komplexe Ereignisse. Beim Distributed Complex Event Processing wird der Datenstrom in seperate Fenster unterteilt die dann parallel von verschiedenen Operatoren verarbeitet werden. Dieses Paradigma dient dazu die Last auf einem einzelnen Operator zu reduzieren und ermöglicht dadurch horizontale Skalierbarkeit. Im Falle von hoher Last in kurzer Zeit, kann es jedoch dazu kommen dass die Operatoren die Ereignisströme nicht in hinehmbarer Zeit abarbeiten können. In solchen Situationen kann es sinnvoll sein eine Latenz-Obergrenze zu definieren die das System erfüllen muss. Eine Möglichkeit diese Latenz-Obergrenze zu gewährleisten ist mit Lastabwurf - das abwerfen von ganzen Fenstern.

Dieses Problem ist nicht trivial, da es nicht klar ist, welche und wie viele Fenster abgeworfen werden müssen. Um diese Frage zu beantworten, versuchen wir die aussichtsreichsten Fenster zu finden, also die Fenster welche am ehesten komplexe Ereignisse detektieren und welche am wenigsten Last erzeugen. Dies erzeugt jedoch weitere Herausforderungen, da die Last und Qualität eines Fensters von vielen unterschiedlichen Faktoren abhängt, die teils unbekannt sind. Beispielsweise, die Größe eines Fensters, die Typen der einkommenden Ereignisse, die Positionen der Ereignisse relativ zueinander und die Last eines Fensters. Selbst wenn diese Faktoren klar wären, stünde noch die Frage, welche und wie oft Fenster abgeworfen werden müssen um die Latenz-Obergrenze zu erfüllen.

Im Umfang dieser Arbeit, stellen wir eine Methode zur Bewertung von Fenster vor. Mithilfe dieser Methode, zeigen wir einen Algorithmus, welcher entscheidet welche Fenster im Falle von hoher Last abgeworfen werden sollen um einen möglichst geringen Verlust von komplexen Ereignissen in Kauf zu nehmen.

Contents

Acronyms

CE Complex Event. 2

CEP Complex Event Processing. iii, 1–4, 10, 11, 17

complex event A completed pattern consisting of several events. Sometimes also referred to as *composite event*.. 3, 4

consumption policy The consumption policy defines which events are consumed after having participated in a complex event detection.. 6, 10

count window A window which limits to a certain amount of most recent events to be considered for pattern detection.. 4

DCEP Distributed Complex Event Processing. 2, 9, 10

event The simplest form of data in a Complex Event Processing (CEP) paradigm. Sometimes also referred to as *atomic event* or *primitive event*.. 3

MRP Markov Reward Process. 33

time window A window which limits the amount of events to be considered for pattern detection based on a time frame.. 4

window A window which associated with a window operator and contains multiple events.. 2

Chapter 1

Introduction

1.1 Background and Motivation

In today's highly connected world, there is a constant exchange of data. An ever increasing amount of sensor systems, enterprise applications and smart devices collect immense data of all kinds. Such data can be changes in stock market, sensor readings about the physical world, or generally speaking, data of a dynamic system.

In this area, there is a demand for processing data in real time to enable the possibility for an immediate response to specific situations. For instance a smoke detector measures the carbon dioxide concentration in the air and contains a micro controller for processing the sensor readings which then decides whether to trigger an alarm or not.

This is an example where the paradigm of *Complex Event Processing (CEP)* can be applied. In this paradigm, an event represents a change of the state of a dynamic system. Using many events which occur at different moments of time, more information about the dynamic system can be gathered, i.e. a complex event can be detected. A complex event is detected by inferring information from a number of simpler, basic events, which have happened at different moments of time in the past but are in relation to each other. It can be seen as a form of pattern recognition.

In contrast to relational database systems where queries are inquired for data which is already stored and efficiently indexed, in CEP the query in the form of pattern recognition is inquired on events as they arrive in real-time. As such, the system has no knowledge of past events but still needs to limit the events which might be eligible for the pattern recognition. E.g., for the smoke detector it would not make sense to detect a pattern consisting of events which happened weeks apart. In the context of

CEP this is realized by the concept of windows. Whenever a new pattern is started, a window is defined for the pattern which limits the scope. There are different types of windows. A time based window considers only events which happened a certain amount of time after the window was opened. A count window considers only a certain amount of events which happened after the window was opened.

In the case of the smoke detector, the micro controller retrieves the sensors readings in discrete time steps. Each retrieval is an event as it signalizes a change in carbon dioxide concentration at a specific moment of time. To avoid false alarm, the smoke detector should only trigger when there is a continuous increase of carbon dioxide concentration within a time window of one hour. Therefore, the system needs to detect a pattern consisting of several successive events with increasing values. Whenever such a pattern is detected, a *Complex Event (CE)* is fired, signalizing a situation occurring and thus allowing the smoke detector to set off an alarm.

This example outlines the importance of taking immediate action depending on the state of a dynamic system. As such, there is no surprise that there are many different CEP frameworks available [FTR+10], each with their own strengths. There is an ongoing effort being made to improve on such systems in order to be able to handle more data.

One possibility to increase throughput is through the use of Distributed Complex Event Processing (DCEP). In this paradigm, the load on the system is distributed on multiple parallel working machines. This paradigm, however, brings its own set of challenges. Depending on the type of events and the patterns that need to be recognized, there can be dependencies between the order of the events. This would mean that the machines need to be communicating with each other in order to not detect complex events wrongfully. Another challenge of DCEP is effective load balancing. When you have several machines, then it is important to keep all of them busy to maximize resource utility.

However, sometimes even such techniques are not enough to process all incoming data in time, e.g. in the case of a load spike which transcends the systems processing capabilities. In such a case, there are two options how to react. One option is to queue all incoming events. However, this would mean that the system will be processing data which might be out of date and thus not relevant anymore. The other option is load shedding.

There are different possibilities for load shedding. One approach is by sampling: The incoming data is uniformly dropped to accommodate throughput. For example, if there is twice as much data as the system is able to process, then the data is cut in half. E.g. in the case of the smoke detector, we would ignore every other event.

Another approach is using data based dropping in which the system gives certain data more importance than other based on different criteria. Possible criteria can be how recent the data is, how much processing cost it induces and how likely it is that processing the data will lead to detection of complex events. In the scope of this thesis we investigate this approach.

1.2 State of the Arts

In this section, we go more into detail how CEP works and explain related concepts. We establish the state of the arts in this field by looking into related work previously done in this field.

1.2.1 Complex Event Processing

In section 1.1 we already gave an overview on basic concepts of CEP. As already established, an event is the most simple data point. In literature it is sometimes referred to as *atomic event* or *primitive event*. During this thesis we do not use these terms and stick to the term *event*.

A typical event has a specific data structure containing an event type, event timestamp, event id and event payload [Hed17]. In systems more complex than the smoke detector example given in section 1.1, there can be many different event types. The event type is used to categorize a set of changes in a dynamic system which have the same type of change. An example of an event type can be *change in stock market price*. In the case of the smoke detector, it would only have one event type which would be *change in carbon dioxide concentration*.

The event timestamp is the moment of time the event happened or when the change has been detected. However, the system can also assign addtional timestamps to each event during different phases of pattern detection, e.g. event arrival timestamp, event processed timestamp and so on. The event id is a monotonically increasing number for identifying events. The event payload is the actual value of the sensor reading or change of state.

In Complex Event Processing we try to detect a pattern out of an incoming stream of events which results in a complex event. As such, a complex event is a composite of multiple events. Because of this, in literature, complex events are sometimes also

referred to as *composite events* [CJ09]. However, during this thesis we stick to the term *complex event*.

A CEP engine goes through several phases before detecting a complex event. Figure 3.1 illustrates the structure of such a CEP engine. At first, the incoming data is preprocessed by the *preprocessor*. Here, the incoming data stream is deserialized into concrete event instances. At this point, the events are also assigned arrival timestamps. After the events have been deserialized, the preprocessor decides if a window needs to be opened. For each opened window, a *window operator* is spawned. The *window operator* processes all events in the context of one specific window and detects the pattern which needs to be recognized. If a pattern was completed, the window operator then proceeds to the *Complex Event Generator* which generates the complex event.

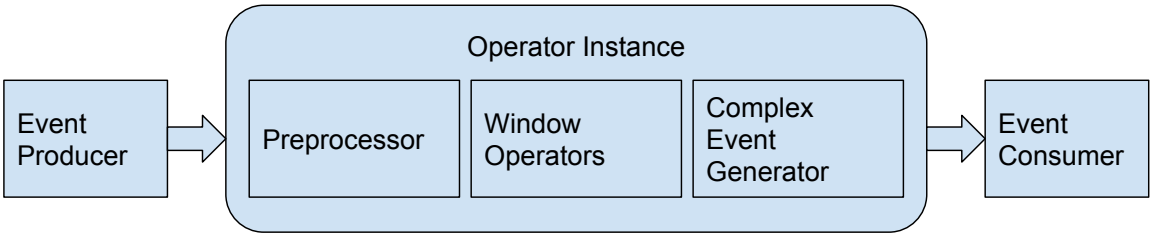


Figure 1.1: The typical architecture of a CEP engine.

The decision when to open and close windows depends on the window strategy. There are two main window strategies: time windows and count windows. Whenever a time window is opened, the window is assigned a remaining time. After the time is run out, the window is closed. During the time the window is open, all incoming events are considered for the pattern detection. The count window is similar, but instead of remaining time we have a remaining count. After the defined amount of events have been processed, the window is closed. In other words, a time window has a fixed duration while a count window has a fixed amount of events.

This is illustrated in Figure 1.2. In this example we consider a window length of three. Therefore, for the count window strategy, the window encompasses the next three successive events. For the time window strategy, the window encompasses all events which are registered in the next three time units.

Additionally to the window strategy, the window slide needs to be defined. The window slide defines when the next window is opened after the previous. If we use a slide size of 1, for the time window, a new window is opened after each unit of time has passed. For the count window, each successive event will open a new window. This difference is illustrated in Figure 1.3. Notice, if we use the count window strategy

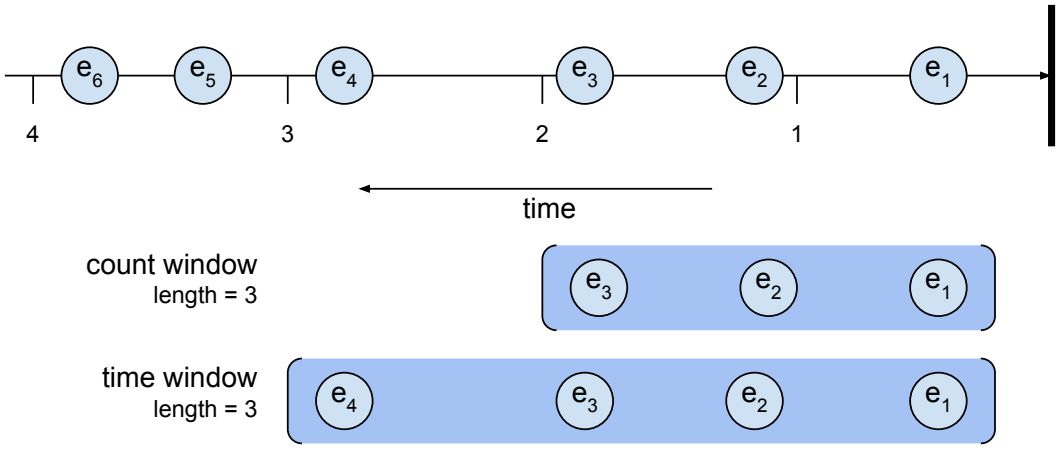


Figure 1.2: Comparison between a count window and a time window with a length of 3.

then there is a relationship between the window slide and the window length. By dividing the window length by the window slide, we know how many active windows there are at any point of time.

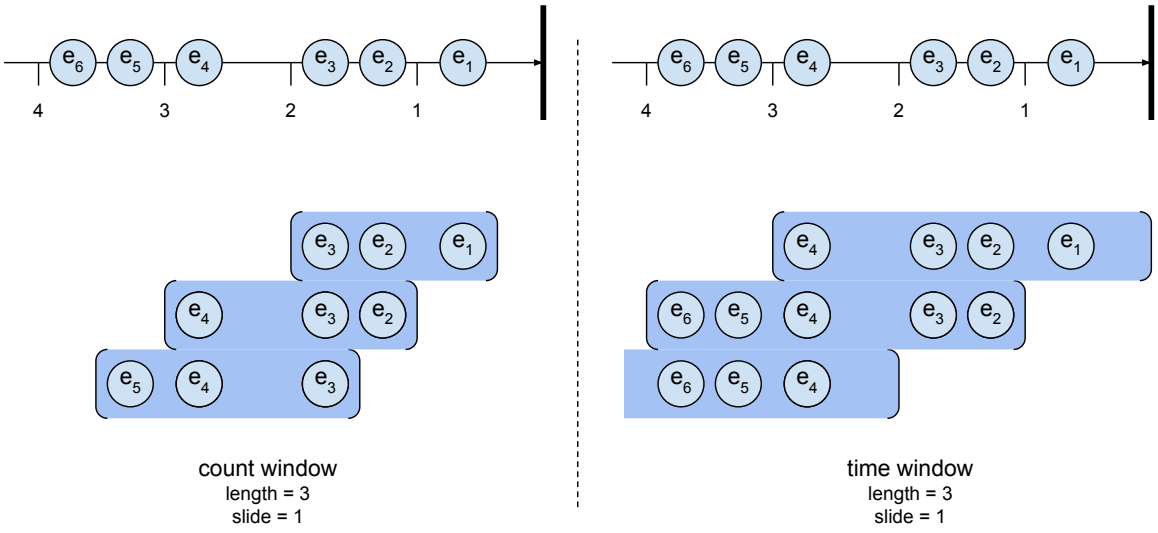


Figure 1.3: Comparison of window slide in a count window and in a time window.

The pattern recognition logic happens in the *window operator*. The window operators process all incoming events, but each window operator processes from the context

of its own specific window. They store the current progress and state of the pattern recognition logic in the form of *partial matches*. We describe the functionality by the example shown in Figure 1.4.

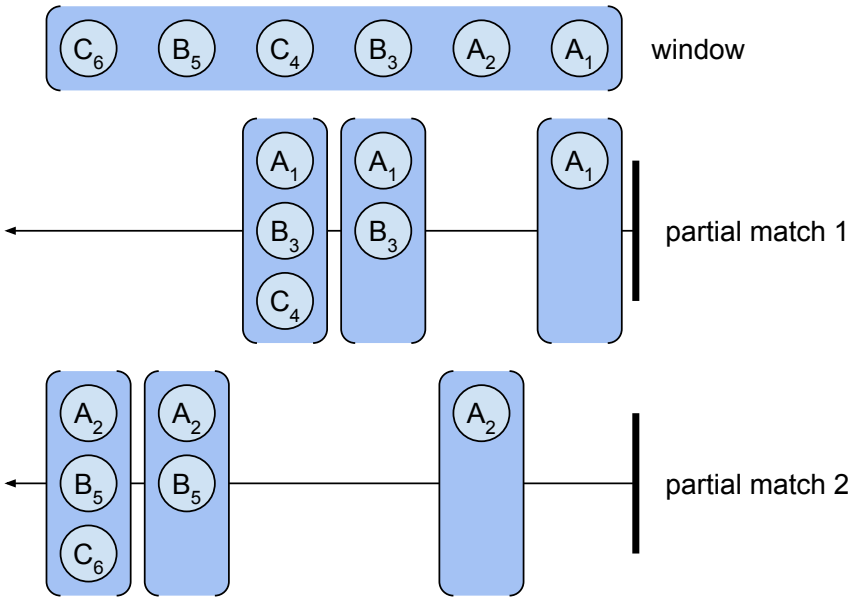


Figure 1.4: How a window operator creates partial matches following the consumption policy `chronicle`.

Consider a window operator which shall detect all occurrences of the pattern (A, B, C) with $\{A, B, C\} \in$ `eventTypes`. In this example, we denote A_1 as an event of type A with id 1. The window of the window operator contains 6 events which come in the order $(A_1, A_2, B_3, C_4, B_5, C_6)$. As the first event A_1 is processed, the window operator matches the first event type A of the pattern and creates a new partial match. The next incoming event A_2 does not match with an existing partial match, but matches with the first event type, another partial match is created. The window operator continues this process until a pattern within a partial match is fully matched. When this happens, a new complex event has been detected. At the end, two complex events are detected which consist of (A_1, B_3, C_4) and (A_2, B_5, C_6).

This example demonstrated the use of the consumption policy `chronicle`. *Consumption policies* are techniques with which the amount of detected complex events can be adjusted. As implied by the name, these policies decide when an event is consumed by the pattern recognition mechanism of a window operator. There are many different consumption policies, some of which are [Hed17]:

- **`Chronicle`** As already mentioned, this consumption policy is illustrated by Figure 1.4. This consumption policy creates a new partial match for each initial event which is not part of another partial match and only considers events for the partial match which have not been included in another partial match. The events of the detected complex events are all disjoint: $(A_1, B_3, C_4), (A_2, B_5, C_6)$.
- **`Regular`** Is a simple consumption policy in which only one partial match is maintained at any point of time. Therefore, the only detected complex event would be (A_1, B_3, C_4).
- **`Unrestricted`** the pattern matching mechanism never consumes any events and always considers all events which happened thus far for a complex event detection. In the previous example we would detect the set $(A_1, B_3, C_4), (A_1, B_3, C_6), (A_2, B_3, C_4), (A_2, B_5, C_6)$.

Window strategies and consumption policies have some things in common. A strict consumption policy reduces the amount of detected complex events. Opening many windows increases the amount of detected complex events. Both techniques can be used in combination to come to the same effect using a different combination of window strategy and consumption policy. For example, if we define the window strategy which opens a new window on each event type A and closes it at the next event type C and use the consumption policy `regular`, then we have essentially the same result as in the previous example. Both possibilities would end up with the same set of detected complex events.

1.2.2 Distributed Complex Event Processing Framework

The thesis builds on top of the Distributed Complex Event Processing (DCEP) framework provided by the Distributed Systems department at the University Stuttgart. The framework is written in Java and consists of four applications:

- **Source** - Generates simple events and sends them to the Splitter
- **Splitter** - Receives simple events from one or more sources, creates and closes windows and relays them to connected Operator Instances. Can connect to multiple Operators Instances.
- **Operator Instance** - Creates a window operator for each received window which operates on all incoming events from the Splitter. Complex events detected by the window operators are sent to the Merger.

- **Merger** - Acts as the sink for incoming complex events. Serializes the incoming events into the correct order. Can connect to multiple Operator Instances and is mainly used to measure metrics.

Figure 1.5 shows the relationship between these components. Each component can run separately on a different machine and no shared memory is required. There are a few differences to classic complex event processing. For one, opening and closing windows happens at the splitter component and not within the operator instance. This is necessary so that work can be split to multiple independent operator instances. The splitter opens and closes windows and distributes the windows among the operator instances for scalability. However, the splitter always sends all events to all operator instances. In other words, an operator instance may not receive all windows but receives all events. This way the total work can be split across several operator instances who all work with their own subset of windows.

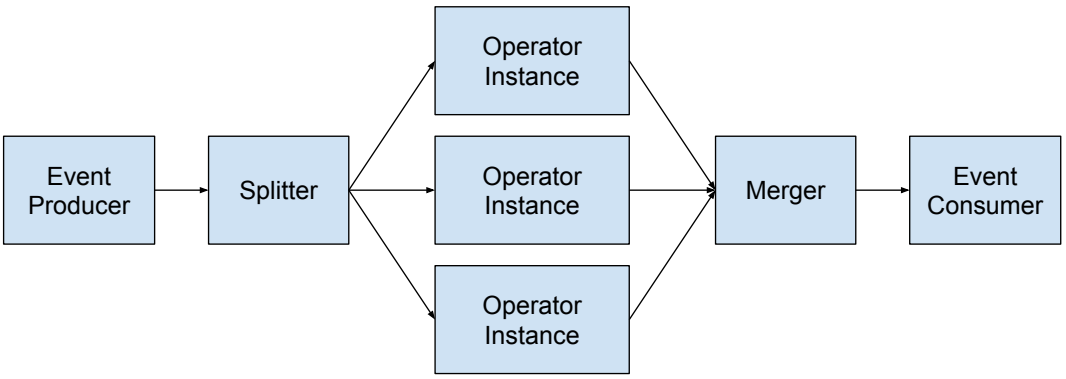


Figure 1.5: Distributed Complex Event Processing framework architecture.

An important challenge for the splitter is load balancing. It must ensure that no operator instance is overloaded with too much work and no other operator instance is idle in order to maximize utility. There are different techniques to ensure this. Among the main techniques are *scheduling* and *backpressure*.

Scheduling is the method of choosing an appropriate operator instance for a window. The scheduler can be static or dynamic. A static scheduler will always follow the same policy regardless of the current load on the operator instances. An example of such a scheduler is *round robin* scheduling. This scheduling algorithm assigns the windows each operator instance by order. For example, if we have two operator instances o_1 and o_2, then every odd window is assigned to o_1 and every even window is assigned to o_2. This is one of the simpler scheduling algorithms. The *batch round robin* strategy

assigns every operator instance a fixed amount of windows before switching to the next operator instance. More complex scheduling mechanism are dynamic, meaning they will periodically get information about the state of the operator instances and when there is high congestion, prioritize other operator instances.

A dynamic scheduler goes hand in hand with the technique of *backpressure*. An operator instance constantly monitors its current congestion level and if it reaches a certain upper bound threshold, the operator instance sends a message to the splitter. When the splitter receives the message, it will stop assigning windows to this operator instance until another message is received. When the congestion level at the operator instance undershoots a certain lower bound threshold, it will send the corresponding message to the splitter, signalizing that it can resume work.

For each connection between components of the DCEP framework there can be an *event out queue* and an *event in queue*. The event out queue manages events which are waiting to be serialized and consequentially sent to the next component. The event in queue manages events which are waiting to be processed. As such the splitter component has an in queue for each connected source and an out queue for each connected operator instance. This approach allows for highly efficient DCEP and follows good design practice through the use of a producer/consumer pattern [MKR15].

An operator instance receives window opening and closing events and normal events from the splitter and puts them into its in queue. A window event, has a unique id for the window, the id of the event that started the window and a flag whether the event opens a window or closes the window. For each window opening event, the operator instance creates a new window operator to process events from the context of that window. The operator instance picks one event from the event queue and all active window operators process it sequentially. After the event has been processed by all window operators, the next event is picket from the queue. When a closing event is received, the operator instance removes the window operator which is associated with the window id as given in the window closing event. As the window operators process incoming events, they detect complex events which are then sent to the merger. The merger receives complex events from multiple operator instances and serializes them in correct order. The sort order is by the time stamp of the complex events.

These are the mechanics of the DCEP framework. We expand on it to add it window drop load shedding capabilities.

1.3 Related Work

There is a great deal of research done on the topic of CEP. A solid groundwork on this topic has been made by [Luc02]. It covers many topics from the perspective of enterprise systems. A more recent overview on this topic was done by [Hed17] which focuses more into the techniques of complex event processing. There are also several broad surveys done on many related topics of CEP on which they cover recent development [FTR+10; Owe07].

In the context of DCEP there is important groundwork done in [MKR15]. It shows the feasibility of a high degree parallelization in the case of an Internet of Things environment. By assuming high volatility and dynamic resource allocation, the paper deeply analyzes the horiontal scalability of such an approach. It provides with efficient algorithms and functions as a groundwork for DCEP. In [MTR16] there is more improvement to reduce communication overhead between operator instances by the use of batch scheduling.

On a more fine-grained note of parallelization stands [BDWT13]. The paper introduces an approach for high parallelization on a shared-memory architecture for the use in multi-core processor systems.

Another paper which improves on that regard is [MST+17], where the author explains the challenges of using consumption policies across multiple operators instances. The paper explains the concept, in which several window versions of a window operator are spawned. A subset of the best window versions are speculatively processed in parallel until an event is consumed, which will eliminate this specific window version. Based on the probability of whether an event will be consumed or not, the best window versions are chosen. The approach shows promising results with near linear increase in throughput proportionally to the number of CPU cores. Continuing work was done in [May18], where the author summarizes key advancements in parallel window processing and brings these challenges together under one coherent framework.

On the topic of load-shedding there is a lot of research done, but is mainly focused for stream processing instead of complex event processing [KBF+15].

One of the earliest work on window aware load shedding in stream processing was done by [TZ06]. The paper introduces a new window drop operator. The operator encodes a drop/keep decision into the tuple which defines the window. The drop/keep decision is described in the form of a window specification which contains an instruction for further downstream operators on how to handle this tuple. The downstream operators decode this window specification and decide whether it is worth it to keep the window or whether to skip it. The window drop operator in this

case only marks windows instead of dropping them. In certain cases, however, the window drop operator can also drop whole windows. It makes this decision by being inherently aware of the whole CEP network and by deriving certain properties of the specifics of certain operators.

[HBN13] generalizes many of the load shedding techniques which are used for stream processing to the level of complex event processing. The paper argues, that the problems are harder in the context of complex event processing. The work continues to categorize each of these problems under different resource bottleneck situations and analytically gives a bound on the hardness of these problems.

However as of now, there is little research done on load shedding specifically for complex event processing yet. To the best of our knowledge, there is no approach available which gathers information about the pattern matching mechanism in a window operator in order to drop windows during high load in complex event processing.

1.4 Problem Statement and Goals

In this thesis we investigate an approach for maximizing the amount of detected complex events in the face of dropping windows under a hard bound on event latency.

There are three key challenges to this problem: (1) Develop a model to estimate the quality and processing latency of windows, (2) design an algorithm that in case of overload drops windows while maximizing the amount of detected complex events, and (3) establish a method to decide how much processing cost needs to be saved in order to not violate the latency bound.

We formulate these challenges as goals and evaluate the proposed approach through experiments.

G1: Develop a model to estimate the quality and processing cost of windows We establish a model which gathers information about window operators and using this information it can predict the quality and processing cost of a window. The quality of a window denotes how many complex events are expected to be detected in this specific window. The processing cost of a window denotes how much CPU resources need to be expended until the window is completed. The model is then used by the algorithm for window load shedding.

G2: Establish a method to decide how much processing cost needs to be saved in order to not violate the latency bound We have set a hard bound on the event latency and thus we need to find a method how to map the processing cost of windows to the event latency. We propose an algorithm which outputs the amount of processing cost which needs to be saved during the current load situation.

G3: Design an algorithm that in case of overload drops windows while maximizing the amount of detected complex events The algorithm uses the window estimation model along with the information how much processing cost needs to be saved and decides with this information which and how many windows to drop.

To reduce the complexity of the problem, we mask a few related concepts from the scope of this thesis:

Load Balancing is an important concept in parallel stream processing. It ensures that no operator is idle while another is busy. A good load balancer keeps all operator instances busy at all times. In the scope of this thesis, we assume that a sufficient load balancing is available.

Distributed Event Consumption Policies When we have multiple window operators which operate on parallel windows, they can independently detect the same complex event which might be undesirable. This however, is a complex topic as there would be a need for communication between different operator instances and window operators. As such, we mask out the issue of event consumption across several window operators. Event consumption policies are still allowed within a single window operator, but not between several.

Whitebox approach In terms of window operators, we follow a whitebox approach. This means, that we can see at any point of time, what is the state of the window operator. We do not know the complete structure and form of the pattern which the window operator tries to match, but we can at least find out how much it has progressed, how many complex events have been detected thus far and how much processing time has been induced by the pattern recognition. The difference to the blackbox approach would be, that we cannot find out anything about the pattern recognition of a window operator, except for the processing time that has elapsed.

Chapter 2

Estimation Model

For our load shedding mechanism we need information about the quality of windows. In this section, we introduce an approach to predict how many complex events a window will detect and how high the processing cost it will induce. We start by defining key terminology, then propose a baseline approach and then explain the sophisticated approach.

2.1 Processing Metrics

Before we establish the estimation model we need to introduce the terminology in which we carefully distinguish between several concepts and show their relationship. These terms are illustrated in Figure 2.1. The picture shows the in-queue of an operator instance containing the events (e_1, e_2, e_3, e_4). It also depicts three windows w_1, w_2, w_3 with a slide size of 1. The picture shows how each event is processed in the context of each window. E.g. e_3 is processed in the context of w_1, w_2 and w_3 separately.

The *event wait duration* is the time difference between the event's arrival time stamp and the time stamp at which the first window operator has started processing the event. It describes how long an event has to wait before it is started being processed in at least one window operator.

The *event processing duration* is the time difference between the beginning and the end of the processing of the event within all window operators. The more windows are open, the longer this duration will be.

The *event latency* is the time difference between the event's arrival time stamp and the time stamp at which the event has been processed in all window operators. The event latency is the sum of the event wait duration and the event processing duration.

The *event throughput* is the metric how many events are processed in the operator instance per second.

We establish similar terms on the window level. The *window duration* is the time difference between when the window was opened and when it was closed. The window duration increases proportionally with the amount of active windows.

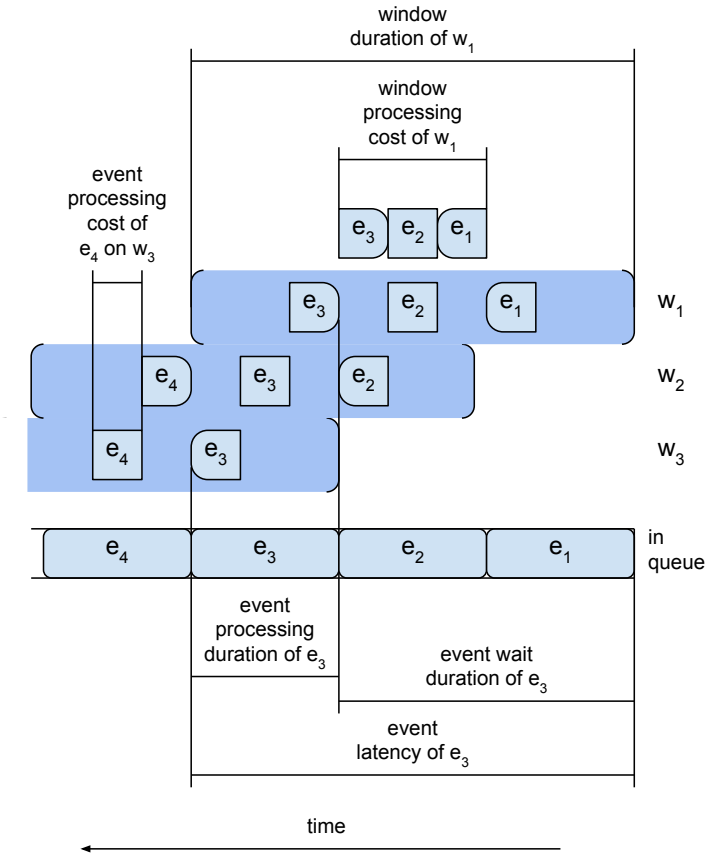


Figure 2.1: Terminology for different metrics regarding the estimation model.

The *window event processing cost* is the time difference between the beginning and the end of the processing of the event within a single window operator.

The *window processing cost* is the load which is induced on the system by processing the window operator. Further on, we will use the shorthand form *cost* of a window. It is measured in time the system would take when processing the window operator

alone, without interference of other active window operators. It is the sum of the window event processing costs of all events in the window of the window operator.

2.2 Assumptions

We propose a model to estimate both quality and latency of windows. Since both metrics depend on many different factors which cannot be grasped in its entirety, it is important to focus on key points and limit the problem space. For this purpose, we lay down several assumptions on which we base our estimation model.

Assumption 1 window processing cost only depends on the number of events in the window, the position of the events relative to each other and the event types.

More particularly, we assume that window cost does not depend on the event payload. However, in the real world, this can very well be the case. To illustrate this, we take the example of the smoke detector. A new requirement states the need that an app can connect to the smoke detector. Within the app it is possible to view at which times the carbon dioxide concentration was especially high. For this, the developer modifies the CEP engine to log all events at which the value was above an arbitrary threshold. Due to this I/O operation, the system experiences high load during events which have a specific event payload.

For our estimation model, we do not consider such cases. Notice, that using a different implementation of the requirement, we can avoid this issue. For example, a preprocessor can analyze incoming events and depending on the event payload, generate new events with a new event type. In the case of the smoke detector, generate a new event with the event type *high concentration*. This approach would be more suitable, since we assume that the window cost does depend on the event type.

It is clear why we assume that the window processing costs depends on the number of events in the window. More events mean more processing cost.

Our assumption is also based on the observation that pattern recognition runs differently depending on the state of the window and the incoming event type. The reasoning behind the assumption that the window cost depends on the event type is that the pattern recognition is looking for specific event types to progress before going into more costly operations. The position of the events relative to each other decides whether the pattern recognition will progress or not. E.g. in the case of matching (A, B) there is a difference if the incoming events are (A, B) or (B, A).

These key factors which affect the processing cost of a window, have also been identified in related work by [May18].

Assumption 2 multiple window operators within one operator instance are processed sequentially, not in parallel.

The consequence of this assumption is, that we can accurately measure how much demand in processing time a single event has taken in a window for a specific window state.

Assumption 3 the pattern recognition of a window operator can approximately be modelled as a deterministic finite state automaton.

Finite state automaton are well known to detect a wide range of possible patterns. All regular expressions can be reduced to a finite state automaton. With this assumption, we actively limit the possibility space of patterns that can be detected. As a consequence and based on this assumption, we will be able to establish the latency model later on.

Assumption 4 the state of an operator instance is visible at all times. As already mentioned in section 1.4, we follow a white box approach. We are able to measure how the pattern recognition of a window operator progresses and see in which state it is at all times. We use this feedback values to gather information necessary for out estimation model.

Assumption 5 the processing duration of an event over all window operators can be estimated accurately enough with the event throughput. The event throughput is the number of events processed per second. Whenever an event is processed by all window operators, it counts as a processed event. Using this metric we can estimate how long it takes until one event is processed on average. Although the processing cost of an event in the context of a single window operator can vary greatly, we assume that on average and over a sufficient amount of active windows, it can accurately represent the current load on the system.

2.3 Approach

We propose two approaches for estimating the quality of a window within a window operator. Approach one is a blackbox approach which simply gathers statistics about the output of an window operator and calculates the average for the estimation. It is meant as a baseline for our more sophisticated approach later on.

The second approach uses a markov reward process for modelling the internal state of a window operator and to predict how many complex events and how much processing cost the window operator will induce.

2.3.1 Weighted Average Strategy

We measure for each active window operator the processing time it takes for completing one event, denoted as the window event processing cost. We do this by measuring the enter and exit timestamp and then calculate the difference. Because each window operator can fully process one event without being interrupted by other threads, this approach can give us a rough estimate on the cost of a single event in a window. This follows from Assumption 2. This is not completely true, as there are still some threads operating in the background, managing serialization and deserialization of events, which may interrupt the event processing, but they are not load heavy and can be neglected in the long run.

Every time a window operator finishes processing an event, we note how long it took and whether a complex event has been detected. For each window operator we additionally maintain how many events it has processed in total, until it is closed. When it is closed, we gather this information and can estimate the average length of a window, i.e. the amount of events in a window. Over many executions of window operators we can estimate:

- The fractional amount of detected complex events per event on average, denoted as $\overline{q(e)}$
- The window event processing cost per event on average $\overline{c(e)}$
- The amount of events a window operator processes on average, denoted as the average length of the window $\overline{L(w)}$

Using these metrics we can estimate the average cost and quality of a window. The average cost of a window is $\overline{c(w)} = \overline{c(e)} \cdot \overline{L(w)}$, the average quality of a window is $\overline{q(w)} = \overline{q(e)} \cdot \overline{L(w)}$.

Now, having these metrics, we can calculate the residual cost and quality of an active window. As a window operator processes events, its remaining length reduces with each processed event. E.g. If the average length of a window is 5 and it has already processed 3 events, then the window expects to only process 2 more. As already mentioned, we keep track of how many events a window operator has processed thus far. We denote the amount of processed events in a window as $n(w)$. It follows, the residual cost of a window is $r(c(w)) \approx n(w) \cdot \overline{c(e)}$. Analogous to it, the residual quality $r(q(w)) \approx n(w) \cdot \overline{q(e)}$.

This can only approximately estimate the residual cost. The approach does not take into account changing distributions of events. It will always output an average value over the entire history of the operator instance. The other problem is that it completely homogenizes cost and quality over all windows. As stated in section 1.4, we want to drop low quality windows and keep high quality windows, which is not possible here. As such this approach is not sufficient for our case and can only serve as a baseline. Henceforth, we will refer to this estimation strategy as *weighted average*, as it weighs the average cost and quality with the number of events left.

2.3.2 Markov Reward-Cost Process

For a more sophisticated approach, we will exploit Assumption 4: The state of a window operator is visible at all times. As such, whenever a window operator processes an event, we note the state of the window operator before and after. Through Assumption 3 the state can be reduced to a single value. As the mechanism of pattern recognition in a window operator can be reduced to a deterministic finite state automaton, the window operator needs to only keep track of a single value.

In such a case, it advantageous to use a markov chain for the estimation model of the window operator. We cannot directly use a corresponding finite state automaton for the window operator as it is generally unknown what kind of event types are arriving. As we cannot predict what kind of event types at which rates and at which position will arrive, we will encode the probability of transition into the transition matrix of the markov chain itself.

In a finite state automaton we progress from one state to another by inputting a symbol which is associated with the transition from the former state to the latter state. In a markov chain we progress from one state to another inherently probabilistic. As such, we can gather statistics on how often a window operator transitions from one state to another and using this statistic, we encode this probability into the transition matrix.

As an example, lets take a window operator which should match the pattern (A, B, C). If we assume that there are no other event types and that they arrive inherently randomly, then the markov chain represented by this window operator is one which has a probability of 0.33 at every state to transition to the next state and a probability of 0.66 to stay at the current state after processing an event. Therefore, we can argue using the markov chain, that after the window operator has processed an event, there is a 0.33 chance that it has progressed in its pattern matching.

Using a markov chain we are able to estimate the state a window operator will be in the future. However, using just a markov chain is not sufficient to predict how many many complex events will have been detected by the window operator at a specific point in time. Therefore, we extend our markov chain by associating each transition with a reward value. A markov chain with reward is called a *markov reward model*. The process of solving a markov reward model is called *markov reward process*. It is a special case of a markov decision process in which there is only one action.

Using an algorithm from the field of reinforcement learning known as *value iteration*, we can solve the markov reward process. The result is the expected reward which has been accumulated after a specific number of steps. We can match the number of steps to the number of events that have been processed. The algorithm works by the use of dynamic programming, by calculating the expected reward at every state in the markov chain iteratively and then reusing these values in future iterations. At the center of this algorithm is the *bellman equation*:

$$V_{i+1}(s) := \sum_{s'} P(s, s')(R(s, s') + V_i(s'))$$

V is the state value of a state, it denotes the iteratively calculated expected reward. s' is a state which can be reached starting from state s. $P(s, s')$ is the transition probability from s to s' and $R(s, s')$ the transition reward. In each iteration, this equation is calculated for each state in the markov chain. We explain this by example.

Figure 2.2 shows a markov reward model with four states. Each transition shows the probability, denoted by p and reward denoted by r. After the first iteration, state 3 has $V(3)_1 = 0$ as it does not have any transitions. State 1 transitions to state 3 with probability $p = 1$ and $r = 1$ and therefore its expected reward after one step is $V(1)_1 = 1 \cdot 1 = 1$. Similarly with state 2, which also has only one transition but with reward 2 and as such $V(2)_1 = 2$. For state 0 the bellman equation calculates $V(0)_1 = 0.25 \cdot (3+0) + 0.75 \cdot (2+0) = 2.25$. In the next iteration, the other states will not have any new values, as their transitions go into state 3 whoose state value $V(3)_1$ is still 0. But since the state values of state 1 and 2 have changed since the previous iteration, the state value of state 0 will now calculate $V(0)_2 = 0.25 \cdot (3 + 1) + 0.75 \cdot (2 + 2) = 4$.

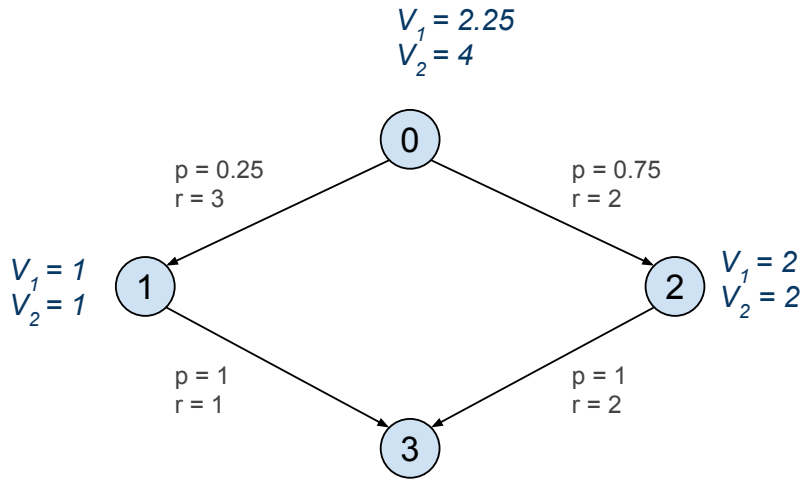


Figure 2.2: Solving a simple Markov Reward Chain using dynamic programming.

By looking at the example, it is clear that the expected reward of 4 for the initial state after two steps is correct. If we go the left path, we will gain a reward of 3 and then a reward of 1. If we go the right path, we will gain a reward of 2 and then another reward of 2. So we always end up with a total reward of 4.

If we model our window operator as a markov reward model, we can use value iteration to estimate the amount of detected complex events after a certain amount of events have been processed starting from a specific state. The only question now remains, how to model the cost of a window operator. To this end, we extend our markov reward model. For each transition we now not only associate reward, but also cost. We call this extended markov reward model a *markov reward-cost model*. When solving the markov reward-cost process, we do value iteration twice. Once for the reward and once for the cost.

To learn a markov reward-cost model we need many observations on how a window operator behaves when it has processed an event. Every time a window operator processes an event we measure beforehand which state it was in and in which state it is afterwards. We also measure how much time it took and how many complex events have been detected. We encode this information as a 4-tuple: (s, s', r, c) and call it a single *observation* henceforth.

Each observation is encoded into a *markov statistics model*. In this model we keep the sum of all values in several matrices. The model contains a matrix for probability P_{stat}, for reward R_{stat} and for cost C_{stat}. Additionally, it also contains a list of all states

which were the source of an observation Q_{stat}. For each observation (s, s', r, c), we store:

- $P_{stat}(s, s') = P_{stat}(s, s') + 1$
- $R_{stat}(s, s') = R_{stat}(s, s') + r$
- $C_{stat}(s, s') = C_{stat}(s, s') + c$
- $Q_{stat}(s) = Q_{stat}(s) + 1$

At any point of time we can then calculate the markov reward-cost model from these statistics by the following rules:

- $P(s, s') = \frac{P_{stat}(s,s')}{Q_{stat}(s)}$
- $R(s, s') = \frac{R_{stat}(s,s')}{P_{stat}(s,s')}$
- $C(s, s') = \frac{C_{stat}(s,s')}{P_{stat}(s,s')}$

Another challenge is how to efficiently learn the markov reward-cost model in order to minimize overhead on the operator instance. For this purpose, we establish a multi-threaded architecture for the markov estimation model. We separate the concept into three components: (1) Markov Model Learner, (2) Markov Model, and (3) Markov Solver. Figure 2.3 shows these components.

The markov model learner is entirely responsible for accepting observations, managing the markov statistics model, and writing the markov model. It operates in its own worker thread and also manages a queue which contains observations as provided by window operators. Periodically, after a set amount of observations have been written to the markov statistics model, it recalculates a new markov reward-cost model using the most recent statistics. As the markov model learner operates in its own thread, there is the need for a swap buffer markov model. Otherwise, external access to it could result in parallel read/write conflicts. As such, the markov model learner always writes on one buffer, while exposing the other buffer (older markov model) for read access.

The markov solver reads the provided markov reward-cost model to solve it using value iteration. It saves the result of the value iteration algorithm at every step. This means, we can potentially find out the expected cost or reward of a window operator starting from any state and with a specific amount of events left. In other words, we save the expected reward or cost for every state and for any amount of steps. Of course, having any amount of steps is not feasible, but we can rather specify an upper

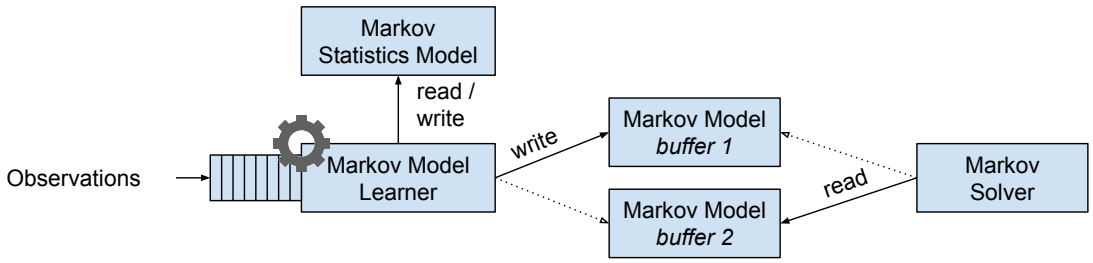


Figure 2.3: Architecture for the estimation model using a markov model, markov model learner and markov solver.

bound of the number of steps that we will want to inquire about later on. It is logical, to use the upper bound of steps to be the maximum length of a window.

However, as the length of windows can be quite large, it might be infeasible to keep all these values stored in memory. To that end, we implement a bucket mechanism to reduce the number of values that we need to store but still support a large number of steps. E.g. we store for a state 0 the expected reward after 1 step, after 2 steps, after 3 steps and after 4 steps. We can summarize the expected reward for step 1 and 2 into one result bucket and the expected reward for step 3 and 4 into another result bucket. To get the expected reward for step 2, we can then interpolate between result bucket 1 and result bucket 2 to get an approximate value of what step 2 could have been.

This concludes our estimation model. Using the state of a window operator and the expected number of events the window operator has left to process we can now, at any point of time and with low impact, inquire about the expected reward the window will yield and what cost it will imply.

Chapter 3

Window Drop Load Shedding

Knowing the cost and reward of a window, we can now proceed into the topic of window drop load shedding. The challenges which are addressed in this section are:

- When do we need to drop windows?
- How many and which windows do we need to drop?

We will begin by introducing the overall architecture of the approach and then explain the individual components.

3.1 Architecture

In this section we go over the architecture of our approach and provide a description for key components. Figure 3.1 illustrates the architecture to which we will now explain the depicted components quickly and later on in detail.

Window Operators The window operators process the incoming events as quickly as possible, however if the load is higher than the throughput, the window operators cannot keep up. This will result in an increasing amount of events waiting in the input queue and thus the waiting time of an individual event also increases. On the other hand, if the throughput is higher than the load, the number of events in the input queue will reduce, as more events are being processed than new events coming in. This results in a shorter waiting time for the events until they are processed. As discussed in subsection 2.3.2, for every event a window operator processes, it provides an observation to the Window Estimator containing the before-state, the after-state,

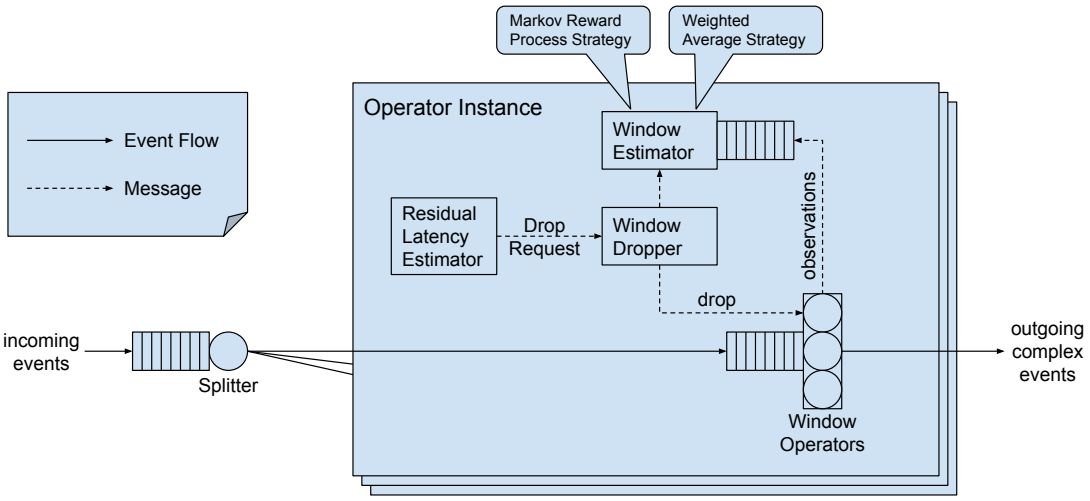


Figure 3.1: Queuing Network of the DCEP framework extended with key components and messages between components.

the number of detected complex events and the duration the window operator has spent processing the event.

Window Estimator The window estimator estimates the cost and quality of a window. It can have one of two modes: using either the markov reward-cost process strategy or the weighted average strategy. When it is set to the weighted average strategy, it receives observations and using these observations gathers statistics as described in subsection 2.3.1. When it is set to the markov reward-cost process strategy, it directly sends any incoming observations to the markov model learner as described in subsection 2.3.2.

Residual Latency Estimator This component periodically estimates the load on the window operators and weighs it against the throughput under consideration of the latency bound. The output is the amount of processing time that is left until the latency bound is violated, i.e. the residual latency. The latency bound is violated, when the time between the arrival of an event and until the event has been processed is higher than the latency bound.

Window Dropper The *window dropper* periodically gets information from the residual latency estimator on how much processing cost needs to be saved. It then starts the algorithm to decide which windows to drop.

This is the quick overview of the architecture of our approach. We now go into more detail on how each component works and what the challenges are.

3.2 Residual Latency Estimator

In this section we depict the approach for estimating the residual latency. The residual latency is the amount of processing time that is left until the latency bound is violated for the last event.

If the arrival rate of new events is higher than the throughput of the operator instance, more and more events will have to wait in the queue. This results in a higher wait time, the further an event is in the queue. The highest wait time in the queue has the event that most recently arrived, as it gets put in the very back in the queue and will be processed after all other events have been processed. This behavior is also illustrated by Figure 3.2

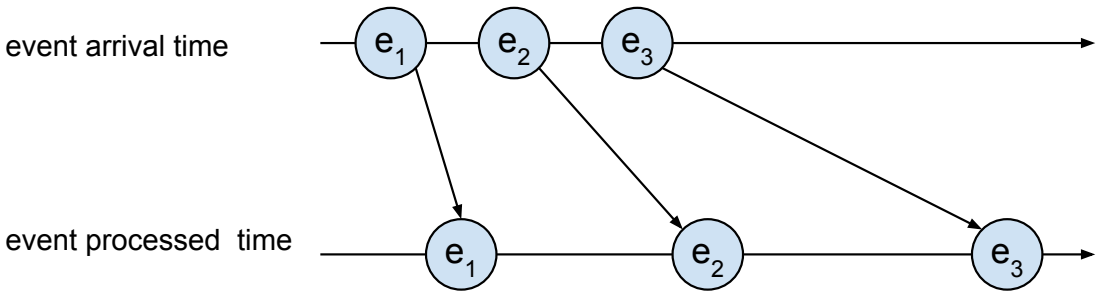


Figure 3.2: When the event queue is filled up, each subsequent event will have a longer latency.

One approach to calculate the residual latency is by measuring the event latency directly. Whenever an event gets put into the queue, its arrival time is noted and upon being processed we note its processed time. In Figure 2.1 one can observe, that the sum of the processing cost of all events in the queue equals to the event latency of the

last event in the queue. Therefore, we only have to ensure the event latency of the last event in the queue at all times to guarantee the latency bound.

If we denote the event latency of the last event as $l(e_n)$, then the residual latency would be $r = l(e_n) - \hat{l}$ with $\hat{l}$ being the latency bound. However, we cannot find out the latency of the last event in the queue, we can only measure the event latency of the current event. As such, the residual latency estimator would provide a value which is lagging behind. The lag would be higher the more events there are waiting in the queue. And as such, we would only notice that the latency bound has been violated when it has already happened. This approach would be of little use to the window dropper.

A better approach is by calculating the throughput of the operator instance. The throughput tells us, how many events we process each second. By inverse logic, it also tells us how long it takes to process a single event. We can scale this value up to the number of events in the queue to get an estimate on the current event latency. We covered with Assumption 5 that the event processing cost can be used to estimate this value.

We calculate the throughput by counting all the events that have been processed in an arbitrary time interval. By dividing the counted number by the interval, we get the throughput of the operator instance. With this approach there is the question on how big of an time interval needs to be set. If it is too high, we get the same problem and will always get a value that is lagging behind. If it is too low, the variety in length of different event processing durations will be noticable and the value cannot be scaled up accurately enough.

A more responsive approach is to store all arrival times of the events in a fixed size queue. Each new arrival time removes the oldest arrival time from the fixed size queue. By keeping track of the oldest arrival timestamp and the newest arrival timestamp, we have a dynamic sliding interval with which we can calculate the throughput.

Calculating the throughput for the estimate of the residual latency has one issue though. As we calculate the residual latency and give it as input to the window dropper, the window dropper will use this information to decide which windows to drop. However, this will create a feedback-loop as dropping a window also increases the throughput.

Therefore, we propose to baseline the residual latency estimation on the lowest possible throughput. The throughput is at its minimum when there are a large number of windows active. As such, we will only record new event arrivals, when the amount of windows is at its maximum. As already mentioned in subsection 1.2.1 we can

calculate the maximum number of windows by dividing the window length by the window slide. These are values that can either be estimated or are generally known.

However, as the number of active windows drops, we will have an interval in which there were no events recorded. This will mean, the throughput calculation will miscalculate, as it will have missing records. To this end, we extend the throughput calculation by the use of multiple intervals. This is shown in Figure 3.3. Whenever we stop recording event arrivals, the throughput calculator shall close its current calculation interval. When it then receives a new record, it creates a new record interval. The throughput is then calculated by the total amount of events in all intervals divided by the sum of all intervals.

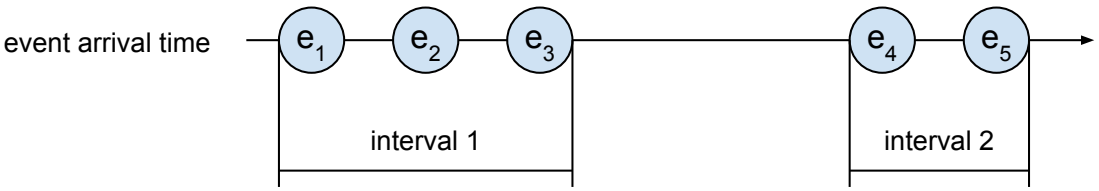


Figure 3.3: A break in between two intervals needs to be excluded from the throughput calculation.

Now using this throughput, we can calculate the residual latency by $r = \frac{N}{\lambda_{\hat{W}}} - \hat{l}$, where N is the number of events in the queue, $\lambda_{\hat{W}}$ the throughput when having the maximum amount of windows and $\hat{l}$ the latency bound.

A positive r is the amount of processing time left until the latency bound will be violated. If r is negative, the latency bound will be violated for the last (newest) event in the queue, assuming the throughput remains unchanged. To counteract this case, we increase the throughput by reducing load by dropping windows. As such, the *Residual Latency Estimator* will send a drop request to the *Window Dropper*.

One pecularity about dropping windows is that the positive effect on the throughput will not be immediate. When we estimate the processing cost of a window, what we get is the processing time that we can save over the entire lifespan of the window, if we decide to drop it. As such, simply deciding how much processing cost to drop is not enough, the residual latency estimator needs also to estimate until which point of time that much processing cost needs to be freed.

We calculate this point in time until which the processing cost must have been freed with $t_{drop} = t_{now} + \hat{l} - r$. The difference between the residual latency and the latency bound is the interval in leeway that we have, until we finally have to drop the window.

3.3 Window Dropper

The window dropper receives drop requests by the residual latency estimator and processes them in order to drop windows. Drop requests must include the processing time which needs to be saved r and the timestamp until when the drop request must have been fulfilled t_{drop}.

At any point of time, the *Window Dropper* maintains a list of pending drop requests and the sum of $R := \sum r$ of all drop requests in the list. Whenever a drop request comes in, the current sum of the residual latencies is recorded and associated with the drop request. As such, a drop request is 3-tuple (r, t_{drop}, R).

As drop requests are not carried out immediately, new drop request may be incoming before old ones have been carried out. Thus, it is important to find out how much processing time will have been saved until that point. To illustrate, imagine the following scenario: The Window Dropper receives a drop request at $t = 0$ with $(1, 5, 0)$. This would mean enough windows need to be dropped so that $r = 1$ unit of processing time can be saved until timestep $t_{drop} = 5$. Before the drop request has been carried out, a new one arrives at $t = 1$ with $(1, 6, 1)$. At this point, the Window dropper can safely ignore the second drop request and discard it. The reason is, that the residual latency estimator sent a drop request at which point of time the older drop request has not been carried out yet. So by calculating $R - r = 0$, the window dropper knows that the new drop request is for saving the same cost that the old drop request is meant for.

Therefore, any drop requests that the window dropper receives, he will try to fuse with an already existing drop request. We try to keep the amount of drop requests as small as possible in order to avoid computational overhead later on.

To that end, we also provide the window dropper with a fuse threshold. The fuse threshold is a constant that describes up to what difference of two drop timestamps $t_{1_{drop}}$ and $t_{2_{drop}}$ the window dropper will fuse these two drop requests together. It is a tradeoff between being accurate and leightweight. I.e. if there is already an existing drop request with $t_{1_{drop}} = 100$ and another drop request comes in with $t_{2_{drop}} = 110$, then if the difference $t_{2_{drop}} - t_{1_{drop}} = 10$ is smaller than the fuse threshold, both drop request will be fused together as one. The consequence is, that the drop amount of the already existing drop request will be summed with the new one. The incoming drop request will be discarded.

Now we have the following information for each drop request:

- amount of processing cost that needs to be saved, denoted as c_{save}

- timestamp until which it needs to save the amount of processing cost, denoted as t_{drop}

Additionally we can obtain by the use of the window estimator:

- the expected number of events a window has left to process $E(L(w))$
- the expected cost of a window $E(c(w))$
- the expected number of complex events a window will detect $E(q(w))$
- the current throughput λ
- the current list of active windows $w_1, ..., w_n$

For our case, we need to find an algorithm which has a low computational overhead. As we are trying to save processing time, we would want to waste as little of it as possible on managing load shedding.

At first glance, the problem presents itself as a 0/1 knapsack problem. Each of the windows is an item associated with a cost and a reward. And we have a limited amount of processing power. Formulated as the knapsack problem, we need to find a combination of windows which we want to process that fit into the limited amount of processing power while still maximizing the amount of detected complex events. However, it is well known that solving a knapsack is a NP-hard problem and as such not feasible for our approach.

A possbility to circumvent this, is to use the greedy approximation scheme for the knapsack problem. It is known that the greedy knapsack has a competitive ratio of 2. Meaning, in the worst case, the greedy algorithm will be off the optimum solution by a factor of 2. This is an acceptable tradeoff for our case. However, even the greedy knapsack approximation algorithm still has complexity of $O(n \log n)$.

We instead, propose an algorithm which runs in $O(n)$ by the use of probabilistic decision making, with n being the number of active windows. The algorithm is described in Listing 3.1.

```
function dropWindows(windowList, dropRequestList)

  LOOP ∀ r ∈ dropRequestList
    LOOP ∀ w ∈ windowList
      u := calculateUtility (w)
      update u_min and u_max using u
      p := calculateDropProbability(r,w,u_min,u_max)
      IF p < random()
        reevaluateDropRequest(r,w)
        w.drop()
      END
    END
  END

END
```

Listing 3.1: Window Dropper pseudo-code

The algorithm calculates the utility for each window. The utility of a window is calculated by $E(u(w)) = E(q(w))/E(c(w))$. It maintains statistics regarding the lowest utility and the largest utility it ever encountered. As there can be windows which have a high cost and windows which have a low cost, it is important to know the scale of things. It then calculates the drop probability of a window and drops it accordingly. When the window has been dropped, the drop request must be reevaluated. E.g. if the drop request has $r = 2$ but the window cost was only 1, then the drop request is updated to only have $r = 1$. If the window cost was higher than the drop request should have saved, then the drop request is removed and the next drop request in the list is taken into account. This is shown in Listing 3.2.

```
function reevaluateDropRequest(r : dropRequest, w : window)

  s := r.saveCostAmount
  c := windowEstimator.estimateCost(w,e)

  IF s < c
    r.remove()
    continue with next dropRequest
  ELSE
    r.saveCostAmount = r.saveCostAmount − c
  END

END
```

Listing 3.2: Utility calculation

Now, to calculate the drop probability of a window, we take a lot of information into account. Listing 3.3 shows the calculation. The probability of dropping a window

mostly depends on its utility. However, we also take into account whether the window will be able to actually complete in time.

This is calculated by the processing time left. We can estimate the number of events a window has left to process and also get the current throughput. With this information we can get an estimate at which point in time, the window will be closed.

If the processing time of a window is higher than the drop time left of the drop request, then we need to reduce the utility of that window. The reason is that we can only partially save its cost for the drop request. That way, we also want to reduce the probability that this window will be picked for being dropped.

```
function calculateDropProbability(r : dropRequest, w : window, u_min, u_max)

  t := windowEstimator.estimateThroughput()
  e := windowEstimator.estimateEventsLeft(w)
  c := windowEstimator.estimateCost(w,e)
  q := windowEstimator.estimateQuality(w,e)

  processingTimeLeft := t / e

  IF(processingTimeLeft > r.dropTimeLeft())
    u := calculateUtility (w) * r.dropTimeLeft() / processingTimeLeft
  ELSE
    u := calculateUtility (w)
  END

  p := greedyness(u_min, u_max, u) * patience(processingTimeLeft / r.dropTimeLeft())

  RETURN p

END
```

Listing 3.3: Drop probability calculation

The drop probability can be influenced by two additional functions: (1) the greedyness function and (2) the patience function. Both of these functions map a value between 0 and 1 to a value between 0 and 1. The idea behind the greedy function is, that we can provide a function, which more agressively prefers windows with high utility over windows with low utility. However, an aggressive greedyness function has the consequence that windows which may have had a wrong early estimation by the window estimator, may be dropped even though they were high quality windows. On the other hand, if the utility values of different windows are numerically close to each other, this would mean that we do not really exploit having the knowledge which window is preferable. By default we can take here the linear function to be neutral in both regards.

The patience function maps the percentage of processing time left until which the window should be dropped. The idea behind the patience function is to tell the window dropper when to drop windows. There is a tradeoff here at which the algorithm shall either pass dropping a window in order to have more information about the window later on, or if it should drop it as soon as possible, so that it has more leeway in decision making with other windows.

Another question that comes up is, when do we need to drop windows? Let's consider the extreme case of windows consisting of a single event. In this case, we cannot gather information about the quality of a specific window or the latency. As such, when there is a high spike in load, the system might not be able to keep up and windows will need to be dropped. If the latency bound is approached then windows need to be dropped as fast as new windows being created. But in the normal case, we can just trigger the window dropper in fixed time intervals. A reasonable time interval would be the latency bound divided by 10.

Chapter 4

Evaluation

In this chapter we present our evaluation. We depict how the experiments were conducted and how we validated our results.

4.1 Experiment Design

We design two experiments for the evaluation. One experiment aims to estimate the goodness of the estimation model. The other experiment aims to provide the feasibility of the window dropper algorithm.

For the case study, we will use three different operators: (1) regular pattern matcher, (2) limited chronicle pattern matcher and (3) fork pattern matcher. For each operator we simulate a heavy operation by the use of a busy wait whenever it matches an event type.

The regular pattern matcher is a simple operator that matches event stream patterns of the form $(A, B, C, D, ...)$. It follows the consumption policy *regular* as explained in subsection 1.2.1. During the experiment we set the pattern size to 10. The pattern size decides how many event types there are and how many of these the operator has to match before detecting a complex event. I.e. a pattern size of 3 would mean, the operator only needs to match (A, B, C).

The limited chronicle pattern matcher is similar but it follows the consumption policy *chronicle,* as explained in subsection 1.2.1. However, it is only limited chronicle. The reason is, that a full chronicle pattern matcher cannot be represented as a finite state automata. The consequence is that it is not clear what the state of the operator is

supposed to be. I.e. if the pattern matcher progressed with one partial match and then opens a new partial match, this cannot be represented as a single value.

Therefore, we compress the state of the operator into a single value with the help of a bitpacker. This of coure, explodes the value as there are many different states such an operator can have. To that end, we limit the number of partial matches to 3. That way, the number of states the operator can have is limited to $(\text{pattern size})^3$.

The last operator for the experiment is the fork pattern matcher. The fork pattern matcher is meant as a synthetic experiment and shall show how the window dropper exploits its knowledge of bad windows. Figure 4.1 shows the finite state automaton of the pattern for a pattern size of 10. If the operator matches the event type A first, then it goes into a path where all subsequent event matches only take a low load to match. If it matches the event type F first, then it goes into a path where all subsequent event matches present a high load, ten times higher than the normal simulated load.

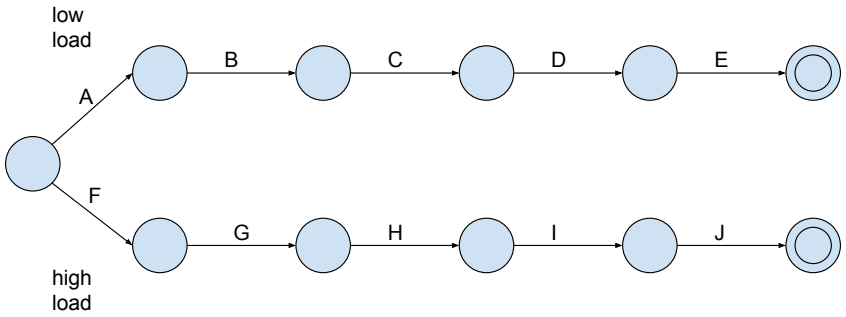


Figure 4.1: The fork synthethic experiment.

That way, the markov reward-cost process will learn, that windows which are in state $B - E$ are more promising than windows in states $G - J$.

The experiment is conducted on a cluster of 4 machines. One machine runs the event source, one for the splitter, one for the instance operator and one for the merger. The parameters which are used for all the experiments are depcited in Figure 4.2. The bootstrap and warmup event count are used for the drop quality experiment. The pattern size is 10 with a window size of 90. Also there is a slide size of 9, meaning there will be 10 windows active at any point in time. The simulated load is 1ms, meaning every match a window operator performs will cost at least 1ms in processing time. The latency bound is set to 10 seconds which is only relevant for the drop quality experiment.

bootstrap event count	10000
warmup event count	20000
pattern size	10
window size	90
slide size	9
simulated load	1ms
latency bound	10s

Figure 4.2: The parameters for conducting the experiments.

4.2 Estimation Quality

We measure the estimation quality by comparing the expected value es determined by the window estimator to the actual value. We measure the estimation quality at different rates of completion of a window to see how accurate the window estimator is able to estimate the remaining cost or quality.

If a window has not processed a single event yet, its completion rate is 0. At this point, neither the markov reward-cost process nor the weighted average have any meaningful information about the state of the window operator. As the window operator is still in its initial state, both strategies can only estimate the average quality and cost of the window.

After the window operator has processed a few events and has or has not progressed in its pattern matching, we have gained more information. The markov cost-reward process may argue, that it did not progress but there are now fewer events left to be processed, as such the value of such a window may decrease. On the other hand, if the window operator did progress in its pattern matching, then it can argue that the value of that window increased as it got closer to a full pattern match.

The weighted average strategy has no state information about the window and therefore can only provide the projected average value depending on the amount of events left.

In this experiment, we measure how accurate the prediction of these strategies was at different window completion rates. The experiment iteratively increases the completion rate at which point to measure. A window operator reaches the current completion rate when it has proportionally processed more events than the completion rate, i.e. if the window length is 10 and the completion rate is currently set to 0.5 then the window operator reaches the completion rate when it processes its fifth event. At

this point, we measure how many complex events the window operator has detected thus far and how much processing cost it induced and we also measure what the window estimator predicts the window operator will match and how much processing cost it will cost more until it is closed. Then, when the window has been closed, we measure what the actual amount of detected complex events was and the actual processing cost.

With this information, we get at different window completion rates the value which the window estimator predicted and the real value. We then calculate the mean squared error between these two values to represent it in a chart. In other words:

$$relativeErrorCE = MSE(actualCE, measuredCE + estimatedCE)$$

$$relativeErrorCost = MSE(actualCost, measuredCost + estimatedCost)$$

What we expect from this experiment is a chart showing a downwards trend till at the end it reaches the value 0. We expect that the markov reward-cost process is more accurate, as it considers state information while the weighted average does not.

The result of the experiment regarding processing cost is illustrated in Figure 4.3. The estimation for the processing cost of a window is closing in to the real value as expected in all charts. The markov reward-cost process is able to predict the cost of the window operators more accurately than the weighted average strategy, as expected. Interesting is, however, that in the experiment with the regular window operator, both methods are effectively the same. The reason lies in the fact, that there is only one path in the regular pattern matcher and the cost and complex events detection is proportionally to the completion rate. In such a simple pattern there is no advantage to use the markov reward-cost process, as predicting the subsequent cost can be done by simply upscaling the previously experienced cost.

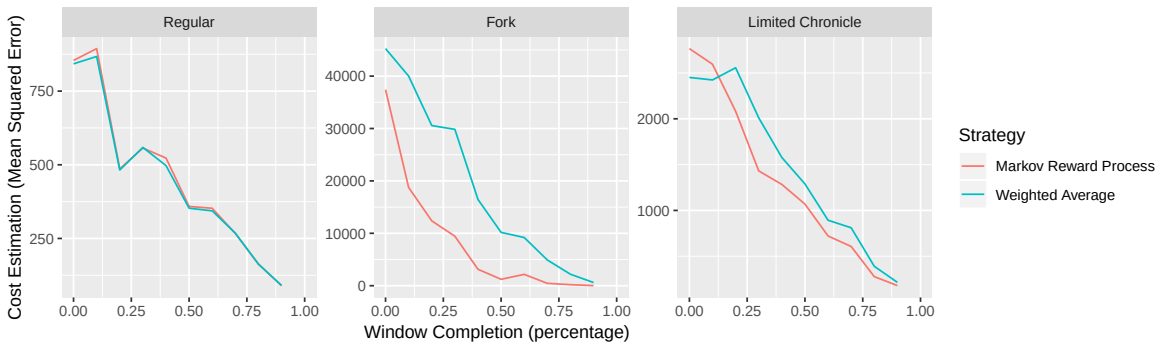


Figure 4.3: The mean squared error of the window cost estimation compared to the actual value.

The result regarding the prediction of the number of detected complex events is shown in Figure 4.4. Again, the chart shows that the markov reward-cost process is more accurate than the weighted average. In the fork operator experiment it can very accurately predict the expected detected complex events because both paths of the fork experiment are equally long. It is also interesting, that it is even perfoming better in the limited chronicle experiment.

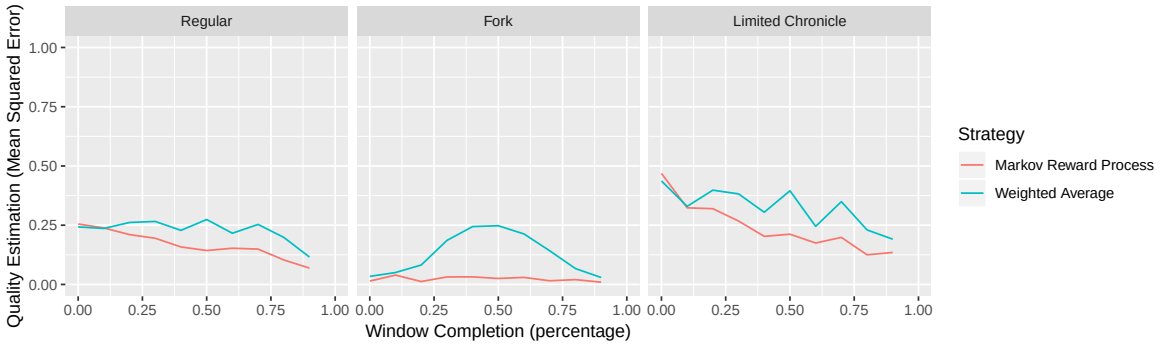


Figure 4.4: The mean squared error of the expected complex events estimation compared to the actual value.

4.3 Drop Quality

To measure the drop quality of the window dropper, we need to know at first, how many events the operator instance is capable of handling without needing to drop. Afterwards we increase the load in order to trigger the window dropper. The window dropper must fulfill several criteria: (1) no event violates the latency bound, (2) during load spike the window dropper will quickly react to drop as many windows as needed and (3) the window dropper has a better quality rate when using the markov reward-cost process as estimation strategy than the weighted average strategy.

To that end, we split the experiment in three phases: (1) bootstrap phase, (2) warmup phase and (3) measure phase.

In the bootstrap phase we generate a specific amount of random events in the source and after that amount of events have been generated, we move to the next phase. During bootstrapping we let the markov model learner learn its model on the basis of the event stream. We also measure at this stage the throughput of the operator instance. We measure the throughput with the help of a seperate *event rate controller*. The event rate controller is set up between the splitter and the operator instance as

depicted in Figure 4.5. As the event queue of the operator instance fills up and reaches a certain upper bound threshold, it sends a block-message to the event rate controller. When the event rate controller gets this message for the first time, it starts measuring the time and number of events that are sent to the operator instance from this point on. When the queue of the operator instance reduces and falls under a certain lower bound threshold, it sends another message to the event rate controller telling it to resume sending events.

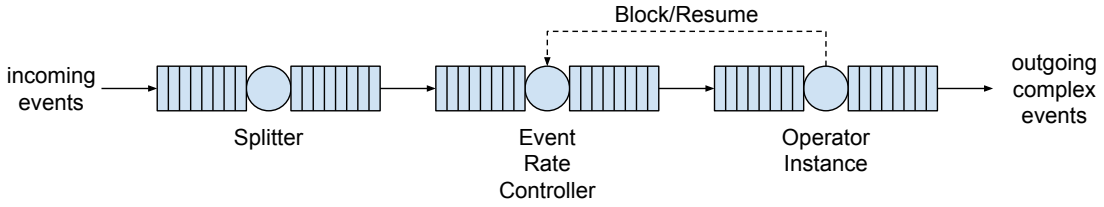
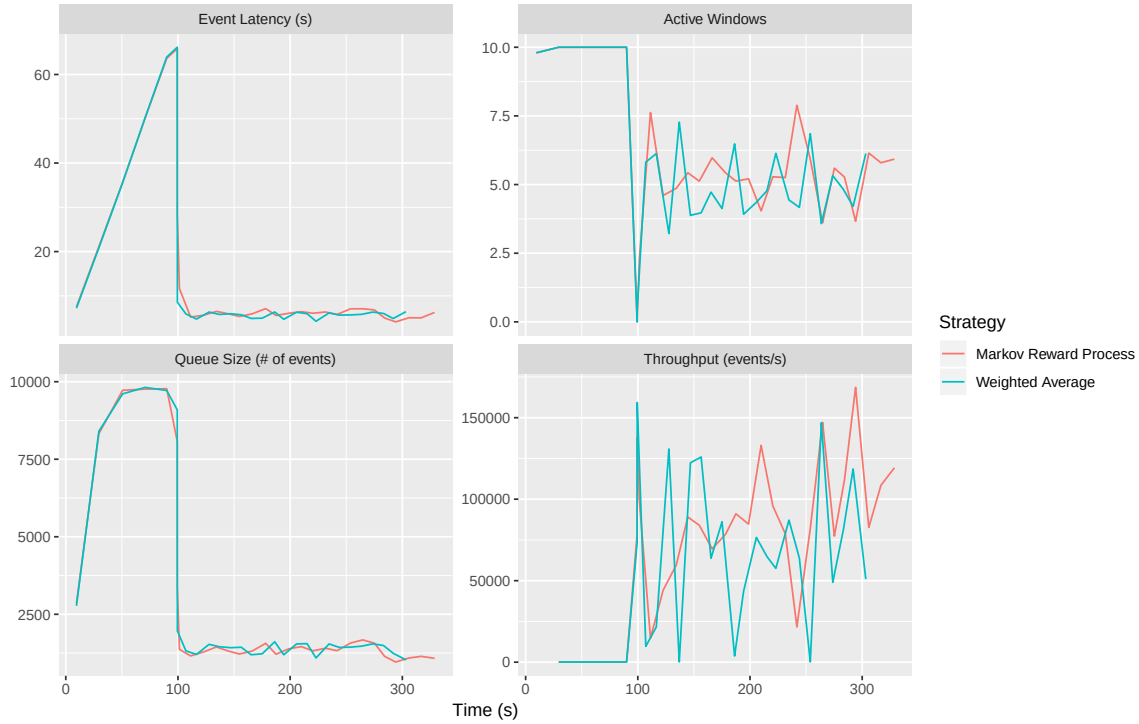


Figure 4.5: The event rate controller between splitter and operator instance.

After a certain amount of events have been sent during the bootstrap phase, we go over to the warm up phase. At this point, the event rate controller checks how many events have been sent to the operator instance since it started measuring and uses this information as the throughput that the operator instance is able to handle without dropping events. We then double the throughput in order to force the operator instance to drop windows and activate the window dropper. At this point many events have been queued up at the operator instance and the window dropper will drop all windows, until the latency bound is not violated anymore and the queue almost empty. After another certain amount of events we then move to the measure phase.

During the measure phase, the event latency must be kept under the latency bound at all times. We continuously measure the event latency, the queue size, the number of active windows and the throughput of the operator instance. At the end of the experiment we get the number of complex events that have been detected during the measure phase, the duration of the measure phase and the number of windows that have been dropped.

The results of the experiment run with the regular operator are shown in Figure 4.6. In the chart for the queue size, we can observe how the queue fills up during the bootstrap phase and stays at the peak for a while. This is because the operator instance sends block/resume messages to the event rate controller so the queue size remains fixed during the bootstrap phase. As soon as the warmup phase begins, the window dropper is set active and it instantly drops all windows, as the event latency is already way over 10s. The throughput is only being measured starting from the warmup phase



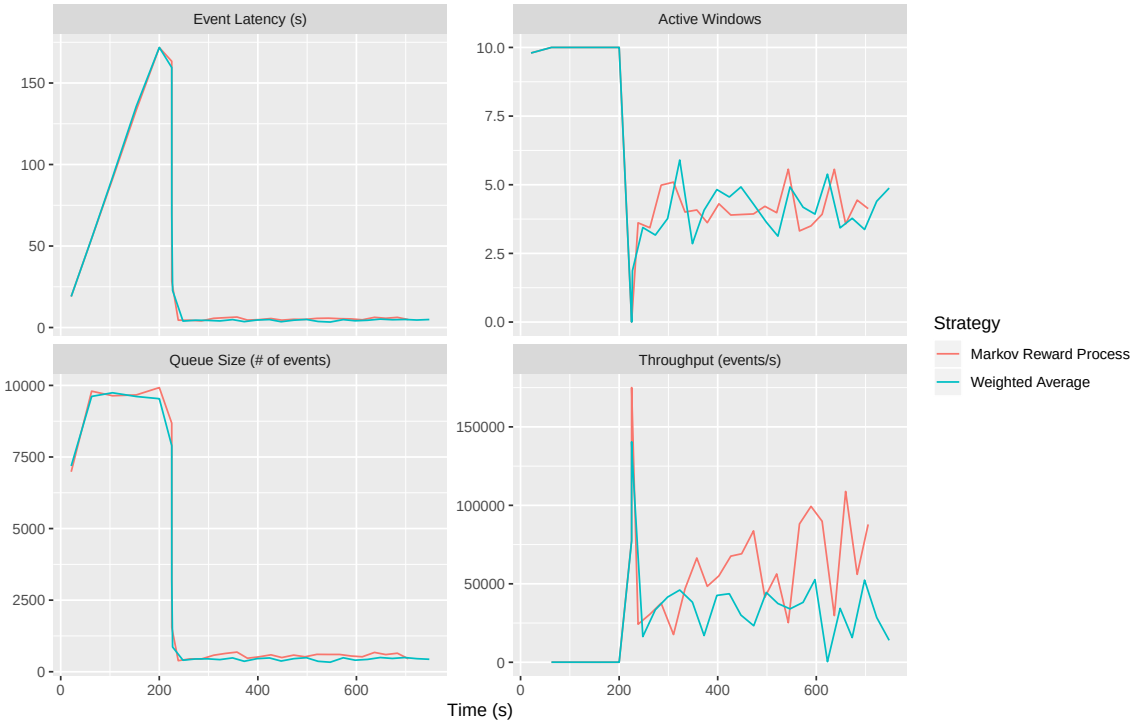
Experiment Regular	Markov Reward-Cost Process	Weighted Average
Measure Phase duration	162.12	150.31
Complex Events	729	496
Utility	4.5	3.3
Dropped Windows	1840	2272

Figure 4.6: Results of the experiment for the regular operator.

and as such peaks out at the 100s mark. We can further observe, that the window dropper keeps the event latency below the 10 second mark at all times after being activated. The number of active windows are approximiately kept at 5. The reason is that this is the amount of windows that can serve double the throughput of 10 windows. The experiment is setup to double the throughput as soon as the warumup phase starts and as such, the window dropper needs to drop so many windows to keep the latency bound satisfied. In the final results, the markov reward cost process detects more complex events than the weighted average strategy, however also at the cost of a longer duration. Utility is the the number of complex events divided by the measure phase duration.

The results of the fork experiment are similiar as shown in Figure 2.3. Interestingly, the markov reward-cost process manages a higher throughput despite having a similar

amount of active windows as the weighted average strategy. In the fork experiment, one path is a lot more expensive in terms of processing cost than the other. The markov cost-reward process strategy is able to actively exploit this knowledge in order to drop the expensive windows and thus having a higher throughput.



Experiment Fork	Markov Reward-Cost Process	Weighted Average
Measure Phase duration	351.6	376.14
Complex Events	1523	1299
Utility	4.3	3.4
Dropped windows	2674	2699

Figure 4.7: Results of the experiment for the fork operator.

Figure 4.8 shows how the markov reward-cost process strategy has evaluated the utility of each state in the long run. The utility statistics were gathered from the window dropper. Whenever a window dropper was evaluating a window in a specific state, it would calculate its utility and note it down. The figure shows the result of the average utilities in these states over the course of the experiment. The values are scaled down by a factor of 100 for easier readability.

We can observe, that it clearly values the top states more than the bottom states, as they induce less processing cost. Also we see, that it values a window more, the closer it is to a state which matches a complex event.

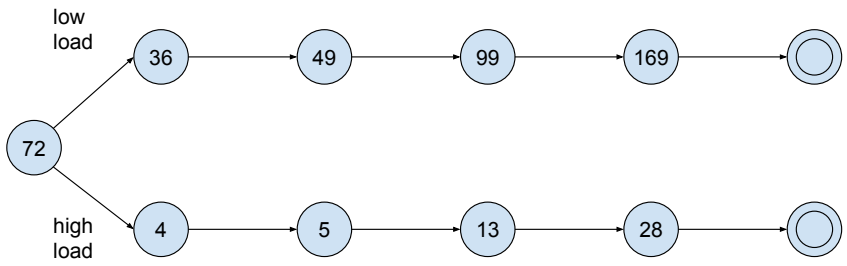
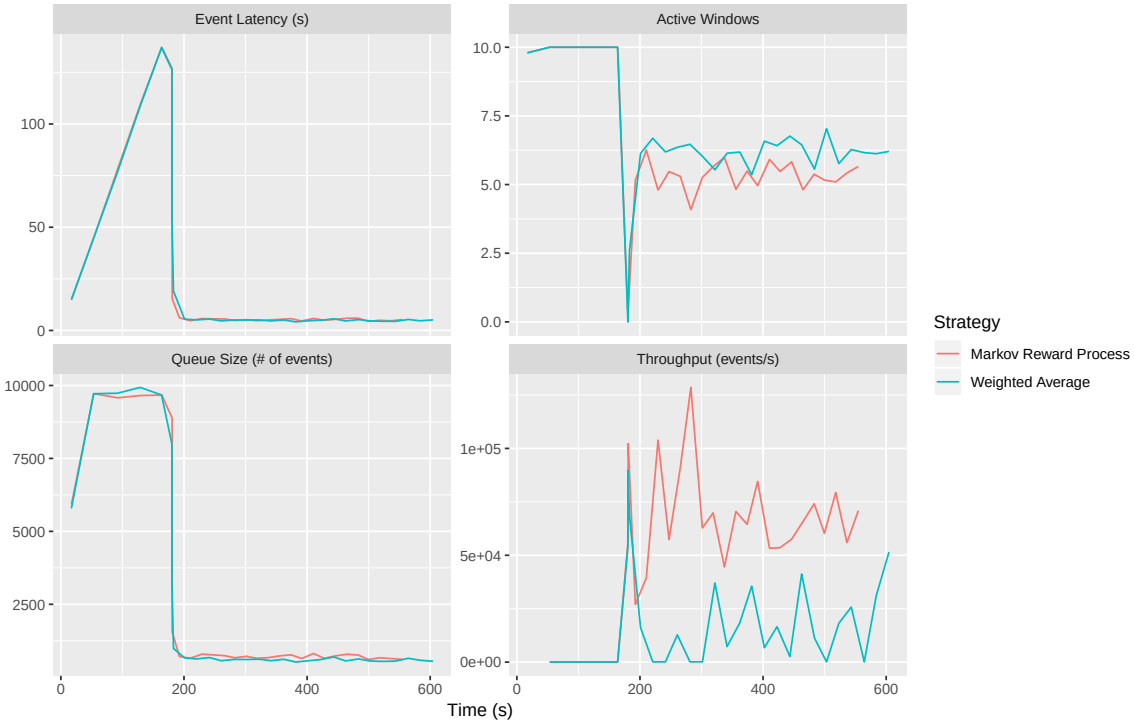


Figure 4.8: Utility estimation of each state according to the markov reward-cost process

The last experiment is the experiment for the limited chronicle operator. The results are shown in Figure 4.9. The results are a bit surprising, as the markov reward-cost process clearly outperforms the weighted average strategy. Judging by the evaluation results in section 4.2, we would expect the results of the experiments to be a bit closer to each other. However, the markov-reward cost process strategy is able to maintain a higher throughput despite having a slightly lower average count on active windows. It has also managed to drop less windows than the weighted average counterpart.

The results of the evaluation show the feasability of the window dropper algorithm. The window dropper is able to keep the event latency below the threshold and can decide to drop less favorable windows when the event latency threatens to be violated.

The markov reward-cost process has also shown to be viable in the case of window operators which can be modelled as finite state automata and proven to be superior than the weighted average strategy in these experiments.



Experiment Chronicle	Markov Reward-Cost Process	Weighted Average
Measure Phase duration	275.24	303.81
Complex Events	777	578
Utility	2.8	1.9
Dropped windows	1943	2081

Figure 4.9: Results of the experiment for the chronicle operator.

4.4 Threats to Validity

In this section we discuss the threats to the validity of our evaluation and to the approach in general. Internally speaking, we have the threat of measure bias.

Measure bias As complex event processing is a highly dynamic system, many factors can influence the performance of the system. We have chosen arbitrary parameters for our experiments which can influence how the window dropper performs. E.g. choosing a very high window size or a very high slide size can negatively impact the performance of the window dropper.

From the external validity, there are two main issues: (1) no real world data and (2) exploration vs exploitation issue.

Real World data we conducted the experiment on synthetic experiments only and showed there the viability of the approach. In future work, it could be a viable point to expand the experiments to a real world example like stock market. Basing the work on real world examples might yield important insights on how to improve the algorithm.

Exploration vs Exploitation If we take the approach for estimation of windows using the Markov Reward Process (MRP), then there is concern regarding the exploration/-exploitation issue. As the MRP estimates the values for the windows, at the same time, windows are being dropped by the Window Dropper using these values. Therefore, if we continue to drop bad windows and continue keeping good windows, we will gain less information on the bad windows for the markov model learner. In the case that the distribution of event types in the event stream changes, we will not be able to react accordingly as the windows responsible for detecting these events are dropped before they come to the relevant state.

However, this issue is alleviated as our window dropping algorithm is inherently probabilistic. Additionally, we provided a way to further improve this situation by choosing a greedyness function and patience function. The greedyness function can be used to amplify or dampen the exploitation and the patience function can be used to adjust exploration.

Chapter 5

Conclusion

In this chapter we give a quick summary over the discussed topics. We also give recommedation on future work to expand on this topic.

5.1 Summary

We have given several contributions during this work. (1) we provided an estimation model and showed that it is viable to use for a subset of CEP operators. (2) we discussed the concept on how to calculate the residual latency of a operator instance and (3) provided an algorithm which probabilistically decides the windows to drop.

The estimation model uses a with additional cost extended markov reward process. With value iteration we solve the model to get for each state a list of expected reward and cost the window will induce after a given number of events have been processed. This is a powerful tool and has shown itself to be quite accurate, as long as the underlying model of the window operator fits.

With the residual latency estimator we found a way to estimate the current load on the system and map the overload into a value which decides the amount of windows that need to be dropped. The challenge here was to infer the value from the current throughput and eliminate the resulting feedback-loop.

The window dropper is designed to be leightweight in favor of precise and heavyweight solutions like solving for knapsack. Instead we use probabilisitic drop probabilities for each window according to its expected utility, calculated from its expected amount of complex event and cost. Additionally we consider the processing time the window

has left and weigh it against the drop time boundary of the given drop request and incorporate this into the drop probability.

Complex event processing is a highly dynamic paradigm in which each factor influences many others. It is challenging to find an approach that works consistently in all situations and under any assumptions. We limited our scope to a narrow field in order to find an algorithm under this specific scope.

As to the best of our knowledge, there is no approach yet which sheds load by dropping windows in the context of complex event processing, the contributions made in this thesis can serve as a starting point in this area of research.

5.2 Future Work

The proposed approach has several limitations that may not be feasible in practical uses. The hardest limitation in practice, is that we follow a whitebox approach in which we can see the state of the pattern matching mechanism. In the real world, this is not nearly as easily possible, as normally operators execute some user specific pattern matching mechanism. The user would need to carefully design the pattern matching mechanism in order to be able to use this approach for window drop load shedding. As such, it would be intersting to research other options to guess the internal state of the window operator, i.e. a blackbox approach. Using techniques from machine learning one might be able to synthesize a representation of the internal state machine by observing the cost the window operator induces during operation, as this is the only feedback we would still get even in a blackbox scenario.

Futhermore, there is a limitation that only operators can be used which can be reduced to a finite state automaton. This is not entirely true, as we showed with the experiment for the limited chronicle operator that we can also use other types of operators if we there is the possibility to compress or downsample the state space. It can be interesting to expand on this area and see, how far this concept can hold.

Appendix

Bibliography

[BDWT13] C. Balkesen, N. Dindar, M. Wetter, N. Tatbul. “RIP: run-based intra-query parallelism for scalable complex event processing.” In: *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM. 2013, pp. 3–14 (cit. on p. 10).

[CJ09] S. Chakravarthy, Q. Jiang. *Stream data processing: a quality of service perspective: modeling, scheduling, load shedding, and complex event processing*. Vol. 36. Springer Science & Business Media, 2009 (cit. on p. 4).

[FTR+10] L. J. Fülöp, G. Tóth, R. Rácz, J. Pánczél, T. Gergely, A. Beszédes, L. Farkas. “Survey on complex event processing and predictive analytics.” In: *Proceedings of the Fifth Balkan Conference in Informatics*. Citeseer. 2010, pp. 26–31 (cit. on pp. 2, 10).

[HBN13] Y. He, S. Barman, J. F. Naughton. “On load shedding in complex event processing.” In: *arXiv preprint arXiv:1312.4283* (2013) (cit. on p. 11).

[Hed17] U. Hedtstück. *Complex Event Processing: Verarbeitung von Ereignismustern in Datenströmen*. Springer-Verlag, 2017 (cit. on pp. 3, 6, 10).

[KBF+15] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, S. Taneja. “Twitter Heron: Stream Processing at Scale.” In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. Melbourne, Victoria, Australia: ACM, 2015, pp. 239–250. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742788. URL: http://doi.acm.org/10.1145/2723372.2742788 (cit. on p. 10).

[Luc02] D. Luckham. *The power of events*. Vol. 204. Addison-Wesley Reading, 2002 (cit. on p. 10).

[May18] R. D. Mayer. “Window-Based Data Parallelization in Complex Event Processing.” Englisch. Dissertation. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, März 2018, p. 189. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIS-2018-02&engl=0 (cit. on pp. 10, 16).

[MKR15] R. Mayer, B. Koldehofe, K. Rothermel. “Predictable Low-Latency Event Detection with Parallel Complex Event Processing.” Englisch. In: *IEEE Internet of Things Journal* (Jan. 2015), pp. 1–13. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2015-02&engl=0 (cit. on pp. 9, 10).

[MST+17] R. Mayer, A. Slo, M. A. Tariq, K. Rothermel, M. Gräber, U. Ramachandran. “SPECTRE: supporting consumption policies in window-based parallel complex event processing.” In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM. 2017, pp. 161–173 (cit. on p. 10).

[MTR16] R. Mayer, M. A. Tariq, K. Rothermel. “Real-time Batch Scheduling in Data-Parallel Complex Event Processing.” In: *Techn. Ber. Technical Report* 4 (2016), p. 22 (cit. on p. 10).

[Owe07] T. J. Owens. *Survey of event processing*. Tech. rep. Air force research lab ny information directorate, 2007 (cit. on p. 10).

[TZ06] N. Tatbul, S. Zdonik. “Window-aware load shedding for aggregation queries over data streams.” In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment. 2006, pp. 799–810 (cit. on p. 10).

All links were last followed on October 25, 2018.

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