Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

The Influence of Load Shedding on the Stream Quality in CEP Applications

Marc-Sven Wunschik

Course of Study:	Informatik
Examiner:	Prof. Dr. K. Rothermel
Supervisor:	Henriette Röger, M.Sc.
Commenced:	June 20, 2019
Completed:	December 20, 2019

Abstract

Both the Digitalisation and Industry 4.0 result in continuous data streams in all areas of life. To get high-level information out of these streams in real-time, we can use Complex Event Processing (CEP) and Stream Processing (SP). This is often done on several operator nodes on different sites. Each of these operators works on a fraction of the whole information gathering process. There are different types of operators for different tasks. The most important for this work are the propositional logic operator, the pattern-finding operator, and the average-building operator.

The operators are organized in a directed acyclic graph, called the operator graph. Often the operator graph has to deal with great amounts of continuously incoming data. The data is bundled in discrete events. We can use load shedding to reduce the workload and to ensure that the operators process incoming data quickly. Load shedding removes some of the events from the waiting queue at an operator. As such it leads to lower latency and workload. But it can also lower the accuracy of the found information.

Most publications on load shedding in CEP or stream processing examine, how to make good load shedders. But in this work, we examine how inaccuracies caused by load shedding propagate through the operator graph. This means how load shedding at a preceding operator can influence the current operator. We examine which operator types can potentially repair the inaccuracies in the data stream caused by prior load shedding. That means that the data stream is more accurate after the operator than it was before. As a baseline for comparison, we take a run of the same scenario without load shedding.

To do this, we created a Java program that can simulate CEP and stream processing for different scenarios, using different operator types. For the experiment, we apply load shedding to the operators in the operator graphs of the different scenarios. In the evaluation, we explain how and why load shedding affects the different operator types in the way it does. We give guidelines on how to optimally shed load for each operator type.

If the propositional logic operator has an input event type that is the limiting factor it reacts well to load shedding of the non-limiting factor. If the limiting factor is greatly outnumbered, this operator type can even repair the accuracy of the data stream.

The accuracy of the pattern-finding operator is very negatively affected by load shedding, if we shed whole input events. If at some point in the operator graph we can shed load in such a way that only the attributes of the input events of the pattern-finding operator are altered, the resulting accuracy is much better. This could happen if we e.g. aggregate events prior to the pattern-finding operator. There is no potential here to improve the accuracy of the data stream.

The average-building operator has very high accuracy when we use load shedding. We need to do the load shedding in a way that we shed all input data for the average-building operator proportionally. If that is the case the accuracy remains on average at 100%.

There is only a small variance for higher load shedding drop rates. This operator type can, therefore, repair a prior loss of accuracy of the data stream almost entirely.

Abstract- German Translation:

Die Digitalisierung und Industrie 4.0 sorgen für kontinuierliche Datenströme in allen Lebensbereichen. Um Informationen aus diesen Streams in Echtzeit zu erhalten, können wir Complex Event Processing (CEP) und Stream Processing verwenden. Dies geschieht häufig an mehreren Operator-Knoten an verschiedenen Standorten. Jeder dieser Operatoren bearbeitet einen Bruchteil des gesamten Informationserfassungsprozesses. Es gibt unterschiedliche Typen von Operatoren für verschiedene Aufgaben. Die wichtigsten Typen für diese Arbeit sind der Aussagenlogik-Operator, der Musterfindungsoperator und der Durchschnittsbildungsoperator.

Die Operatoren sind in einem gerichteten azyklischen Graphen organisiert, der als Operatorgraph bezeichnet wird. Oft muss der Operatorgraph mit großen Mengen kontinuierlich eingehender Daten umgehen. Die Daten werden oft in diskreten Events gebündelt. Mit Load-Shedding können wir den Arbeitsaufwand reduzieren und sicherstellen, dass die Operatoren eingehende Daten schnell verarbeiten. Durch das Load-Shedding werden einige Events aus der Warteschlange eines Operators entfernt. Dies führt zu einer geringeren Latenz und Arbeitsbelastung. Es kann aber auch die Genauigkeit der gefundenen Informationen verringern.

In den meisten Veröffentlichungen zum Load-Shedding in CEP oder Stream Processing wird untersucht, wie man gute Load-Shedder herstellt. In dieser Arbeit untersuchen wir jedoch, wie sich durch Lastabwurf verursachte Ungenauigkeiten im Operatorgraphen ausbreiten. Das bedeutet, wie ein Load-Shedding bei einem vorhergehenden Operator den aktuellen Operator beeinflussen kann. Wir untersuchen, welche Operatortypen möglicherweise die Ungenauigkeiten im Datenstrom beheben können, die durch vorheriges Load-Shedding verursacht wurden. Das heißt, dass der Datenstrom nach dem Operator genauer ist als zuvor. Als Vergleichsbasis nehmen wir genau das gleiche Szenario ohne Load-Shedding.

Zu diesem Zweck haben wir ein Java-Programm erstellt, das CEP und Stream Processing für verschiedene Szenarien unter Verwendung verschiedener Operatortypen simulieren kann. Für das Experiment wenden wir Load-Shedding auf die Operatoren in den Operatorgraphen der verschiedenen Szenarien an. In der Auswertung erklären wir, wie und warum sich das Load-Shedding auf die verschiedenen Operatortypen auswirkt. Wir geben Richtlinien an, wie die Load-Shedding für jeden Operatortyp optimal betrieben werden kann.

Hat ein Aussagenlogik-Operator einen Event-Typ, der der begrenzende Faktor ist, reagiert er gut auf das Abwerfen des nicht begrenzenden Faktors. Wenn der begrenzende Faktor stark in der Unterzahl ist, kann dieser Operatortyp sogar die Genauigkeit des Datenstroms reparieren.

Die Genauigkeit des Musterfindungsoperators wird durch den Load-Shedding sehr negativ beeinflusst, wenn ganze Input-Events abgeworfen werden. Wenn wir an einer Stelle im Operatorgraphen die Last so reduzieren können, dass nur die Attribute der Input-Events des Mustersuchoperators geändert werden, ist die resultierende Genauigkeit viel besser. Das kann z.B. geschehen, wenn wir vor dem Musterfindungsoperators Events aggregieren. Hier gibt es kein Potenzial, die Genauigkeit des Datenstroms zu verbessern.

Der Durchschnittsbildungsoperator hat eine sehr hohe Genauigkeit, wenn wir den Load-Shedding verwenden. Wir müssen das Load-Shedding so durchführen, dass wir alle Input-Events für den Durchschnittsbildungsoperator proportional abwerfen. In diesem Fall bleibt die Genauigkeit im Durchschnitt bei 100%. Es gibt nur eine geringe Varianz für höhere Load-Shedding-Raten. Dieser Operatortyp kann daher einen vorherigen Genauigkeitsverlust des Datenstroms nahezu vollständig beheben.

Contents

List of Figures

1 Introduction

Many modern IT solutions generate continuous streams of data. Be it user transaction of online retailers, internal processes of big companies or sensor data in a building, in all these scenarios we have a continuous stream of data. This data stream is a series of events, e.g. one sensor data value per event.

We use Complex Event Processing (CEP) to extract information out of these data streams in near real-time. CEP is a method of analyzing data streams and deriving information from it. In CEP we implement a query through a directed, acyclic graph of operators. There are different types of operators that we can use, e.g. to filter, to aggregate, to analyze or to detect patterns. A complex event consists of a combination of several events. The events can have different types and sources. The goal of CEP is to detect complex events. To do this we sent the data stream through our operator graph. The output of the previous operators is the input for the following ones. If the last operator of the graph detects a complex event, we have found the desired information.

Often it is not only important that we obtain the information, but also that we get it on time. Thus it is important, that the processing of incoming events at the operator has low latency. If e.g. CEP uses sensor data to detect a fire, any delay in the detection caused by high latency could be very costly. But including outdated data in the processing could lead to false alarms. To ensure the timeliness of the information we need to take measures. If we keep the latency low, this also ensures that the operators are not overloaded and potentially crash.

One mechanism to decrease workload and latency is parallel processing [RBR19; RM19]. In high load situations, we create new operator instances on multiple processing nodes. The workload is then split between the parallel operators. This reduces the workload of individual operators. But there are situations, where we do not have the necessary resources for parallelization. In this thesis, we will not concentrate on parallelization. Instead, we look at another mechanism called load shedding. Load shedding reduces the workload at operators and can ensure low latency.

Load shedding drops events from the input queue of an operator in case of high workloads. This ensures that the operator can process the remaining events with low latency. The dropping of events can negatively impact the result. This can lead to inaccurate end results. In an operator graph with multiple operators, it is important, to look for the best location to perform load shedding at. The correct choice of load shedding location can minimize resulting inaccuracies. It can also result in a greater reduction in latency per dropped event.

When speaking about inaccuracy it needs to be specified, that in this thesis we are not looking at the inaccuracy of incoming data. Imprecision of the rules at the operators that can lead to wrong results is also not part of this thesis. Instead, we investigate how event shedding at different operators can affect the stream quality and the end result. The stream quality is a measure of how accurate the data stream is between the operators. At 100% stream quality, the processing operator detects all possible matches in the data stream. When we perform load shedding we drop some of the events needed for matches. As a result, we find fewer matches and the stream quality is thus lower than 100%. The goal when using load shedding is to keep stream quality as close as possible to 100%.

Upstream load shedding can influence the quality of the result downstream, but it is challenging to understand the dependencies. The inaccuracy of the stream can lead to false positives or false negatives. That can in turn influence the following operators through faulty input data. This can lead to wrong end results (e.g. a faulty detected match). Especially interesting is, how good the stream quality and the quality of the end result are. To get a measure for this we compare them to a run of the same input data on the same operator graph, without load shedding or with load shedding applied at different operators.

We analyze these dependencies using a representative application. We have three main operator types that we want to examine: A propositional logic operator, a pattern-finding operator, and an average-building operator. Each of these operators has its own operator graph, where it is the sink. In each operator graph, we have several operators before the sink that prepare the data stream. At these operators, we can use load shedding and change the composition of incoming events at the sink. Then we can examine, how the sink reacts to these changes and how accurate the result is.

Questions we answer are how inaccuracies, that we cause through load shedding, propagate through the operator graph. After load shedding at an operator, how does the stream quality on the following operators develop? Does it remain the same, further decrease or even increase? Do certain types of operators propagate inaccuracies differently? Is it more advantageous to do load shedding before certain operator types? These are some of the main questions we examine in this thesis.

For our analysis, we extended a provided Java Framework [SBFR19; SBR19]. The Framework handles a CEP stream processing system with load shedding. It enables the use of different load shedding approaches. That program tests different scenarios of applying load shedding to an operator graph and captures the data about the stream quality. This data contains the output of operators and the dropped events. Additionally, we wrote Python scripts to analyze the stream quality and the accuracy of the end result.

Based on the evaluation the goal is to determine if we can find general rules for load shedding in a CEP system. The criteria for this are the resulting inaccuracies and the reduction in latency.

Structure

The work is structured in the following way:

Chapter 1 – Introduction: Introduces the topic and scope of this work.

Chapter 2 – Definitions: Defines concepts that are important in order to understand this work.

Chapter 3 – Related Work: Presents and summarizes publications related to the topic of this work.

Chapter 4 – Scenarios to Analyze Quality Propagation: Describes the program used for our experiment.

Chapter 5 – Evaluation: Evaluates the results of our experiments.

Chapter 6 – Conclusion: Summarizes the results of the evaluation.

Chapter 7 – Summary and Outlook: Summarizes the results of this work and presents starting points for future work.

Chapter 8 – Attachment: Contains figures that are referenced in this work, but were too numerous to be include in the text.

2 Definitions

In the following, we explain the concepts used throughout this thesis.

2.1 Events

Events generally have event types, which denote what kind of information they store. An event contains attribute fields with corresponding values. If necessary for its processing an event can also have a timestamp from the moment of the event's generation. This type of events are the input of an operator graph in stream processing or complex event processing. They are also called primitive events. An example of a primitive event would be the data of a temperature sensor, that periodically sends its current value to the first source of an operator graph.

2.2 Data Stream

In general, a data stream is a continuous flow of information. In the case of this work, where we focus on an operator graph that performs stream processing and complex event processing, we store the data in events. As such a data stream is a continuous flow of events through the operator graph.

2.3 Operator

The operator defines how events from the data stream are processed. The events arrive at the in-queue of the operator. In general the operator processes events after the principle First In -First Out. Each operator has a predefined task, that determines how it processes events. After the operator processed the events, it can generate an output event. This output is then sent on. The task an operator has and the type of output event it generates is different for each operator type. In this work, we have two big classes of operators: stream processing and complex event processing.

2.4 Operator Graph

The operator graph, also denoted as Ω, is the topology of the operators that are used to process an incoming data stream. An operator graph has a source, the operator the data stream arrives at. It also has a sink, the operator that generates the end result. The sink often sends the end result to a receiver outside of the topology. The most simple topology contains only one operator that is both source and sink.

In many cases, we need a more structured approach to process the data stream efficiently. To do so, we use several operators that are connected in a direct acyclic graph. The output of the preceding operator is the input of the current one. This enables us to run the operators on different machines as a distributed system.

2.5 Stream Processing

Stream processing can stand for many types of operations. We give a brief overview in the following.

An often used operator type is the filter operator. It passes on only a subset of incoming events according to predefined criteria. Another typical stream processing operation is the averaging of a certain attribute over all events in a window. The average is then sent in one output event to the next operator. Other cases can require to find the maxima or minima of an attribute over all events in a window. There can also be operators that prepare events for processing at following operators. E.g. by bringing the data an event contains, into an easier processable form.

2.6 Complex Event Processing

Complex Event Processing (CEP) processes incoming primitive events and analyze them. Out of numerous input events, it generates a small number of output events. Such an output event is called a complex event. A complex event combines the desired information of the primitive events that contributed to its detection into one. As such it contains information in a condensed form. CEP often provides the context that transforms raw information of a data stream into a usable form.

CEP searches for, e.g. patterns in the data. One such pattern could be the increase of an attribute value in five consecutive events. Another case of CEP is the application of propositional logic on the events of the data stream. For all events in a window on operator could e.g. search for the occurrence of one certain event type (A) and also the occurrence of another event type (B). Only if both of these event types are present in the same window does the operator generate an output event (C) and sends it on ($A \wedge B \rightarrow C$). Depending on the configuration, the operator can produce only one output event per window or several, if the condition is satisfied more than once.

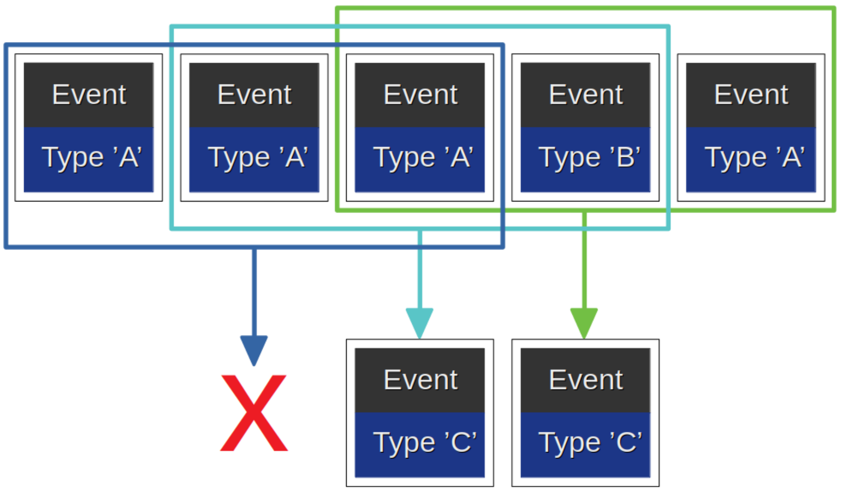


Figure 2.1: Windows with a slide of 1. For a match we need an A and a B in a window. We see, that both the green and light blue windows share the same B. Thus they detect a match and put a C out. The dark blue window does not include a B, it generates no output event.

2.7 Windows

Windows separate the continuous data stream into chunks that the operator then processes together. This is necessary because the data stream is potentially never-ending. As such an operation, like building an average over a certain attribute, cannot wait for all events to arrive. The operator always processes a certain number of events together. The size of a window can be predefined or dynamically changed at runtime. When using windows that do not overlap to process events we may miss some matches, because the matching events are in different windows. To counteract that we use windows that have an overlap or slide.

The slide parameter of a window determines how many events to consecutive windows overlap. In Figure 2.1 we see, that two windows that have a size of 5 and a slide of 1, have 4 events in common. In this case, every event is processed among other things together with all 4 preceding events and also together with all 4 following events. It is impossible to miss a match if we selected the window size appropriately for this operator. A result is that a found match can be potentially detected in several consecutive windows.

2.8 Load Shedding

We can use Load Shedding (LS) if an operator is overloaded, to reduce its workload. We can also use load shedding to reduce the latency in the processing of incoming events.

Load shedding drops a certain percentage of the events, that are in a window, before the processing. By using load shedding we reduce the latency and workload. But we also reduce the accuracy of the result we gain.

2.9 Random Load Shedding

Random Load Shedding (RLS) is load shedding that drops each event with an equal probability. The probability depends on the overall target drop rate.

2.10 Utility-based Load Shedding

Utility-based Load Shedding (ULS) is load shedding that drops events depending on their utility. The utility value represents how likely the event is to contribute to an output event. Events that are very likely to contribute have a high utility. Events that have a small change to contribute have a low utility. If we use load shedding, we preferably drop events with low utility. ULS offers an overall more accurate result than RLS when they have the same target drop rate. But ULS does generally have a higher overhead than RLS. The potential reduction in latency is, therefore, lower for ULS.

2.11 Stream Quality

To determine the stream quality we establish a baseline. This baseline is an average value for output events for an operator, while no load shedding is used.

The stream quality is the percentage of output events that an operator still sends, when load shedding is used.

E.g. operator 1 generates on average 100 events when no LS is used. Now we apply load shedding with 20% drop probability. Operator 1 now only generates 80 output events on average. The stream quality after operator 1 is therefore 80%.

2.12 Limiting Factor

The limiting factor is a relevant concept to analyze the influence of load shedding in certain operator types. Let us assume an operator needs two types of events for a match, e.g. A and B. If only a quarter of the input events are A and the remaining events are B, then A is the limiting factor. In this case, there are triple the amount of 'B's, thus 'A's are outnumbered three times at this operator.

3 Related Work

In this chapter we present a selection of papers on the topic of load shedding in CEP and stream procession systems.

3.1 On Load Shedding in Complex Event Processing

He et al. [HBN13] give an overview of load shedding in CEP. They describe the data and query model used in CEP systems. For load shedding, it describes what utility values are. Utility values are the importance of a primitive event towards detecting a complex event. In the case that load shedding necessary, operators drop events according to their utility. Operators drop events in such a way, that the overall utility is maximized. This results in the maximum amount of detected complex events.

The authors discuss two types of resource constraints, that can make load shedding necessary. These are CPU and memory constraints. After that, they define three types of load shedding, depending on the limiting resource. These are memory-bound, CPU-bound and dual-bound (both CPU and memory constraints) load shedding.

The authors also define two types of load shedding mechanisms. These are integral and fractional load shedding. In integral load shedding an operator drops certain incoming event types or even whole queries. The operators do that after the principal all or nothing. In fractional load shedding the operator analyses a random subset of incoming events. Based on that, the operator drops a fraction of certain event types and or queries. In the next part He et al. discuss integral and fractional load shedding for all the above-mentioned resource constraints. These are memory-bound, CPU-bound and dual-bound constraints. Firstly the discuss Integral Memory-based Load Shedding (IMLS). IMLS works by solving an optimization problem. It maximizes the product of the number of expected query matches and their utility value. It does that while not violating the memory constraints. The Fractional Memory-based Load Shedding (FMLS) works similar to IMLS. An optimization problem is solved, while not violating the memory constraints. The difference is, as described above, which part of the input event stream the operators drop. Thus the optimization problem is built differently.

The CPU-bound load shedding variants Integral CPU-bound Load Shedding (ICLS) and Fractional CPU-bound Load Shedding (FCLS) work similar to their memory-bound counterpart. The only difference is the type of limited resource. The dual-bound variants Integral Dual-bound Load Shedding (IDLS) and Fractional Dual-bound Load Shedding (FDLS) are also only different in the type of resource constraints.

For all these variants He et at. give a complexity analysis (they are NP-hard), approximations and further theorems including proofs.

3.2 eSPICE: Probabilistic Load Shedding from Input Event Streams in Complex Event Processing

Slo et al. [SBR19] introduce a probabilistic load shedder called eSPICE. eSPICE can be used for window-based CEP-frameworks. In this paper, the authors assume, that there are infinite resources available. Thus the goal of this load shedder is not to reduce the strain on the system and its components. The goal is to ensure that certain latency bounds are not violated while processing events. This is necessary because in CEP detected complex events often need to be found in a certain time interval. This time interval starts from the moment of the creation of the primitive events. It ends with the detection of the complex event. In many applications, it is better to have less accurate current data, than to have totally accurate, but outdated data.

At the application start, eSPICE learns from the processed data and builds a probabilistic model. To do this it analyses which events contribute to a complex event and its position in its window. Both the type of an event, as well as its position in a window are important for the overall utility value of an event. The position of the event in a window is important because the authors assume that in a CEP system operators are of the form $A \wedge B \rightarrow C$, with A and B primitive input event types. The authors also assume, that only one complex event can be detected per window. In this case, only the first A in a window would contribute to the complex event C. Any following A in a window would not contribute. As a result following $'A'$s would have a low utility value.

Load shedding at an operator in eSPICE starts, when a predefined percentage of the maximum capacity of the input queue is filled. If this is the case, the operator drops several primitive events from the input queue. The operator does this, to ensure that it can process incoming events promptly. It ensures that the latency bound is not violated. The detected complex events are thus still relevant once the operator detects them.

The authors also evaluated eSPICE. The result is, that eSPICE is, after the learning phase, at the beginning, an efficient and lightweight load shedding framework. With only a small overhead it maintains the set latency bound while delivering a result with only a small loss of accuracy.

In the test, the authors compare eSPICE to state-of-the-art load shedder [cite] for CEP systems. eSPICE outperforms them.

3.3 Concept-Driven Load Shedding: Reducing Size and Error of Voluminous and Variable Data Streams

Katsipoalakis et al. [KLC18] introduce the approach of Concept-Driven Load Shedding. The authors call the relevant part of the information in a data stream the concept. They postulate that the concept of a data stream can change over time. This change in concept leads to shrinking accuracy when using uniform load shedders. Uniform load shedders use apriori knowledge to calculate utility values. Then they use these utility values to create the load shedding function. From that point on the load shedding function is static. As such it cannot react to concept drift. An example would be the problem of finding the top routes of taxi traffic in a city, from a data stream. The top route can change drastically over the span of the day or be affected by other factors like the season or roadworks. Using a filter created with knowledge from a morning in winter, on the data stream containing the data from a night in summer, can lead to great inaccuracies.

The Concept-Driven Load Shedding Algorithm divides the input stream into overlapping time windows. For each window, it estimates the context by sampling the data from the data stream from this period. It does that by applying the query in question to it. The information gathered from this is used to filter the stream of the corresponding window. As such the filter is always fitting for the concept.

Concept-Driven Load Shedding scales comparable to uniform load shedder. In terms of performance, it is equal at worst or significantly better at best. The accuracy is equal to uniform load shedders when there is no concept drift in the data stream. Concept-Driven Load Shedding is more than an order of magnitude better if a concept drift is present. It achieves more than 90% accuracy.

3.4 THEMIS: Fairness in Federated Stream Processing under Overload

The paper [KFSP16] from Kalyvianaki et al. Introduces THEMIS, a federated stream processing system (FSPS) for resource-constrained hardware. A FSPS splits its queries into query fragments. These query fragments can then be executed on multiple sites. FSPSs often suffer from overload. Load shedding in an FSPS is hard because the different sides that process the query fragments are autonomous. There is no centralized entity that can coordinate load shedding.

The goal of THEMIS is to provide a globally fair load shedding algorithm for FSPSs. For that purpose, it uses the concept of Source Information Content (SIC). SIC is the value a tuple has towards a query result. The authors base this value on the amount of source data that went into the creation of this tuple. As a result, highly aggregated data is valued more highly. The SIC value is query-independent. The idea is, to balance the SIC values for each query on every node. THEMIS implements the BALANCE-SIC algorithm. The

BALANCE-SIC algorithm always keeps the least amount of tuples for every query to reach a certain SIC-threshold. This balances the SIC values for each query on a node. Ever site in a FSPS implements that. The result is a globally fair load shedding algorithm.

In the evaluation BALANCE-SIC scores 33% higher than random load shedding on the Jain's Fairness Index [Raj16]. It is also fairer than [ZXLT10] and outperforms [TÇZ07]. The overhead of BALANCE-SIC is also small. The authors measured only an increase of 11% in runtime compared to random load shedding.

3.5 Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing

Tatbul et al. [TÇZ07] introduce three solutions on how to shed load. The first is a distributed load shedding algorithm called D-FIT (Distributed Feasible Input Table). The second is a centralized version of this algorithm called C-FIT. The third is a linear programming solution (Solver).

These approaches focus only on CPU restriction. They do not address potential memory overloads. All mentioned approaches have in common, that they calculate load shedding plans off-line. The approaches are all based on the same basic principle. They use available metadata about the system to calculate plans off-line. This enables to quickly determine of best load shedding plan at run time. Then they use these plans to determine where and what to drop in case an overload occurs. During the run time each operator estimates its load level, using samples of the data stream. If a node detects an overload, it selects the most beneficial load shedding plan. The node then executes the according drop operations.

The distributed approach structures the operator graph as a tree. Each node calculates its own Feasible Input Table (FIT). It contains all possible input combinations that do not cause a CPU-overload. This FIT is afterwards sent to its parent node, which merges the FIT of all its children. The parent node also removes all input combinations that are infeasible for itself. This FIT is then sent to the parent of the current node. Through its FIT a node can perform load shedding for itself and all its descendants. This requires no further communication.

The centralized approaches perform the planning at one dedicated coordinator node. The coordinator also decides when and where load shedding happens. It sends the load shedding order to the nodes in question. The nodes use the most beneficial load shedding plan available. C-FIT works similar to D-FIT. The difference is that all FITs are sent to the coordinator node. The coordinator plans and controls the whole load shedding process.

The solver based approach formulates the load shedding problem as a linear optimization problem. This approach groups overload causing inputs, that do not differ too much, together. For this group, a load shedding plan is calculated, through solving the linear program. This plan can be used for the whole group, without exceeding a certain error percentage. If an overload situation occurs, the node in question selects the load shedding plan for this input and executes it. The authors test only the plan generation time. They did not perform a comparison to other load shedding algorithms.

The same author also wrote several other papers [Tat02; Tat07; TÇZ+03; TZ06] on the topic of load shedding.

4 Scenarios to Analyze Quality Propagation

In the following chapter, we take a closer look at the program that we used for our experiment. At the beginning we take a look at the basic architecture. Afterwards we present the three different scenarios that were used in the experiment. The scenarios were designed to allows us to influence the composition of input events at different operators in several ways. This allows us to examine the propagation of losses in stream quality in various ways.

4.1 Program Architecture

For the experiment in this thesis, we modified the provided Java Framework [SBFR19; SBR19]. The framework could not run several nodes in parallel. This was a feature that was necessary for my experiment. Additionally, we programmed three different scenarios for load shedding and everything needed for their evaluation, from scratch. The scenarios are described at a later point. We describe in the following the basic architecture of the Java Program, that we used for the experiment. It gives insight into how we obtained the data for the evaluation.

As seen in Figure 4.1 the program starts a splitter, executor and merger for every operator in the operator graph. The operator graph is also denoted as Ω. If the operator is a source in the topology of Ω, it starts a source as well. Each of these separate components of an operator has its own thread. The program starts operators and all their internal components in inverse order. This guarantees that at the moment the first operator of Ω starts running, all other operators are already online and ready to process incoming events. The structure of Ω is determined by the connections between the single operators. On creation, the operator is passed a reference to the splitter of the following operators. The communication between the components happens via queues. We use the receiveElement() method of the class Runable for that purpose.

Depending on the scenario run in the program, the source either reads in data from a file or generates random data values. The source uses the so obtained data to create events, that are then sent to the splitter. Each scenario has its own event type(s). This is necessary because all scenarios have events with different attributes. After the source sent all events, it sends an EndOfStream (EOS) event. The EOS signals to the following operators, that no more events will be sent over this connection.

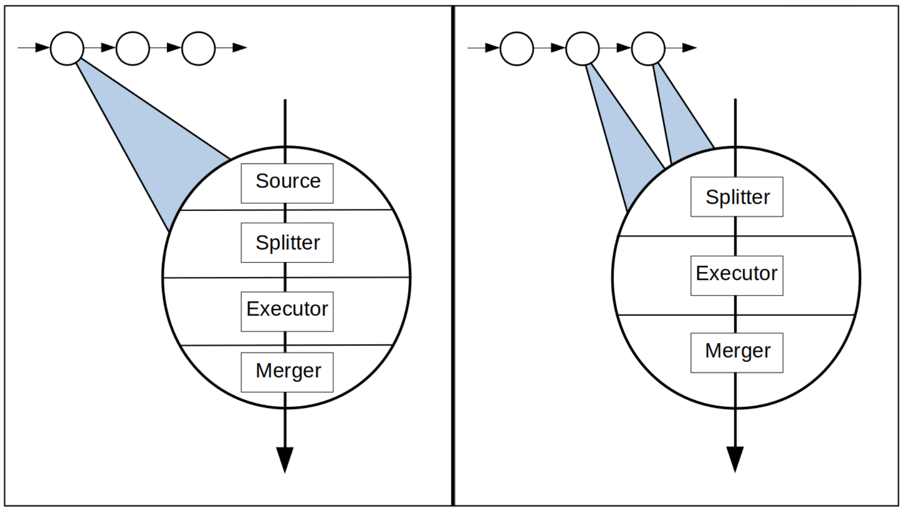


Figure 4.1: Program Architecture. On the left we see the internal structure of the source of Ω. On the right we see the internal structure of every operator, that is not the source of Ω.

The splitter distributes the events to the windows. The windows are necessary for the later processing of events. The splitter sends the start and end of a window to the queue of the executor. The splitter sends the events belonging in this window between these two markers as well. The windows can overlap. As such events can be in multiple windows at once.

The executor processes the events. It processes each incoming event for every window that contains it. If load shedding is active, it happens here in the executor before the processing of the event. For the processing of an event, the executor applies a pre-defined logic on the incoming events for a window, e.g. pattern finding. If the necessary conditions are met (e.g. pattern detected), the executor generates output events and sends them to the merger. The logic that is used to process events is dependent on the scenario currently running and on the operator id. The operator id signals the position of the operator in Ω. The program defines Ω on creation, after that it is static.

The merger receives incoming events and forwards them to the next operator(s) in Ω. If the operator is the sink of Ω, the merger sends no events. In that case, it terminates the program once it reaches the end of the stream. The merger can recognize when all events have been processed, by the number of EOS events that it has received.

At several spots, the program writes data to file for evaluation. As data format we used the CSV-format, to simplify evaluation. The program captures the output of each operator's executor and writes a statistic of it to file. It also captures and writes to file the overall runtime of the program.

The program offers the option to start each operator as an individual program. In this case, the program writes all outputs of the operator's executor completely to file, including a timestamp. When the program starts the next operator, it reads in the output of the previous operators from the file. Using the timestamps, it sends the events, as close as possible to the order, they would have in the standard execution of the program. The operator can never reach this order completely if one or more operators have multiple input files, that they need to read in. The timestamps used to order the events can only be used to order the incoming events of one source completely. If there are two or more sources the operator can not reconstruct the order of the events from different sources completely. It can only approximate it. This results in a significantly different result than running single-node mode, compared to the standard execution (all in one program).

4.2 Operator Types

In the following, we introduce the operator types we use in the three different scenarios.

4.2.1 Filter Operator

An operator that only sends incoming events on, if they a certain value in a certain attribute.

4.2.2 Forwarding Operator

This is a filter operator, that forwards all its input events.

4.2.3 Merge Operator

An operator that merges input events that are in the same window, if they fulfill certain conditions. One condition could be the same value for an attribute.

We have two types of merge operators in this work. One type merges all events in a window that can be merged. Then it sends on all events, even those that could not be merged. The other type has a restriction, it sends only the results of a merge on.

4.2.4 Propositional Logic Operator

An operator of the form $A \land B \to C$. It gets events of type A and B as input. If there are both a A and a B present in the same window, the operator generates an output event C. In this work, the propositional logic operator can only generate at most one output event per window.

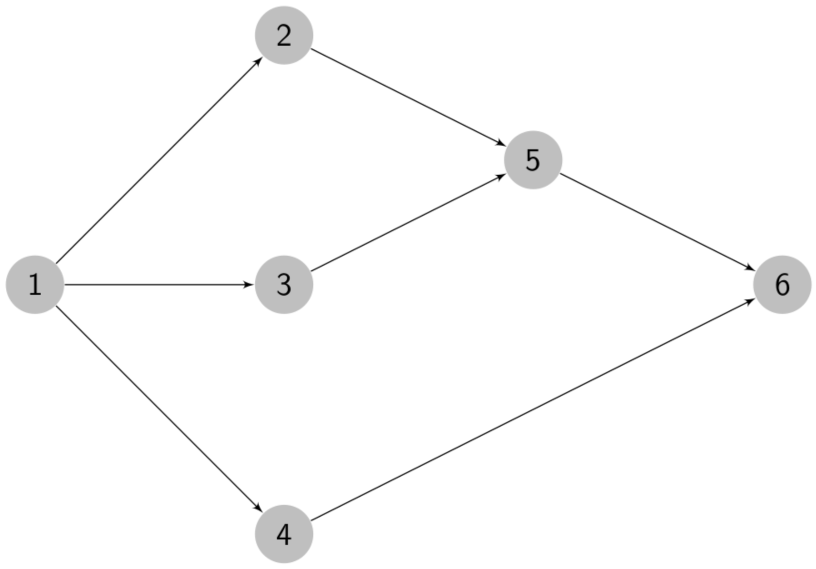


Figure 4.2: The operator graph of the Fire Scenario.

4.2.5 Average-building Operator

An operator that builds an average over a certain attribute of all input events that are in the same window. It generates one output event per window.

We do not take the number of output events as the criterion for the average-building operator. This number would remain the same even for high drop probabilities. Instead we take the accuracy of the average as criterion for the stream quality.

4.2.6 Pattern-finding Operator

An operator that searches for a pattern in the attributes of incoming events.E.g. an operator that detects an increase in a certain attribute for a certain number of events in a row.

4.3 Fire Scenario

In this scenario, we want to detect a fire by using three different types of values. These values are randomly generated. The first value (type A) is the smoke value. The second value (type B) is the temperature. The last value (type C) is the humidity value. A fire should be detected if all three types have a value that exceeds a certain threshold. Furthermore, all these values need to be close to each other in time. This is necessary because only if all three types have high values at the same time, a fire occurs. Otherwise, a false alarm could happen if we group together values of different points in time. In

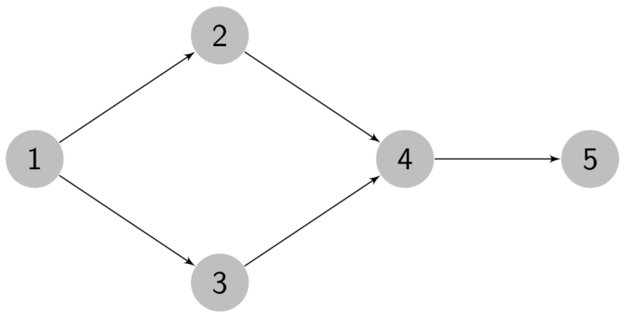


Figure 4.3: The operator graph of the Twitter Scenario.

the following, we discuss the structure of the operator graph Ω, that implements the Fire Scenario. We address each operator as ω, with the id it has in Figure 4.2.

- **ω_1:** This is the source of Ω. It generates the values of event types A, B and C. Events of type A are sent to ω_2. ω_1 sends events of type B to ω_3 and events of type C to ω_4.
- **ω_2:** ω_2 is a forwarding type operator. It forwards incoming events (of type A) to ω_5.
- **ω_3:** ω_3 is a forwarding type operator. It forwards incoming events (of type B) to ω_5.
- **ω_4:** ω_4 is a forwarding type operator. It forwards incoming events (of type C) to ω_6.
- **ω_5:** This is a propositional logic type operator. It receives events of type A and type B. ω_5 has count-based windows of size 10, that do not overlap. If in a window one A and one B exceed their threshold, an event of type D is send to ω_6. ω_5 can only detect one output event of type D per window.
- **ω_6:** This is a propositional logic type operator. It is the sink of Ω, it receives events of type C and type D. ω_6 has count-based windows of size 5, with a slide of 1. That means that windows always overlap in one event with the preceding window and also overlap in one event with the following window. If in a window a C that exceeds its threshold is present and also a D, ω_6 detects a fire. ω_6 can only detect one fire per window. Because ω_6 is the sink of Ω, events are not sent to another operator.

4.4 Twitter Scenario

In this scenario, we read in log files containing captured Tweets in the source of the operator graph. The captured Tweets were provided by the institute and are saved in Tweet JSON [Twi19]. The structure of a Tweet log file can be seen in Figure 4.5.

The structure of the Twitter Scenario operator graph Ω can be seen in Figure 4.3. Each operator processes incoming events differently. As such each operator produces different

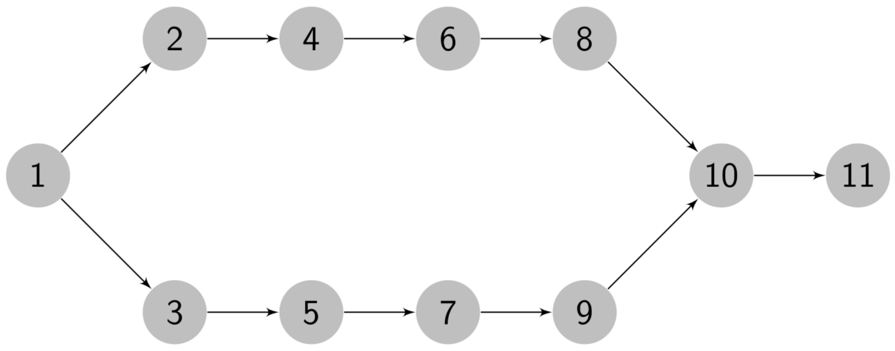


Figure 4.4: The operator graph of the Taxi Scenario.

output events. In the following, we address the operators as ω, with the id they have in Figure 4.3.

ω_1: ω_1 only forwards events representing a Tweet with an id string, that is divisible by two. It does not forward all other events.

ω_2: ω_2 is a filter type operator. It only forwards events with the language attribute ‘en’ for English. It does not forward all other events.

ω_3: ω_3 is a filter type operator. It only forwards events with a language attribute different than ’en’ for English, which also have a friend’s count greater or equal than 300. It does not forward all other events.

ω_4: ω_4 is a filter type operator. It forwards only events representing a Tweet that is a response to another Tweet. It does not forward all other events.

ω_5: ω_5 is an average-building type operator. ω_5 processes events in windows of a predefined size (here 100 events per window). It builds the average of the follower’s count of the incoming events. When the end of the stream is reached, ω_5 puts the average value into an output event. Because ω_5 is the sink of Ω, events are not sent to another operator.

4.5 Taxi Scenario

In this scenario we read in two log files ’tripData’ and ’tripFare’, in the source operator. The log files were downloaded from [Who14]. Both of these files contain each one half of the information on taxi rides in NYC in August 13. The structure of the log files can be seen in Figure 4.6.

The structure of the Taxi Scenario operator graph Ω can be seen in Figure 4.4. Each operator processes incoming events differently. As such each operator produces different output events. In the following, we address the operators as ω, with the id they have in Figure 4.4.

ω_1: The source of Ω. It reads in two separate files containing each one half of information on taxi rides in NYC in August 13. ω_1 sends the information from the 'tripData' file (Figure 4.6) to ω_2. It then sends information of the 'tripFare' (Figure 4.6) file to ω_3.

ω_2, ω_3: These operators are merge operators. They are located on different branches of Ω. They have both a window size of 150. Both these operators do the same for the events that pass through them. These operators merges all events that contain information about taxi rides on the same day are merged into one event. After the window is complete, the operator sends the merged to the next operator. ω_2 sends its output to ω_4. ω_3 sends its output to ω_5.

ω_4: ω_4 is a forwarding operator. It has a window size of 150. All input events that it receives contain information about taxi rides on the same day. Those events are the result of the merging of events in ω_2. But the input events still contain the individual values of the events that they were created from. ω_4 gets events from ω_2 and computes the starting districts of the taxi ride from the GPS data. We use the coordinate $(40.7342645, -73.928357)$ and sort the taxi rides into rides that started northwest, northeast, southwest or southeast of the coordinate. ω_4 then sends the events to ω_6.

ω_5: ω_5 is a forwarding operator. It has a window size of 150. All input events that it receives contain information about taxi rides on the same day. Those events are the result of the merging of events in ω_3. But the input events still contain the individual values of the events that they were created from. From the price of the individual taxi rides, ω_5 computes the average price per ride. It then sends the events to ω_7.

ω_6, ω_7: These operators are merge operators. They are located on different branches of Ω. They have both a window size of 150. Both these operators do the same for the events that pass through them. These operators merge all events that contain information about taxi rides on the same day are merged into one event. For each merge, the operator computes the average values for the attributes. After the window is complete, the operator sends the merged event to the next operator. ω_6 sends its output to ω_8. ω_7 sends its output to ω_9.

ω_8, ω_9: These operators are merge operators. They are located on different branches of Ω. They have both a window size of 150. Both these operators do the same for the events that pass through them. These operators merge all events that contain information about taxi rides on the same day are merged into one event. For each merge, the operator computes the average values for the attributes. After the window is complete, the operator sends the merged event to the next operator. Both of these operators send their output to ω_{10}.

ω_{10}: This is a merge operator. After ω_1, which reads in data from separate files containing each half the information of the taxi rides on one day, the data is kept separately. ω_1 splits the data between the two branches of Ω. ω_{10} merges the two data streams coming from ω_8 and ω_9. As such ω_{10} gets one event with half the information of one day from ω_8. It gets the other half from ω_9. These events are then merged into only one event. This event then contains all data of that day. ω_{10} then sends the merged event to ω_{11}.

ω_{11}: ω_{11} receives events from ω_{10}. These events represent the cumulative information on taxi rides in NYC on a single day. Each event has five attribute values that are of interest here: The number of taxi rides in four districts and the average price for a ride on this day. In a window, we have five events, each event represents one days. We search for the following: In at least one of the five attributes we want to have an increase two times in a row. For that purpose, we consider each attribute separately from the others. Because this ω_{11} the sink of Ω, events are not sent to another operator.

4.6 Experiment Setup

In the experiment we first determine the baseline for our scenarios. We do this by running each scenario several times and saving the average number of output events after each operator, and the total run time. We determine the stream quality and accuracy the end result by how many percent of the output events are still sent, when load shedding is used. When using load shedding, we always run the scenarios after the same principle. We only use load shedding at one operator at a time. For random load shedding we start with a drop probability of 5% and increase it every run by 5%. We do this up until 95% drop probability. Then we repeat this for every operator in the operator graph. This whole process is one complete run.

For the Fire Scenario we generate 3,000,000 events per run, 1,000,000 of each types. In the experiment we do 50 runs for random load shedding and another 50 runs for utility-based load shedding.

In the Twitter Scenario we read in 751545 Tweets in each run. In the experiment we do 15 runs for random load shedding and another 15 runs for utility-based load shedding. We do not need to do as many runs as in the Fire Scenario, because we do not generate random data for the events.

In the Taxi Scenario we read in 1,048,675 taxi rides in each run. In the experiment we do 15 runs for random load shedding. For for utility-based load shedding we do only one run, because we use fixed utility values to drop the events in a fixed order. We do not need to do as many runs as in the Fire Scenario, because we do not generate random data for the events.

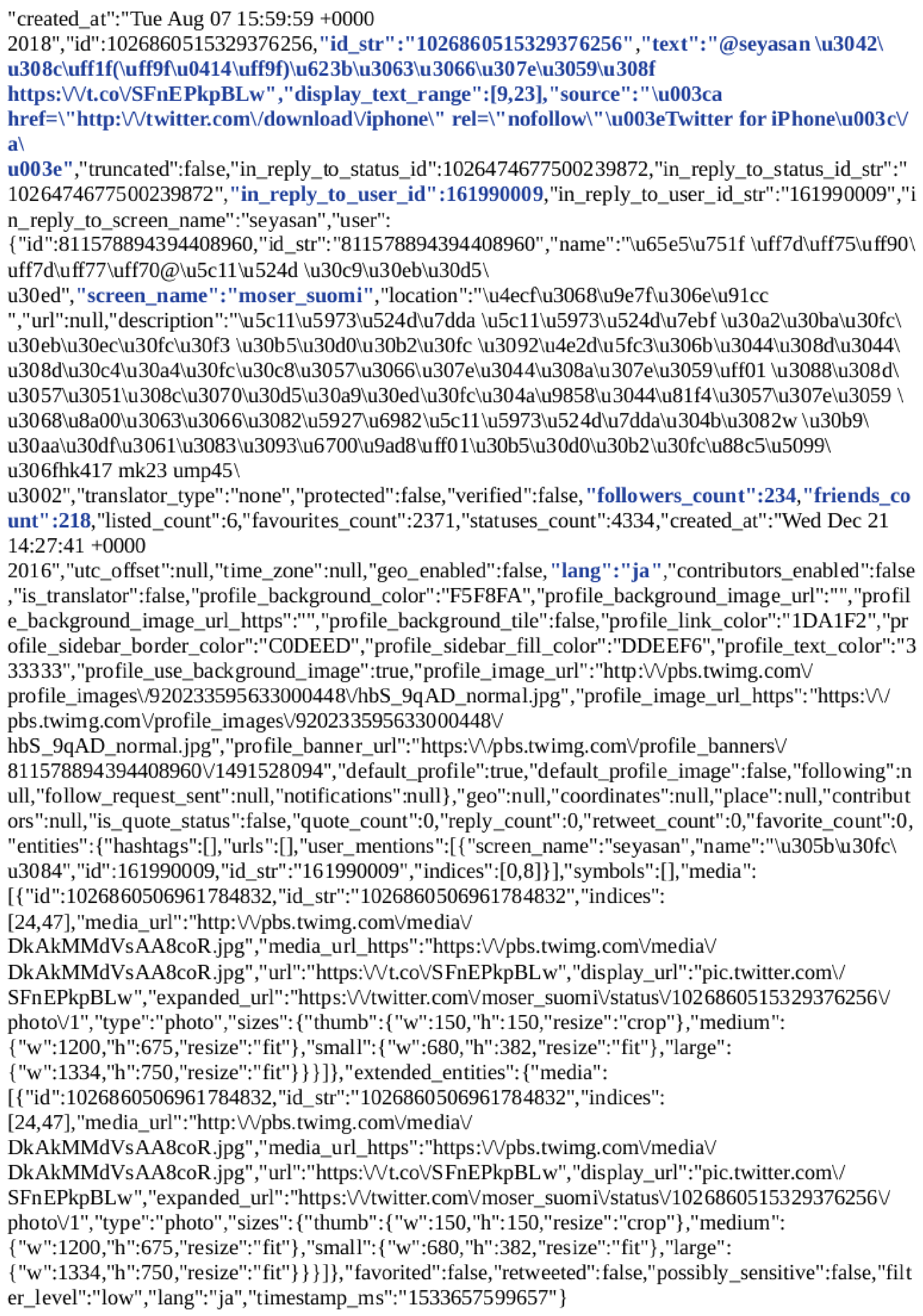
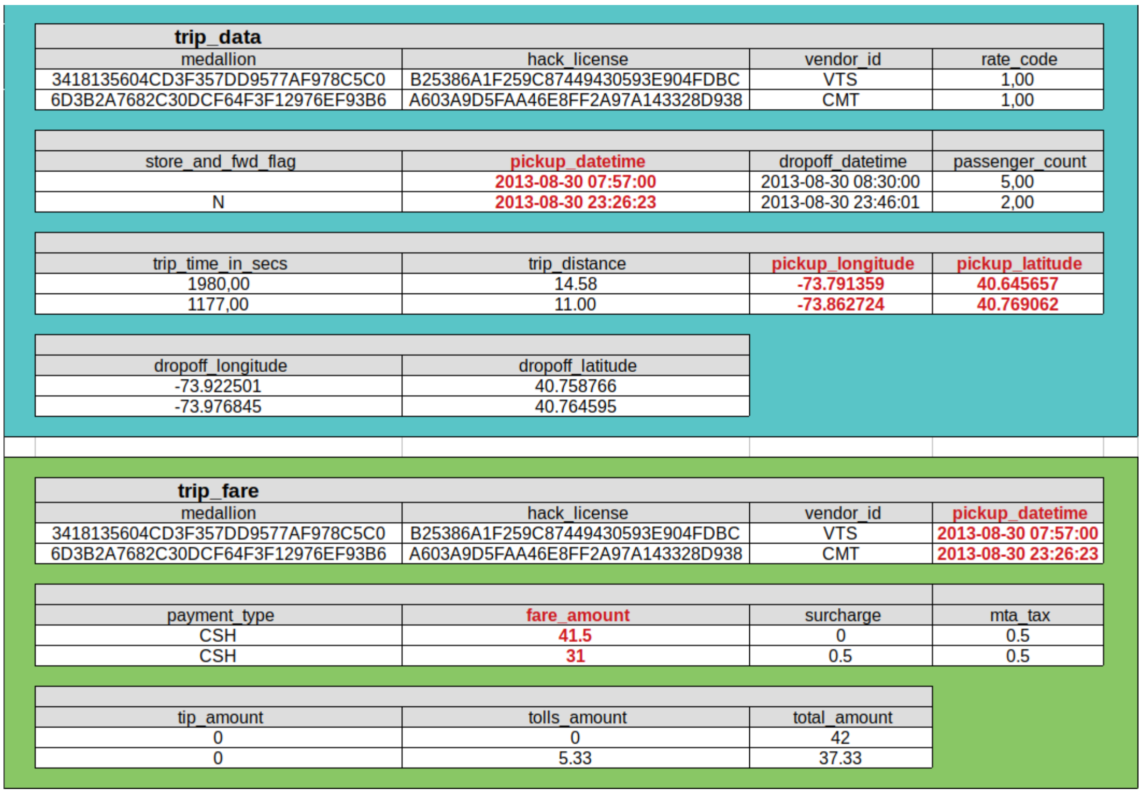

"created_at":"Tue Aug 07 15:59:59 +0000
2018","id":1026860515329376256,**"id_str":"1026860515329376256","text":"@seyasan \u3042\
u308c\uff1f(\uff9f\u0414\uff9f)\u623b\u3063\u3066\u307e\u3059\u308f
https:\/\/t.co\/SFnEPkpBLw","display_text_range":[9,23],"source":"\u003ca
href=\"http:\/\/twitter.com\/download\/iphone\" rel=\"nofollow\"\u003eTwitter for iPhone\u003c\/
a\
u003e"**,"truncated":false,"in_reply_to_status_id":1026474677500239872,"in_reply_to_status_id_str":"
1026474677500239872",**"in_reply_to_user_id":161990009**,"in_reply_to_user_id_str":"161990009","i
n_reply_to_screen_name":"seyasan","user":
{"id":811578894394408960,"id_str":"811578894394408960","name":"\u65e5\u751f \uff7d\uff75\uff90\
uff7d\uff77\uff70@\u5c11\u524d \u30c9\u30eb\u30d5\
u30ed",**"screen_name":"moser_suomi"**,"location":"\u4ecf\u3068\u9e7f\u306e\u91cc
","url":null,"description":"\u5c11\u5973\u524d\u7dda \u5c11\u5973\u524d\u7ebf \u30a2\u30ba\u30fc\
u30eb\u30ec\u30fc\u30f3 \u30b5\u30d0\u30b2\u30fc \u3092\u4e2d\u5fc3\u306b\u3044\u308d\u3044\
u308d\u30c4\u30a4\u30fc\u30c8\u3057\u3066\u307e\u3044\u308a\u307e\u3059\uff01 \u3088\u308d\
u3057\u3051\u308c\u3070\u30d5\u30a9\u30ed\u30fc\u304a\u9858\u3044\u81f4\u3057\u307e\u3059 \
u3068\u8a00\u3063\u3066\u3082\u5927\u6982\u5c11\u5973\u524d\u7dda\u304b\u3082w \u30b9\
u30aa\u30df\u3061\u3083\u3093\u6700\u9ad8\uff01\u30b5\u30d0\u30b2\u30fc\u88c5\u5099\
u306fhk417 mk23 ump45\
u3002","translator_type":"none","protected":false,"verified":false,**"followers_count":234,"friends_co
unt":218**,"listed_count":6,"favourites_count":2371,"statuses_count":4334,"created_at":"Wed Dec 21
14:27:41 +0000
2016","utc_offset":null,"time_zone":null,"geo_enabled":false,**"lang":"ja"**,"contributors_enabled":false
,"is_translator":false,"profile_background_color":"F5F8FA","profile_background_image_url":"","profil
e_background_image_url_https":"","profile_background_tile":false,"profile_link_color":"1DA1F2","pr
ofile_sidebar_border_color":"C0DEED","profile_sidebar_fill_color":"DDEEF6","profile_text_color":"3
33333","profile_use_background_image":true,"profile_image_url":"http:\/\/pbs.twimg.com\/
profile_images\/920233595633000448\/hbS_9qAD_normal.jpg","profile_image_url_https":"https:\/\/
pbs.twimg.com\/profile_images\/920233595633000448\/
hbS_9qAD_normal.jpg","profile_banner_url":"https:\/\/pbs.twimg.com\/profile_banners\/
811578894394408960\/1491528094","default_profile":true,"default_profile_image":false,"following":n
ull,"follow_request_sent":null,"notifications":null},"geo":null,"coordinates":null,"place":null,"contribut
ors":null,"is_quote_status":false,"quote_count":0,"reply_count":0,"retweet_count":0,"favorite_count":0,
"entities":{"hashtags":[],"urls":[],"user_mentions":[{"screen_name":"seyasan","name":"\u305b\u30fc\
u3084","id":161990009,"id_str":"161990009","indices":[0,8]}],"symbols":[],"media":
[{"id":1026860506961784832,"id_str":"1026860506961784832","indices":
[24,47],"media_url":"http:\/\/pbs.twimg.com\/media\/
DkAkMMdVsAA8coR.jpg","media_url_https":"https:\/\/pbs.twimg.com\/media\/
DkAkMMdVsAA8coR.jpg","url":"https:\/\/t.co\/SFnEPkpBLw","display_url":"pic.twitter.com\/
SFnEPkpBLw","expanded_url":"https:\/\/twitter.com\/moser_suomi\/status\/1026860515329376256\/
photo\/1","type":"photo","sizes":{"thumb":{"w":150,"h":150,"resize":"crop"},"medium":
{"w":1200,"h":675,"resize":"fit"},"small":{"w":680,"h":382,"resize":"fit"},"large":
{"w":1334,"h":750,"resize":"fit"}}}]},"extended_entities":{"media":
[{"id":1026860506961784832,"id_str":"1026860506961784832","indices":
[24,47],"media_url":"http:\/\/pbs.twimg.com\/media\/
DkAkMMdVsAA8coR.jpg","media_url_https":"https:\/\/pbs.twimg.com\/media\/
DkAkMMdVsAA8coR.jpg","url":"https:\/\/t.co\/SFnEPkpBLw","display_url":"pic.twitter.com\/
SFnEPkpBLw","expanded_url":"https:\/\/twitter.com\/moser_suomi\/status\/1026860515329376256\/
photo\/1","type":"photo","sizes":{"thumb":{"w":150,"h":150,"resize":"crop"},"medium":
{"w":1200,"h":675,"resize":"fit"},"small":{"w":680,"h":382,"resize":"fit"},"large":
{"w":1334,"h":750,"resize":"fit"}}}]},"favorited":false,"retweeted":false,"possibly_sensitive":false,"filt
er_level":"low","lang":"ja","timestamp_ms":"1533657599657"}

Figure 4.5: The structure of a Twitter log file. Here we can see the log of a single Tweet. In blue we can see the attributes that we need for the Twitter Scenario. Reading in these log files is very time consuming, as a direct result of the structure and the amount of information they contain.



trip_data

medallion	hack_license	vendor_id	rate_code
3418135604CD3F357DD9577AF978C5C0	B25386A1F259C87449430593E904FDBC	VTS	1,00
6D3B2A7682C30DCF64F3F12976EF93B6	A603A9D5FAA46E8FF2A97A143328D938	CMT	1,00

store_and_fwd_flag	pickup_datetime	dropoff_datetime	passenger_count
	2013-08-30 07:57:00	2013-08-30 08:30:00	5,00
N	2013-08-30 23:26:23	2013-08-30 23:46:01	2,00

trip_time_in_secs	trip_distance	pickup_longitude	pickup_latitude
1980,00	14.58	-73.791359	40.645657
1177,00	11.00	-73.862724	40.769062

dropoff_longitude	dropoff_latitude
-73.922501	40.758766
-73.976845	40.764595

trip_fare

medallion	hack_license	vendor_id	pickup_datetime
3418135604CD3F357DD9577AF978C5C0	B25386A1F259C87449430593E904FDBC	VTS	2013-08-30 07:57:00
6D3B2A7682C30DCF64F3F12976EF93B6	A603A9D5FAA46E8FF2A97A143328D938	CMT	2013-08-30 23:26:23

payment_type	fare_amount	surcharge	mta_tax
CSH	41.5	0	0.5
CSH	31	0.5	0.5

tip_amount	tolls_amount	total_amount
0	0	42
0	5.33	37.33

Figure 4.6: The taxi data files. As an example we see two taxi rides. Outlined in blue we see the file 'trip_data'. Outlined in green we see the file 'trip_fare'. The attributes that are marked in red are used in the Taxi Scenario. We can see that the first taxi ride in 'trip_fare' is the continuation of the taxi ride in 'trip_data'. This principle is true for all taxi rides.

5 Evaluation

5.1 Hypotheses

In the following, we present hypotheses on how we expect Random Load Shedding (RLS) and Utility-based Load Shedding (ULS) to affect the different operator types.

5.1.1 Filter/Forwarding Operator

The percentage of events not detected should be equal to the drop probability of the RLS. We do expect that this operator type cannot improve the stream quality when we use RLS.

With a well-working utility function for this operator the percentage of events not detected should be significantly better than the overall drop rate of the ULS. Therefore we expect, that operator type can improve the stream quality when we use ULS.

5.1.2 Merge Operator

The worst-case the result should look like the filter operator. The reason is, that in this case, the operator does not merge events. The events are sent on in the state they had as an input event.

The higher the number of merged together events in a window is, the better the result should be for RLS. The reason is that the chance for a single dropped events to reduce the number of outputs would continuously sink. The higher the number of events that the operator merges into one is, the better the result.

If we have a high number of events that the operator merges into one, we also expect this operator type to improve the stream quality. The unrestricted merge operator type should have a clear advantage in that regard.

ULS should not be a great improvement over RLS in this case. It can potentially improve the attributes of the merged events by controlling which types of events are left to be merged. For the number of output events, ULS should only give a small advantage over RLS, if at all. Only if we can drop events, that are merged together with a high number of other events, with a higher drop probability, should we see an improvement in the result, compared to RLS.

5.1.3 Propositional Logic Operator

This operator type is e.g. of the form $A \wedge B \rightarrow C$. In the case that A and B are relative evenly in numbers it should result in a similar situation as in the case of the filter operator. If either A or B are present in greater number than the other, then the less numerous one is the limiting factor. Because it is also dropped with the same drop chance it should also lead to a result that misses a percentage of events equal to the drop chance of the RLS.

In the case that A and B are relative evenly in numbers, ULS should have similar results as RLS. If A or B are present in a greater number than the other (e.g. more A than B), then the less numerous one is still the limiting factor. But if the ULS preferably sheds events of type A, overall not many, events should be missed, if any at all. This is a significant improvement compared to RLS, where in this case the percentage of events missed is equal to the drop chance.

5.1.4 Average-building Operator

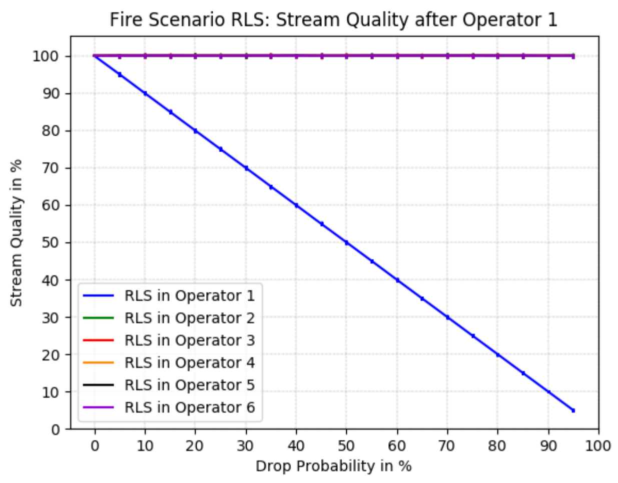
If the values of the events are relatively similar, RLS should have only a very small amount of influence on the result, if any at all. Therefore it is more interesting to look at a case where this is not true. E.g. 100 events total, five events have a very high value and 95 events a very small one. The RLS now drops events with a chance of a 25% drop chance. If now, for example, the operator drops two of the high-value events, that could lead to a big impact in the resulting average, making it much smaller than it should be. ULS should lead to similar results as RLS.

We expect, that this operator type can improve the stream quality greatly. As long as a single input event is in a window, we generate one output event. This is the same number of output events that is generated for a full window. What is more interesting in this case is the stability of the average value in this output event. We expect there to be a difference to the baseline average value.

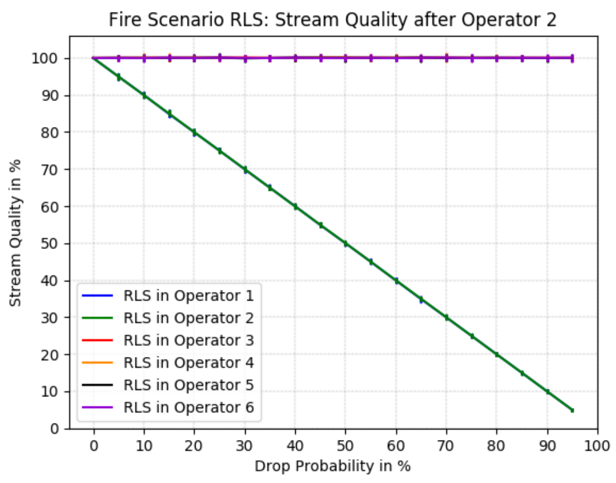
5.1.5 Pattern-finding Operator

RLS should have a great impact on the accuracy of the result. Dropping a single event can destroy a pattern, or lead to the creation of a new one. Overall this is the operator type we expect to have the highest loss in accuracy through RLS. Therefore we thing that this operator type cannot improve the stream quality.

ULS should lead to better results if we have the knowledge to preserve as many patterns as possible when dropping events. This is still the operator type we expect to have the highest loss in accuracy through both ULS and RLS.

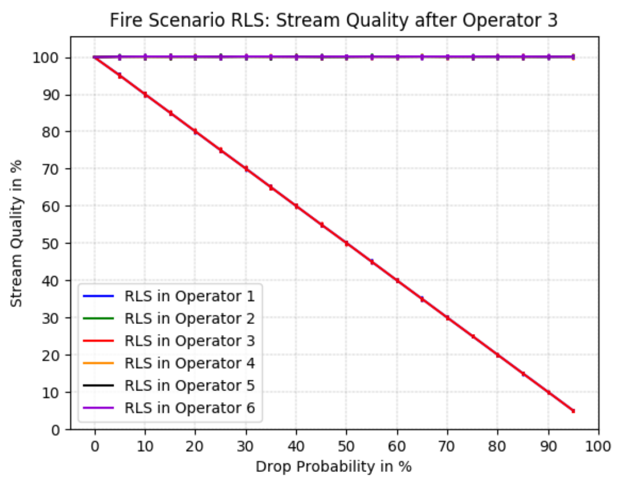


(a)

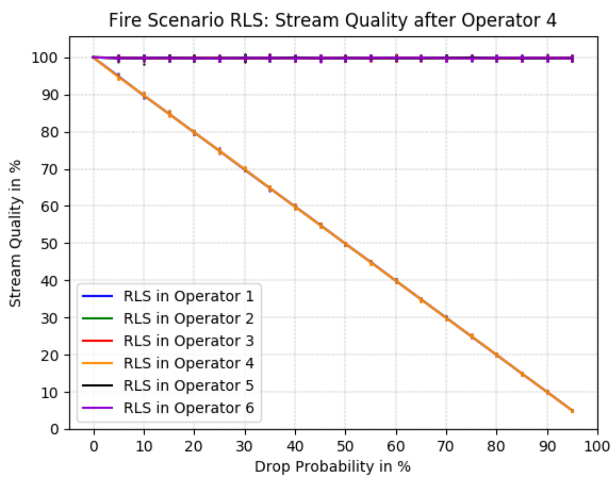


(b)

Figure 5.1: Fire Scenario: Figure 5.1a depicts the stream quality after operator 1. Figure 5.1b depicts the stream quality after operator 2.

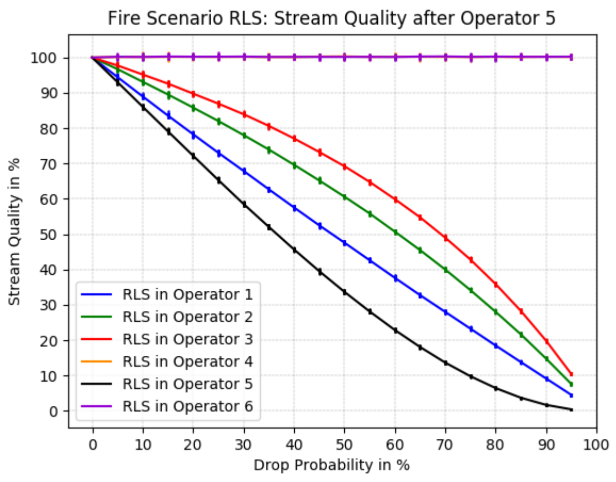


(a)

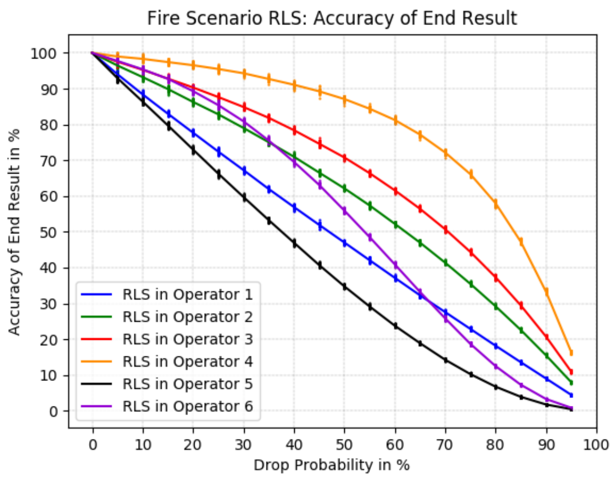


(b)

Figure 5.2: Fire Scenario: Figure 5.2a depicts the stream quality after operator 3. Figure 5.2b depicts the stream quality after operator 4.



(a)



(b)

Figure 5.3: Fire Scenario: Figure 5.3a depicts the stream quality after operator 5. Figure 5.3b depicts the accuracy of the end result.

5.2 Fire Scenario: Random Load Shedding

In the Fire Scenario we address each operator as ω, with the id it has in Figure 4.2. The operator graph is also denoted as Ω. Above we described the expected influence of RLS on the different types of operators. In the Fire Scenario, we have four filter operators (ω_1-ω_4) and two propositional logic operators ($\omega_5 - \omega_6$).

In the following, we examine the results of the experiment. Afterwards, we discuss the accuracy of the hypotheses. We examine the stream quality after each operator separately.

For each operator we examine the result of load shedding at all preceding operators and the operator itself. For every operator, where load shedding happens, we give a brief overview of the inner workings of the operator. Afterwards, we explain the impact on the stream quality caused by load shedding at this operator. Lastly, we compare the hypothesis for this event type with the results.

5.2.1 Fire Scenario RLS: Stream Quality after Operator 1

The development of stream quality after operator 1 can be seen in Figure 5.1a. It shows the stream quality for Load Shedding (LS) at one operator at a time for every tested drop probability. In the following, we explain the results seen in Figure 5.1a.

ω_1: The source of Ω, it generates all events with random values. The decline in stream quality after ω_1 is linear ($m = -1, b = 100$) to the increase in drop probability. As such the line in Figure 5.1a for ω_1 has a slope of -1. This is a forwarding operator, the influence of RLS on the stream quality after ω_1 is thus exactly as expected.

ω_2-ω_6: These operators are located after ω_1 in Ω (see Figure 4.2). Thus load shedding at these operators can not retroactively influence the stream quality after ω_1. The stream quality after ω_1 remains therefore at 100 % if we use LS at these operators.

5.2.2 Fire Scenario RLS: Stream Quality after Operator 2

The development of stream quality after operator 2 can be seen in Figure 5.1b. It shows the stream quality for LS at one operator at a time for every tested drop probability. In the following, we explain the results seen in Figure 5.1b.

ω_1: The source of Ω, it generates all events with random values. It sends events to ω_2, which are the only input events that ω_2 receives. The decline of the stream quality after ω_2 is linear ($m = -1, b = 100$) to the increase in drop probability in ω_1.

This is a filter operator, the influence of RLS on the stream quality after ω_2 is thus exactly as expected.

ω_2: ω_2 forwards its input events to ω_5. Therefore RLS on this operator results in a decline of the stream quality after ω_2 that is linear ($m = -1, b = 100$) to the increase in drop probability.

This is a filter operator, the influence of RLS on the stream quality after ω_2 is thus exactly as expected.

ω_3-ω_6: These operators are located after ω_2 in Ω (see Figure 4.2). Thus load shedding at these operators can not retroactively influence the stream quality after ω_2. The stream quality after ω_2 remains therefore at 100 % if we use LS at these operators.

5.2.3 Fire Scenario RLS: Stream Quality after Operator 3

The development of stream quality after operator 3 can be seen in Figure 5.2a. It shows the stream quality for LS at one operator at a time for every tested drop probability. In the following, we explain the results seen in Figure 5.2a.

ω_1: The source of Ω , it generates all events with random values. It sends events to ω_3, which are the only input events that ω_3 receives. The decline of the stream quality after ω_3 is linear ($m = -1, b = 100$) to the increase in drop probability in ω_1.

This is a filter operator, the influence of RLS on the stream quality after ω_3 is thus exactly as expected.

ω_2: ω_2 is located parallel to ω_3 in Ω (see Figure 4.2). Thus load shedding at ω_2 can not influence the stream quality after ω_3. The stream quality after ω_3 remains therefore at 100 % if we use LS at ω_2.

ω_3: ω_3 forwards its input events to ω_5. Therefore RLS on this operator results in a decline of the stream quality after ω_3 that is linear ($m = -1, b = 100$) to the increase in drop probability.

ω_4-ω_6: These operators are located after ω_3 in Ω (see Figure 4.2). Thus load shedding at these operators can not retroactively influence the stream quality after ω_3. The stream quality after ω_3 remains therefore at 100 % if we use LS at these operators.

5.2.4 Fire Scenario RLS: Stream Quality after Operator 4

The development of stream quality after operator 4 can be seen in Figure 5.2b. It shows the stream quality for LS at one operator at a time for every tested drop probability. In the following, we explain the results seen in Figure 5.2b.

ω_1: The source of Ω, it generates all events with random values. It sends events to ω_4, which are the only input events that ω_4 receives. The decline of the stream quality after ω_4 is linear ($m = -1, b = 100$) to the increase in drop probability in ω_1.

This is a filter operator, the influence of RLS on the stream quality after ω_4 is thus exactly as expected.

ω_2-ω3: These operators are located parallel to ω_4 in Ω (see Figure 4.2). Thus load shedding at these operators can not influence the stream quality after ω_4. The stream quality after ω_4 remains therefore at 100 % if we use LS at these operators.

ω_4: ω_4 forwards its input events to ω_6. Therefore RLS on this operator results in a decline of the stream quality after ω_4 that is linear ($m = -1, b = 100$) to the increase in drop probability.

This is a filter operator, the influence of RLS on the stream quality after ω_4 is thus exactly as expected.

ω_5-ω_6: These operators are located after ω_4 in Ω (see Figure 4.2). Thus load shedding at these operators can not retroactively influence the stream quality after ω_4. The stream quality after ω_4 remains therefore at 100 % if we use LS at these operators.

5.2.5 Fire Scenario RLS: Stream Quality after Operator 5

The development of stream quality after operator 5 can be seen in Figure 5.3a. It shows the stream quality for LS at one operator at a time for every tested drop probability. In the following, we explain the results seen in Figure 5.3a.

ω_1: The source of Ω, it generates all events with random values. ω_1 generates events of type A, B and C in equal numbers. RLS drops events of all types in the same proportion. All input events at ω_5 are of type A and B. Thus the decline of the stream quality after ω_5 is linear ($m = -1, b = 100$) to the increase in drop probability in ω_1.

This is a filter operator, the influence of RLS on the stream quality after ω_5 is thus exactly as expected.

ω_2: ω_2 forwards events of type A to ω_5. The decline in stream quality when using LS in ω_2 is linear ($b = 100$), but not as steep as seen when using LS in ω_1. This was expected because only events of type A pass through this operator. ω_5 only needs one event of type A above the threshold per window for a match. Because events of type B do not pass through ω_2 they are not dropped. A match of ω_5 is of the form $A \wedge B \rightarrow D$, with A and B greater threshold. Thus the overall probability to find a match at ω_5 is higher when using load shedding at ω_2, compared to using RLS in ω_1.

ω_2 is of type filter operator and the result matches the expectations.

ω_3: ω_3 forwards events of type B to ω_5. The decline in stream quality when using LS in ω_3 is linear ($b = 100$), but not as steep as seen when using LS in ω_1 or ω_2. This was expected because only events of type B pass through this operator. The threshold for a B to contribute to a match in ω_5 is also lower than for on an A. Events of type B are thus on average more likely to contribute to a match in ω_5 than an event of type A. We can thus shed comparatively more 'B's than 'A's to receive the same stream quality after ω_5. The probability of detecting a match in ω_5 is higher when using RLS at ω_3, compared to using RLS at ω_2 or ω_1.

ω_3 is of type filter operator and the result matches the expectations.

ω_4: ω_4 is on a different branch in Ω than ω_5 (see Figure 4.2). Thus load shedding at this operator can not influence the stream quality after ω_5. The stream quality after ω_5 remains therefore at 100% if we use LS at ω_4.

ω_5: ω_5 receives input events from ω_2 and ω_3. From ω_2, it receives events of type A and from ω_3, it receives events of type B. Overall the number of 'A's and 'B's is equal, but that proportion can be different for the individual windows of size 10. In each window, one event of each type needs to be present, which also passes the threshold for its event type. For events of type A, roughly 10% of events pass this threshold. For events of type B, the passing rate is about 20%.

It is not easy to find a match, even without using load shedding. When using load shedding, the load shedder removes events from a window. This reduces the probability to find a match even further. The loss in stream quality after ω_5 is the greatest if load shedding is done at ω_5 itself (seeFigure 5.3a). The degradation of stream quality is even worse than a linear decline ($m = -1, b = 100$). The reduction of available window spots reduces the stream quality to a great degree. It is worse than shedding input events with the same drop probability.

ω_6: ω_6 is located after ω_5 in Ω (see Figure 4.2). Thus load shedding at this operator can not influence the stream quality after ω_5. The stream quality after ω_5 remains therefore at 100% if we use LS at ω_6.

5.2.6 Fire Scenario RLS: Accuracy of the End Result

The accuracy of the end result is the stream quality after operator 6. The development of the accuracy of end result can be seen in Figure 5.3b. It shows the accuracy of end result for LS at one operator at a time for every tested drop probability. In the following, we explain the results seen in Figure 5.3b.

ω_1: The source of Ω, it generates all events with random values. ω_1 generates events of type A, B and C in equal numbers. RLS drops events of all types in the same proportion. All input events at ω_6 are of type C and D, and type D results as a match of A and B at ω_5. Thus the decline of the stream quality after ω_6 is linear $(m = -1, b = 100)$ to the increase in drop probability in ω_1.

This is a filter operator, the influence of RLS on the stream quality after ω_6 is thus exactly as expected.

ω_2: ω_2 forwards events of type A to ω_5. The decline in stream quality when using LS in ω_2 is linear $(b = 100)$, but not as steep as seen when using LS in ω_1. This was expected because only events of type A pass through this operator. ω_5 only needs one event of type A above the threshold per window for a match. Because events of type B do not pass through ω_2 they are not dropped. A match of ω_5 is of the form $A \wedge B \rightarrow D$, with A and B greater threshold.

ω_2 sends half of the input for ω_5. ω_6 follows after ω_5 and ω_5 sends the limiting factor for ω_6. Thus a higher probability of finding a match in ω_5 results in a higher probability to find a match in ω_6.

ω_2 is of type filter operator and the result matches the expectations.

ω_3: ω_3 forwards events of type B to ω_5. The decline in stream quality when using LS in ω_3 is linear $(b = 100)$, but not as steep as seen when using LS in ω_1 or ω_2. This was expected because only events of type B pass through this operator. The threshold for a B to contribute to a match in ω_5 is also lower than for on an A. Events of type B are thus on average more likely to contribute to a match in ω_5 than an event of type A. We can thus shed comparatively more 'B's than 'A's to receive the same stream quality after ω_5.

ω_3 sends half of the input for ω_5. ω_6 follows after ω_5 and ω_5 sends the limiting factor for ω_6. Thus a higher probability of finding a match in ω_5 results in a higher probability to find a match in ω_6. ω_3 is of type filter operator and the result matches the expectations.

ω_4: ω_4 forwards events of type C to ω_6. The decline in accuracy of the end result when using LS in ω_4 is significantly better than linear $(m = -1, b = 100)$. Because of the topology of Ω, we need a smaller amount of events of type C than of type A and B to detect a match in ω_6. The reason is, that 'A's and 'B's need to be matched two times, one time directly at ω_5, one time indirectly at ω_6. A C only needs to contribute to one match in ω_6 to contribute to the end result. Thus a threshold exceeding C is more likely to reach ω_6, compared to threshold exceeding 'A's and 'B's. Therefore more events of type C can be shed to get an end result of a certain accuracy compared to events of type A and B.

ω_4 is of type filter operator. Thus we expected a linear decrease in accuracy of end result when using load shedding at Ω. Because of the many spare 'C's that exceed the threshold at ω_6 we do not see a linear decrease in this case. The loss in accuracy of end result is significantly better than linear decrease with $m = -1, b = 100$.

ω_5: ω_5 is a propositional logic operator. It sends events of type D to ω_6. Events of type D represent a match of a threshold exceeding A and a threshold exceeding B. As explained above in ω_4, events of type C are more likely to reach ω_6 than events of type D. Thus 'C's reach ω_6 in far greater number. RLS in ω_5 leads to the overall greatest loss in accuracy of end result. The reason is, that it results in fewer events of type D, that are sent to ω_6. The 'D's become the limiting factor in ω_6. ω_6 is flooded with 'C's. But without a D in the same window, a threshold exceeding C can not lead to a match.

ω_5 is a propositional logic operator that sends the limiting factor to ω_6. Thus I expected a linear decrease $(m = -1, b = 100)$ of the accuracy of end result when using RLS at ω_5. As we have seen, the actual loss in accuracy of the end result is far worse than that. The reason for this is, how greatly outnumbered 'D's are at ω_6. The result would look different for a more equal composition of input events at ω_6. The closer in number the event types, the closer the loss in accuracy would get a linear decrease $(m = -1, b = 100)$.

ω_6: ω_6 is a propositional logic operator. The decline in the accuracy of end result through load shedding at ω_6 is better than linear $(m = -1, b = 100)$ to the drop probability. This is true until roughly 67% drop probability, afterwards, it gets continuously worse. Incoming events of type C are vastly outnumbering events of type D. Thus 'D's are more important for a match. With lower drop probabilities unnecessary 'C's get dropped in far greater proportion. This is because many windows only contain 'C's. It gets worse with higher drop probabilities. ω_6 drops 'D's and also threshold exceeding 'C's in such a great number, that matches are becoming very unlikely.

ω_6 is a propositional logic operator. We expected, that using RLS would result in a linear decrease $(m = -1, b = 100)$. We have seen that this is not the case. For lower drop probabilities the result is better, for higher drop probabilities worse. For higher drop probabilities the number of remaining events per window is very low. Finding matches become very unlikely as a result.

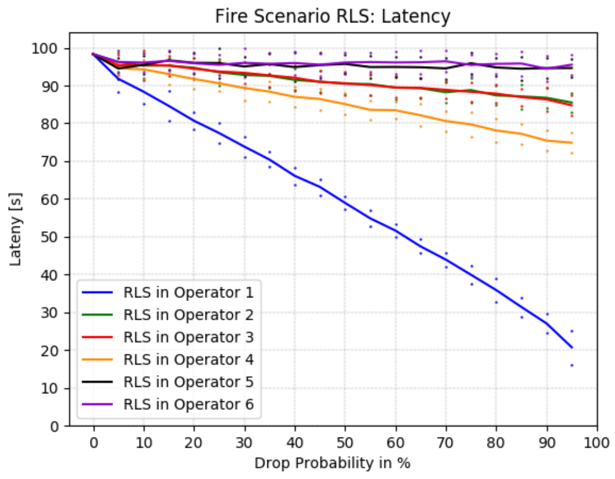


Figure 5.4: The latency in the Fire Scenario when using random load shedding.

5.2.7 Fire Scenario RLS: Latency

In the following, we examine the latency of the experiment while using random load shedding. We examine each operator separately. We address the operator as ω, with the id it has in Figure 4.2. The operator graph is also denoted as Ω. The latency for each operator for every tested drop probability can be seen in Figure 5.4.

ω_1: The source of the Ω, it generates all events. Thus a decline in the events send from ω_1 leads to a nearly linear ($m = -1, b = 100$) decrease in overall latency. It linearly decreases the workload of all other operators in Ω.

ω_2: ω_2 forwards its input events to ω_5 When using RLS here, there is still a high number of events that need to be processed at ω_5 and ω_6. This is because ω_2 controls only half the input for ω_5 and even less for ω_6. The latency of the preceding ω_1, which is responsible for a big part of the overall latency, cannot be influenced here. ω_3 and ω_4 are on different branches of Ω and can also not be influenced. As a result, the decrease in overall latency is far smaller compared to using load shedding at ω_1.

ω_3: ω_3 is parallel to ω_2 in Ω. Both operators do the same kind of work: They send the same number of events to ω_5. The reduction of latency is therefore analogous to ω_2. But we know that RLS at ω_3 leads to a more accurate result than RLS at ω_2. ω_3 is therefore the preferable spot in Ω to shed load, compared to ω_2.

ω_4: ω_4 forwards its input events to ω_6. When using RLS here, it leads to a strong decrease in the latency of ω_6. The reason is, that events send by ω_4 are the majority of input events at ω_6. Using RLS here leads to the second greatest decrease in overall latency.

The latency of the preceding ω_1, which is responsible for a big part of the overall latency, cannot be influenced here. ω_2 and ω_3 are on different branches of the Ω and can also not be influenced.

ω_5: ω_5 is a propositional logic operator. When using load shedding here the number of opened windows in ω_5 remains the same. This is because events are only dropped from the windows, but the windows are still created in the first place. But reducing the overall number of input events reduces the number of windows opened. With increasing drop probability ω_5 is causing less latency by processing events. But at the same time, it is causing comparable latency by waiting longer to determine if a match happens or not.

The latency of the preceding ω_1-ω_3 that are responsible for a big part of the overall latency, cannot be influenced here. ω_4 is on a different branch of the Ω and can also not be influenced.

ω_6: ω_6 is a propositional logic operator. When using load shedding here the number of opened windows in ω_6 remains the same. This is because events are only dropped from the windows, but the windows are still created in the first place. But reducing the overall number of input events reduces the number of windows opened. With increasing drop probability ω_6 is causing less latency by processing events. But at the same time, it is causing comparable latency by waiting longer to determine if a match happens or not.

ω_6 is the sink of Ω, the latency of all other operators cannot be influenced here. This makes ω_6 the worst operator to use RLS on.

Overall RLS can reduce the latency by a high degree, as seen when using RLS in operator 1. It is most effective if used as early as possible in Ω. RLS at ω_4 is the best trade-off between reducing latency and accuracy of end result.

5.2.8 Fire Scenario RLS: General Observations

Overall the predictions in were accurate for filter operators. They react as expected with a linear decrease of stream quality after that particular operator.

For propositional logic operators (PLO) the predictions were too simple. Several unexpected results could be seen. The loss in stream quality after the PLO depends on how the load shedding affects the ratio of the incoming event types.

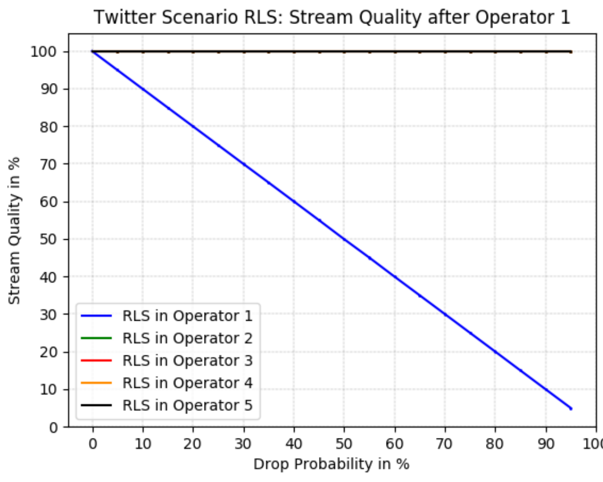
We gained new knowledge through the experiment. Load shedding at a propositional logic operator itself can lead to the greatest loss in stream quality. This is because it removes events from potentially small windows, leaving only a few spots to find a match. This could be different for window sizes that are far bigger than necessary to find a single match. In this case, load shedding at the operator itself could lead to better results. The results could be comparable to dropping the input events on preceding operators in the correct ratio to each other.

5.3 Twitter Scenario: Random Load Shedding

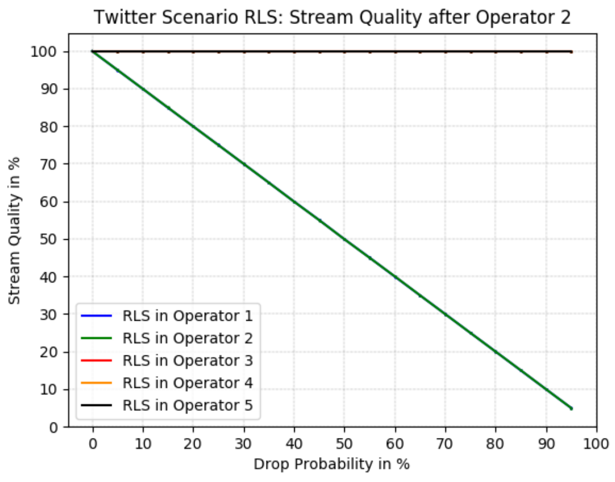
In the Twitter Scenario we address each operator as ω, with the id it has in Figure 4.3. The operator graph is also denoted as Ω. In the Twitter Scenario, we have four filter operators (ω_1 - ω_4) and one average-building operator (ω_5).

In the following, we examine the results of the experiment. Afterwards we discuss the accuracy of the hypotheses.

For each operator we examine the results of load shedding at all preceding operators and the operator itself. For every operator, where load shedding happens, we give a brief overview of the inner workings of the operator. Afterwards, we explain the impact on the stream quality caused by load shedding at this operator. Lastly, we compare the hypothesis for this event type with the results.

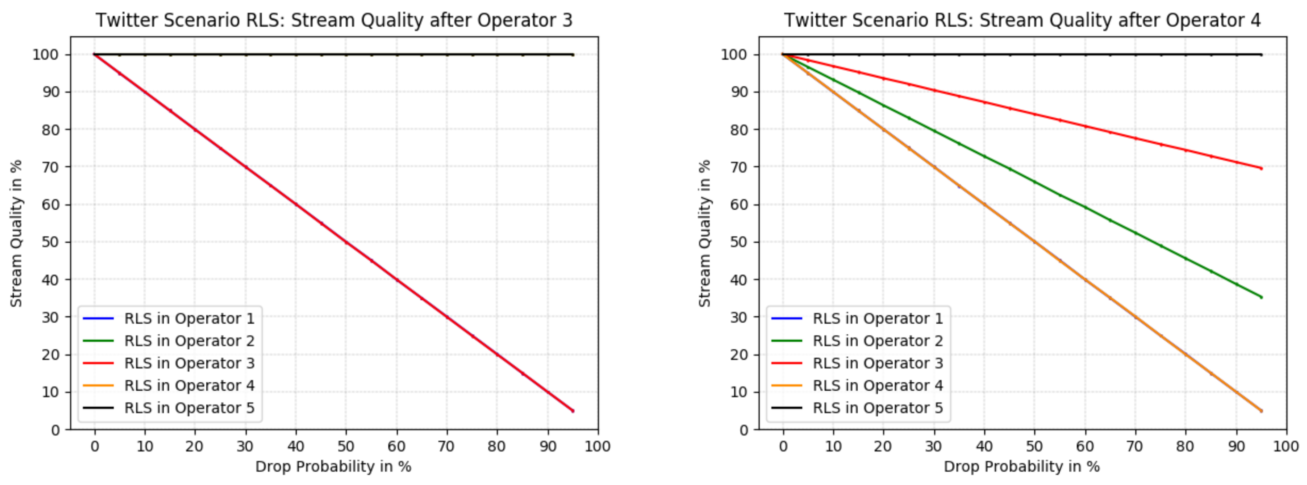


(a)



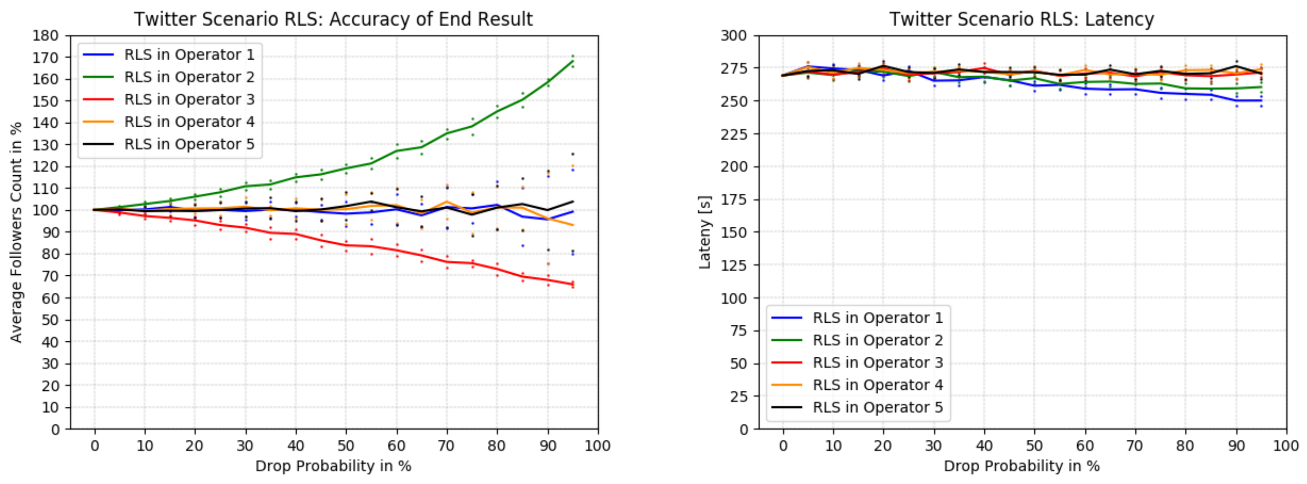
(b)

Figure 5.5: Twitter Scenario:
Figure 5.5a depicts the stream quality after operator 1.
Figure 5.5b depicts the stream quality after operator 2.



(a) **(b)**

Figure 5.6: Twitter Scenario:
Figure 5.6a depicts the stream quality after operator 3.
Figure 5.6b depicts the stream quality after operator 4.



(a) **(b)**

Figure 5.7: Twitter Scenario:
Figure 5.5a depicts the accuracy of the end result.
Figure 5.5b depicts the latency of the experiment.

5.3.1 Twitter Scenario RLS: Stream Quality after Operator 1

The development of stream quality after operator 1 can be seen in Figure 5.5a. It shows the stream quality for LS at one operator at a time for every tested drop probability. In the following, we explain the results seen in Figure 5.5a.

$\boldsymbol{\omega_1}$**:** The source of Ω. ω_1 reads in the Twitter files and generated the events using the read in data. Events that have a Twitter id string divisible by two are sent to ω_2 and ω_3.

The decline in stream quality after ω_1 is linear ($m = -1, b = 100$) to the increase in drop probability.

This is a filter operator, the influence of RLS on the stream quality after ω_1 is thus exactly as expected in.

$\boldsymbol{\omega_2}$**-**$\boldsymbol{\omega_5}$**:** These operators are located after ω_1 in Ω (see Figure 4.3). Thus load shedding at these operators can not retroactively influence the stream quality after ω_1. The stream quality after ω_1 remains therefore at 100% if we use LS at these operators.

5.3.2 Twitter Scenario RLS: Stream Quality after Operator 2

The development of stream quality after operator 2 can be seen in Figure 5.5b. It shows the stream quality for LS at one operator at a time for every tested drop probability. In the following, we explain the results seen in Figure 5.5b.

$\boldsymbol{\omega_1}$**:** The source of Ω. ω_1 reads in the Twitter files and generated the events using the read in data. Events that have a Twitter id string divisible by two are sent to ω_2 and ω_3.

The decline in stream quality after ω_2 is linear ($m = -1, b = 100$) to the increase in drop probability.

This is a filter operator, the influence of RLS on the stream quality after ω_2 is thus exactly as expected.

$\boldsymbol{\omega_2}$**:** This is an operator of the type filter. It receives its input from ω_1. ω_2 sends all Tweets that are written in English to ω_4. The stream quality after ω_2 is linear ($m = -1, b = 100$) to the RLS drop probability at ω_2. Reducing the events that ω_2 needs to process, decreases the output in equal measure.

This is a filter operator, the influence of RLS on the stream quality after ω_2 is thus exactly as expected.

$\boldsymbol{\omega_3}$**-**$\boldsymbol{\omega_5}$**:** These operators are located after ω_2 in Ω (see Figure 4.3). Thus load shedding at these operators can not retroactively influence the stream quality after ω_2. The stream quality after ω_2 remains therefore at 100% if we use LS at these operators.

5.3.3 Twitter Scenario RLS: Stream Quality after Operator 3

The development of stream quality after operator 3 can be seen in Figure 5.6a. It shows the stream quality for LS at one operator at a time for every tested drop probability. In the following, we explain the results seen in Figure 5.6a.

$\boldsymbol{\omega_1}$**:** The source of Ω. ω_1 reads in the Twitter files and generated the events using the read in data. Events that have a Twitter id string divisible by two are sent to ω_2 and ω_3.

The decline in stream quality after ω_3 is linear ($m = -1, b = 100$) to the increase in drop probability.

This is a filter operator, the influence of RLS on the stream quality after ω_3 is thus exactly as expected in.

$\boldsymbol{\omega_2}$**:** ω_2 is located parallel to ω_3 in Ω (see Figure 4.3). Thus load shedding at ω_2 can not influence the stream quality after ω_3. The stream quality after ω_3 remains therefore at 100 % if we use LS at ω_2.

$\boldsymbol{\omega_3}$**:** This is an operator of the type filter. It receives its input from ω_1. ω_3 sends all Tweets that are not written in English and have a friends count higher than 300 to ω_4. The stream quality after ω_3 is linear ($m = -1, b = 100$) to the RLS drop probability at ω_3. Reducing the events that ω_3 needs to process decreases the output in equal measure.

This is a filter operator, the influence of RLS on the stream quality after ω_3 is thus exactly as expected.

$\boldsymbol{\omega_4}$**-**$\boldsymbol{\omega_5}$**:** These operators are located after ω_3 in Ω (see Figure 4.3). Thus load shedding at these operators can not retroactively influence the stream quality after ω_3. The stream quality after ω_3 remains therefore at 100% if we use LS at these operators.

5.3.4 Twitter Scenario RLS: Stream Quality after Operator 4

The development of stream quality after operator 4 can be seen in Figure 5.6b. It shows the stream quality for LS at one operator at a time for every tested drop probability. In the following, we explain the results seen in Figure 5.6b.

$\boldsymbol{\omega_1}$**:** The source of Ω. ω_1 reads in the Twitter files and generated the events using the read in data. Events that have a Twitter id string divisible by two are sent to ω_2 and ω_3.

The decline in stream quality after ω_4 is linear ($m = -1, b = 100$) to the increase in drop probability.

This is a filter operator, the influence of RLS on the stream quality after ω_4 is thus exactly as expected.

ω_2: This is an operator of the type filter. It receives its input from ω_1. ω_2 sends all Tweets that are written in English to ω_4. The stream quality after ω_4 is linear to the RLS drop probability at ω_2 and better than the linear line ($m = -1, b = 100$), see Figure 5.6b.

The reason why the slope of the line for ω_2 in Figure 5.6b is steeper than the line for ω_3, is because ω_2 sends more events to ω_4. As such RLS at ω_2 leads to far fewer input events at ω_4 than RLS with the same drop probability at ω_3 would.

This is a filter operator, the influence of RLS on the stream quality after ω_4 is thus exactly as expected.

ω_3: This is an operator of the type filter. It receives its input from ω_1. ω_3 sends all Tweets that are not written in English and have a friends count higher than 300 to ω_4. The stream quality after ω_4 is linear to the RLS drop probability at ω_3 and better than the linear line ($m = -1, b = 100$), see Figure 5.6b.

The reason why the slope of the line for ω_3 in Figure 5.6b is not as steep as the line for ω_2, is because ω_2 sends more events to ω_4. As such RLS at ω_3 leads to a lot more overall input events at ω_4 than RLS with the same drop probability at ω_2 would.

This is a filter operator, the influence of RLS on the stream quality after ω_4 is thus exactly as expected.

ω_4: This is an operator of the type filter. It receives its input from ω_2 and ω_3. ω_4 sends all Tweets that are a reply to another Tweet to ω_5.

The stream quality after ω_4 is linear ($m = -1, b = 100$) to the RLS drop probability at ω_4 (see Figure 5.6b). As a filter operator, this outcome was to be expected. Reducing the events that ω_4 needs to process will decrease the output in equal measure.

This is a filter operator, the influence of RLS on the stream quality after ω_4 is thus exactly as expected.

ω_5: This operators is located after ω_4 in Ω (see Figure 4.3). Thus load shedding at these operators can not retroactively influence the stream quality after ω_4. The stream quality after ω_4 remains therefore at 100% if we use LS at these operators.

5.3.5 Twitter Scenario RLS: Accuracy of the End Result

The accuracy of the end result is the stream quality after operator 5. The development of the accuracy of end result can be seen in Figure 5.7a. It shows the accuracy of end result for LS at one operator at a time for every tested drop probability. In the following, we explain the results seen in Figure 5.7a.

ω_1: The source of Ω. ω_1 reads in the Twitter files and generates the events using the read in data. Events that have a Twitter id string divisible by two are sent to ω_2 and ω_3.

The end result is on average not impacted by RLS at ω_1 at all, the average follower count remains roughly the same. The accuracy of end result can deviate by roughly +/- 20% for the individual runs at very high drop probabilities.

ω_2: This is an operator of the type filter. It receives its input from ω_1 and sends all Tweets that are written in English to ω_4.

RLS at ω_2 leads to an increase in the average followers count. This increase grows stronger, the higher the drop probability in ω_2 is. This suggests that Twitter users that Tweet in English, have on average a significantly lower follower count compared to users using other languages.

ω_3: This is an operator of the type filter. It receives its input from ω_1. ω_3 sends all Tweets that are not written in English and have a friend's count higher than 300 to ω_4.

RLS at ω_3 leads to a decrease in the average followers count. This decrease grows stronger, the higher the drop probability in ω_3 is. This suggests that Twitter users that Tweet in other languages have on average a significantly higher follower count than English speaking users. This effect is increased by the required friend count of at least 300. It stands to reason that a Twitter user with a lot of friends also has many followers.

The Tweets that ω_3 sends as output are low in number compared to the number of Tweets sent by ω_2. To have an impact on the accuracy of the end result as seen in Figure 5.7a, Tweets that ω_3 are sent as output need to have much higher followers count than the average Tweet.

ω_4: This is an operator of the type filter. It receives its input from ω_2 and ω_3. ω_4 sends all Tweets that are a reply to another Tweet to ω_5.

The end result is on average not impacted by RLS at ω_4 at all, the average followers count remains roughly the same. The accuracy of the end result can deviate by roughly +/- 20% for the individual runs at very high drop probabilities.

ω_5: The sink of Ω. This operator builds an average of the follower count over all events it receives. It receives its input from ω_4.

The end result is on average not affected by RLS at ω_5 at all, the average follower count remains roughly the same. The accuracy of end result can deviate by roughly +/- 20% for the individual runs at very high drop probabilities.

This is an operator type that is very stable when affected by RLS.

5.3.6 Twitter Scenario RLS: Latency

In the following, we examine the latency of the experiment while using random load shedding. We examine each operator separately. We address the operator as ω, with the id it has in Figure 4.3. The operator graph is also denoted as Ω. The latency for each operator for every tested drop probability can be seen in Figure 5.7b.

In this scenario the potential to reduce latency through load shedding is limited. The reading in of the data is responsible for a big part of the latency. Hence the actual processing of the events is responsible for only a small part of the overall latency. Because load shedding can only affect the number of events that are processed, it can only decrease this small chunk of latency.

$\boldsymbol{\omega_1}$: The source of Ω. It generates all events here from read in data. Therefore a decline in the events sent from ω_1 would be assumed to decrease the overall latency linearly ($m = -1, b = 100$) with increasing drop probability. This is not the case here, because ω_1 reads the events in from data files. The reading in of data is responsible for the biggest part of the runtime. Therefore even high RLS drop probabilities yield only about 20% latency reduction for RLS in ω_1.

Still as expected, RLS here at the source of Ω leads to the highest reduction in overall latency.

$\boldsymbol{\omega_2}$: After ω_1 the Ω branches out, with ω_2 and ω_3 both following after ω_1, as can be seen in Figure 4.3.

RLS at ω_2 leads to the second-highest reduction in latency. This is the case because ω_2 sends, while using the given data set, more than three times as many events to ω_4 than ω_3 does. Therefore RLS in ω_2 has a higher impact on the overall number of events that have to be processed by the following operators.

$\boldsymbol{\omega_3}$: After ω_1 the Ω branches out, with ω_2 and ω_3 both following after ω_1.

RLS at ω_3 leads a smaller reduction in latency compared to RLS in ω_2. This is the case because ω_3 sends, while using the given data set, three times less events to ω_4 than ω_2. Therefore RLS in ω_3 has a smaller impact on the overall number of events that have to be processed by the following operators than RLS at ω_2 would have. Hence the slope of the line is not as steep compared to when RLS is used in ω_2.

$\boldsymbol{\omega_4}$: The two divergent branches of ω_2 and ω_3 merge back in ω_4. Because it is the second to last operator in Ω the potential to reduce latency through load shedding is very low here. Events are already processed multiple times before they arrive here.

RLS in ω_4 leads to a similar reduction in latency than RLS in ω_3. This is the case because ω_4 receives the majority of its input from ω_2, and only a smaller part from ω_3. Thus ω_4 processes overall more events than ω_3 does, but later in Ω. This results in roughly the same potential to reduce latency through RLS at these two operators.

$\boldsymbol{\omega_5}$: The sink of Ω. Here an average over all events that reach ω_5 is built. The operation is not very time consuming for the average event. The backlog of events to be processed is very small at this operator. Therefore the potential to reduce latency through RLS at ω_6 is not very high because of the operator type, but also the position in Ω. The reduction in latency is small even for high RLS drop probabilities. Overall this is the worst operator in Ω to apply RLS too.

Overall the reduction in latency that is possible through RLS, is compared to e.g. the Fire Scenario less pronounced. The reason is, that the Twitter Scenario uses a lot of the runtime to read in the necessary data to generate events. In the Fire Scenario, events are randomly generated and additionally much leaner. The Fire Scenario uses a greater percentage of the runtime to process events at the operators. This time we can reduce through load shedding. But LS can not affect the time used reading in data files.

RLS in this scenario is most effective if used as early as possible in Ω, as seen when using RLS in ω_1. If two branches are parallel to each other, RLS is more effective on the operator that produces more output events. Load shedding at the later operators in Ω results, as was expected, in only a small reduction in latency. At this point, a big part of the total processing is already done.

5.3.7 Twitter Scenario RLS: General Observations

Overall the predictions were accurate, random load shedding at filter operators leads to a linear decrease of stream quality. This can be seen here at ω_1. If the stream quality is already reduced through RLS before the filter operator, the linear decrease in the input stream quality leads to a linear decrease in the output stream quality.

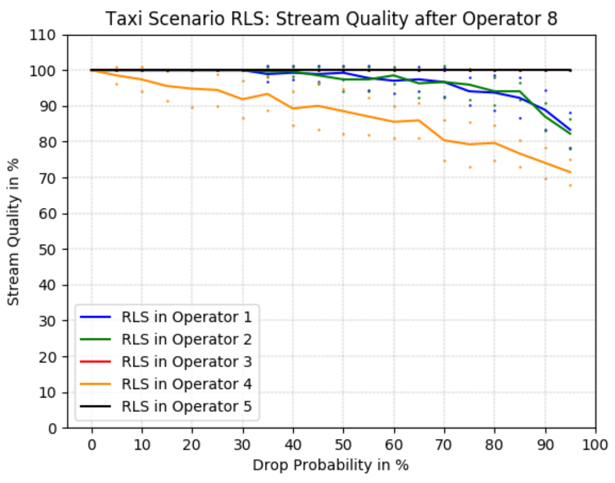
The average-building operator produces a relatively stable result. It is only potentially degraded significantly if, when Ω branches out, RLS is done on only one branch. This is only problematic if the branches produce output events that are significantly different from the average value. In the case that RLS is used in only one of those branches, the accuracy of the end result will degrade strongly. This can be seen in Figure 5.7a, when using RLS in ω_2 or in ω_3.

5.4 Taxi Scenario: Random Load Shedding

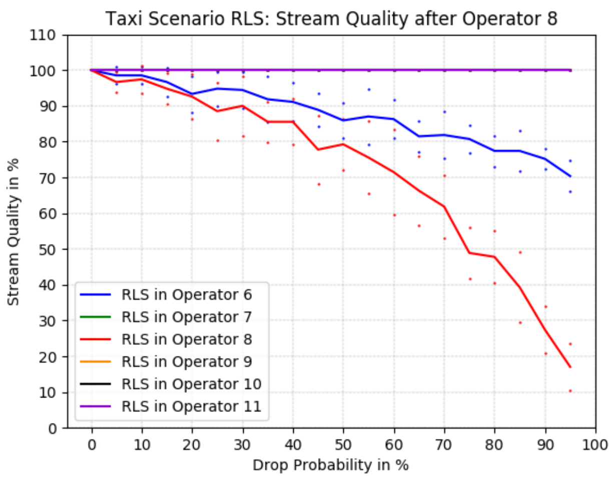
In the Taxi Scenario, we address the operator as ω, with the id it has in Figure 4.4. The operator graph is also denoted as Ω. We have six merging operators (ω_2, ω_3, ω_6, ω_7, ω_8, ω_9) that merge events if possible. Then there are two operators that compute something on the events before sending them on (ω_4, ω_5). The second to last operator requires events to be successfully merged before sending them on (ω_{10}). The last operator (ω_{11}) is a pattern-finding operator.

In the following, we examine the results of the experiment. Afterwards we discuss the accuracy of the hypotheses. We examine the stream quality after each operator separately. We address the operator as ω, with the id it has in Figure 4.4. The operator graph is also denoted as Ω.

For each operator we examine the effects of load shedding at all the preceding operators and the operator itself. For every operator, where load shedding happens, we give a brief overview of the inner workings of the operator. Afterwards, we explain the impact on the stream quality caused by load shedding at this operator. Lastly, we compare the hypothesis for this event type with the results.

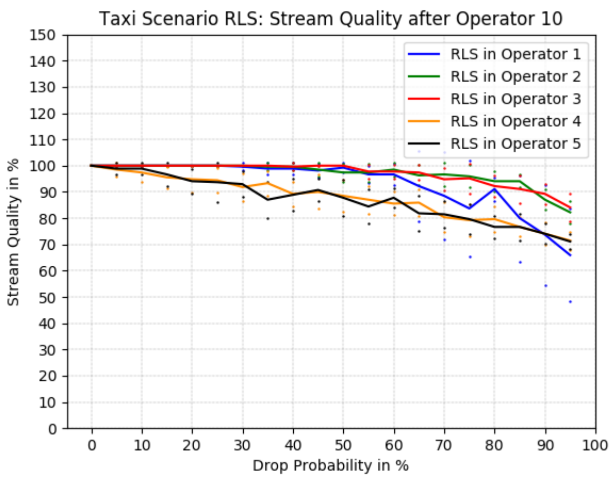


(a)

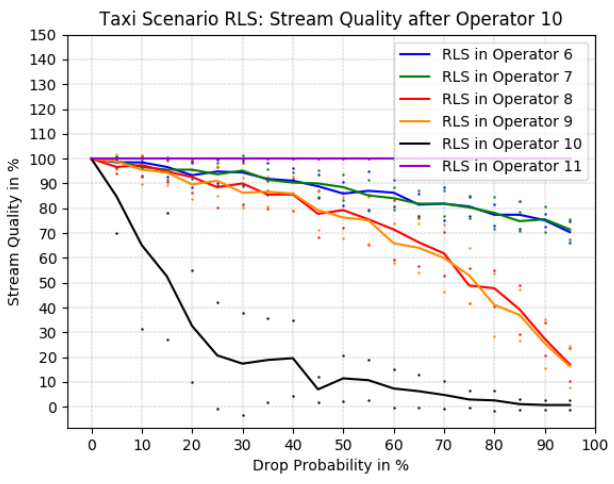


(b)

Figure 5.8: Taxi Scenario: This figures depict the stream quality after operator 8.

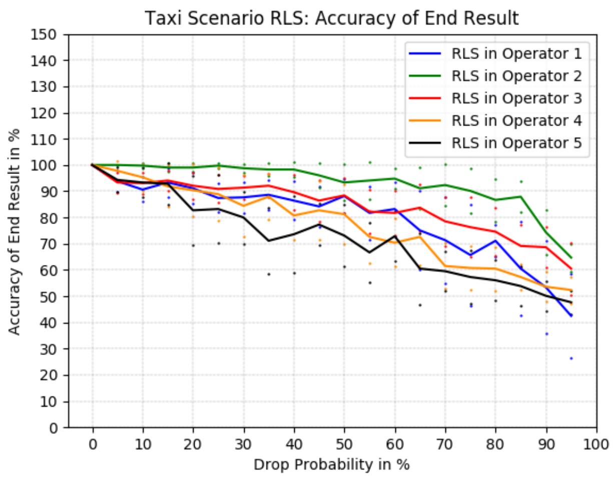


(a)

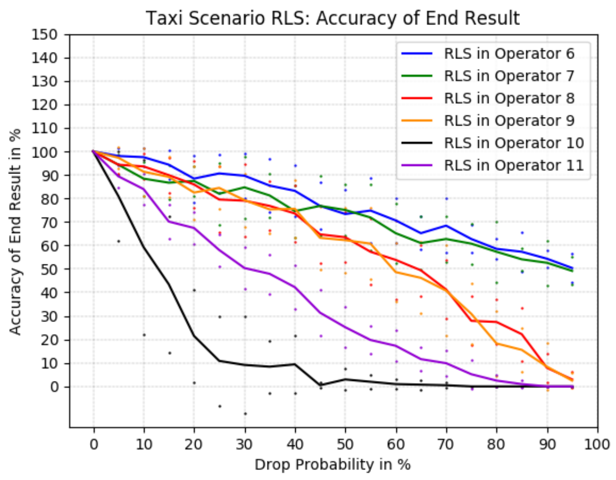


(b)

Figure 5.9: Taxi Scenario: This figures depict the stream quality after operator 10.



(a)



(b)

Figure 5.10: Taxi Scenario: This figures depict the accuracy of the end result.

5.4.1 Taxi Scenario RLS: Stream Quality after Operator 1

The development of stream quality after operator 1 can be seen in Figure 8.1. It shows the stream quality for LS at one operator at a time for every tested drop probability. In the following, we explain the results seen in Figure 8.1.

ω_1: The source of Ω, it produces all events from read in data files. The decline in stream quality after ω_1 is linear $(m = -1, b = 100)$ to the increase in drop probability. As such the line in Figure 8.1a for this operator has a slope of -1.

This is a forwarding operator, the influence of RLS on the stream quality after ω_1 is thus exactly as expected in.

ω_2-ω_{11}: These operators are located after ω_1 in Ω (see Figure 4.4). Thus load shedding at these operators can not retroactively influence the stream quality after ω_1. The stream quality after ω_1 remains therefore at 100% if we use LS at these operators.

5.4.2 Taxi Scenario RLS: Stream Quality after Operator 2 and 3

In the following, we discuss the development of stream quality after operator 2 (Figure 8.2) and operator 3 (Figure 8.3). The figures show the stream quality for LS at one operator at a time for every tested drop probability.

ω_1: The source of Ω, it produces all events from read in data files. The decline of the stream quality after ω_2 and ω_3 is linear $(m = -1, b = 100)$ to the increase in drop probability at ω_1.

This is a forwarding operator, the influence of RLS on the stream quality after ω_2 and ω_3 is therefore exactly as expected.

ω_2-ω_3: ω_2 is parallel to ω_3 on a different branch of Ω. ω_2 sends its output to ω_4, ω_3 sends it output to ω_5. This can be seen in Figure 4.4. These operators are aggregating input events, that contain information on taxi rides on the same day, into one output event. In the majority of windows, we have only events of a single day and aggregate them into one output event. Therefore RLS here does not impact the number of output events at all. The number of output events after the operator is the measure for the stream quality. Therefore the stream quality is after ω_2 and ω_3 is not affected at all if RLS is applied to these operators. This is true for the drop probabilities we use in this experiment – up to 95% drop probability. At 100% percent drop probability, we would have a stream quality of 0%.

These are merge operators, the influence of RLS on the stream quality after ω_2 and ω_3 is therefore as expected.

ω_4-ω_{11}: These operators are located after ω_2 and ω_3 in Ω. (see Figure 4.4). Thus load shedding at these operators can not retroactively influence the stream quality after ω_2 and ω_3. The stream quality after ω_2 and ω_3 remains therefore at 100% if we use LS at these operators.

5.4.3 Taxi Scenario RLS: Stream Quality after Operator 4 and 5

In the following, we discuss the development of stream quality after operator 4 (Figure 8.4) and operator 5 (Figure 8.5). The figures show the stream quality for LS at one operator at a time for every tested drop probability.

ω_1: The source of Ω, it produces all events from read in data files. The decline of the stream quality after ω_2 and ω_3 is linear ($m = -1, b = 100$) to the increase in drop probability at ω_1.

This is a forwarding operator, the influence of RLS on the stream quality after ω_4 and ω_5 is therefore exactly as expected.

ω_2-ω_3: ω_2 is parallel to ω_3 on a different branch of Ω. ω_2 sends its output to ω_4, ω_3 sends it output to ω_5. This can be seen in Figure 4.4.

These operators are aggregating input events, that contain information on taxi rides on the same day, into one output event. In the majority of windows, we have only events of a single day and aggregate them into one output event. Therefore RLS here does not impact the number of output events at all. The number of output events after the operators is the measure for the stream quality. Therefore the stream quality is after ω_4 and ω_5 is not affected at all if RLS is applied to these operators. This is true for the drop probabilities we use in this experiment – up to 95% drop probability. At 100% percent drop probability, we would have a stream quality of 0%.

These are merge operators, the influence of RLS on the stream quality after ω_4 and ω_5 is therefore as expected.

ω_4-ω_5: ω_4 is parallel to ω_5 on a different branch of Ω. ω_4 sends its output to ω_6, ω_5 sends it output to ω_7. This can be seen in Figure 4.4. These operators compute average values for the aggregated input events they receive. Thereafter they send the events to the next operator. RLS at ω_4 or ω_5 degrades the stream quality after these events linearly ($m = -1, b = 100$) to the drop probability. This can be seen in Figure 8.4 and Figure 8.5.

These events are of type forwarding, because they send on all input events. The influence of RLS on the stream quality after ω_4 and ω_5 is therefore as expected

ω_6-ω_{11}: These operators are located after ω_4 and ω_5 in Ω. (see Figure 4.4). Thus load shedding at these operators can not retroactively influence the stream quality after ω_4 and ω_5. The stream quality after ω_4 and ω_5 remains therefore at 100% if we use LS at these operators.

5.4.4 Taxi Scenario RLS: Stream Quality after Operator 6 and 7

In the following, we discuss the development of stream quality after operator 6 (Figure 8.6) and operator 7 (Figure 8.7). The figures show the stream quality for LS at one operator at a time for every tested drop probability.

ω_1: The source of Ω, it produces all events from read in data files. The decline of the stream quality after ω_6 and ω_7 is roughly linear to the increase in drop probability at ω_1. The slope of the line is flatter than -1 (Figure 8.6, Figure 8.7). This is because ω_6 and ω_7 receive heavily aggregated input events and then aggregate them even further. The number of events that ω_1 sends, that results in one output event in ω_6 and ω_7 can vary. It is thus not represented by a linear relationship.

ω_2-ω_3: ω_2 is parallel to ω_3 on a different branch of Ω. ω_2 sends its output to ω_4, ω_3 sends it output to ω_5. This can be seen in Figure 4.4.

These operators are aggregating input events, that contain information on taxi rides on the same day, into one output event. In the majority of windows, we have only events of a single day and aggregate them into one output event.

Therefore RLS here does not impact the number of output events at all. The number of output events after the operators is the measure for the stream quality. Therefore the stream quality after ω_6 and ω_7 is not affected at all if RLS is applied to these operators until high drop probabilities are reached. For high drop probabilities at ω_2 and ω_3 a slight reduction in stream quality after ω_6 (in case of RLS at ω_2), or ω_7 (in case of RLS at ω_3) can be seen. This is true for the drop probabilities we use in this experiment – up to 95%. At 100% percent drop probability, we would have a stream quality of 0% after ω_6 and ω_7.

These are merge operators, the influence of RLS on the stream quality after ω_6 and ω_7 is therefore as expected.

ω_4-ω_5: ω_4 is parallel to ω_5 on a different branch of Ω. ω_4 sends its output to ω_6, ω_5 sends it output to ω_7. This can be seen in Figure 4.4.

These operators compute average values for the aggregated input events they receive. Thereafter they send the events to the next operator. RLS at ω_4 or ω_5 degrades the stream quality after ω_6 or ω_7 linearly to the drop change, as can be seen in Figure 8.6, Figure 8.7.

Here we are dropping aggregated input events when load shedding is applied. Therefore we have a higher degradation in stream quality after ω_6 (if RLS is done at ω_4) or ω_7 (if RLS is done at ω_5) compared to RLS at ω_1. The reason therefore is that we drop non aggregated events at ω_1. Thus we are losing more information per dropped event if we use RLS at ω_4 or ω_5.

These events are of type forwarding, because they send on all input events. The influence of RLS on the stream quality after ω_6 and ω_7 is therefore as expected.

ω_6-ω_7: ω_6 is parallel to ω_7 on a different branch of the Ω. ω_6 sends its output to ω_8, ω_7 sends it output to ω_9. This can be seen in Figure 4.4. These are operators of the same type as ω_2 and ω_3. The aggregate input events with information from the same day into one output event. But in contrast to ω_2 and ω_3 we have here several events of different days in one window. Thus higher RLS drop probabilities lower the number of output events. This reduces the stream quality after these operators. This can be seen in Figure 8.6, Figure 8.7.

These are merge operators, the influence of RLS on the stream quality after ω_6 and ω_7 is therefore as expected.

ω_8-ω_{11}: These operators are located after ω_6 and ω_7 in Ω. (see Figure 4.4). Thus load shedding at these operators can not retroactively influence the stream quality after ω_6 and ω_7. The stream quality after ω_6 and ω_7 remains therefore at 100% if we use LS at these operators.

5.4.5 Taxi Scenario RLS: Stream Quality after Operator 8 and 9

In the following, we discuss the development of stream quality after operator 8 (Figure 5.8) and operator 9 (Figure 8.9). The figures show the stream quality for LS at one operator at a time for every tested drop probability.

ω_1: The source of Ω, it produces all events from read in data files. The decline of the stream quality after ω_8 and ω_9 is only noticeable for higher drop probabilities at ω_1. This is because ω_8 and ω_9 receive heavily aggregated input events and then aggregate them even further. The number of events that are sent by ω_1 that result in one output event in ω_8 and ω_9 can vary. A small number of events send from ω_1, containing information from one day, can result in one output event. Adding a great number of additional events from the same day would result also in only one output. There is only a small decrease in stream quality for higher drop probabilities at ω_1 (Figure 5.8,Figure 8.9).

ω_2-ω_3: ω_2 is parallel to ω_3 on a different branch of Ω. ω_2 sends its output to ω_4, ω_3 sends it output to ω_5. This can be seen in Figure 4.4.

These operators are aggregating input events, that contain information on taxi rides on the same day, into one output event. In the majority of windows, we have only events of a single day and aggregate them into one output event. Therefore RLS here does not impact the number of output events at all. The number of output events after the operators is the measure for the stream quality.

When using RLS at ω_2 or ω_3 the stream quality after ω_8 and ω_9 is barely reduced until high drop probabilities are reached. For high drop probabilities at ω_2 and ω_3 a slight reduction in stream quality after ω_8 (in case of RLS at ω_2), or ω_9 (in case of RLS at ω_3) can be seen. This is true for the drop probabilities we use in this experiment – up to 95%. At 100% percent drop probability, we would have a stream quality of 0% after ω_8 and ω_9.

These are merge operators, the influence of RLS on the stream quality after ω_8 and ω_9 is therefore as expected.

ω_4-ω_5: ω_4 is parallel to ω_5 on a different branch of the Ω. ω_4 sends its output to ω_6, ω_5 sends it output to ω_7. This can be seen in Figure 4.4.

The same reasons as seen for RLS in ω_1, ω_2 or ω_3 apply here as well. But when using RLS in ω_4 or ω_5 the resulting loss in stream quality at ω_8 and ω_9 is greater. The reason is, that the input events at ω_4 and ω_5 are already much more aggregated then at the preceding operators. Thus more information is lost when dropping one event. This results in a higher degradation in the stream quality after ω_8 (when using RLS at ω_4) or ω_9 (when using RLS at ω_5).

ω_6-ω_7: ω_6 is parallel to ω_7 on a different branch of Ω. ω_6 sends its output to ω_8, ω_7 sends it output to ω_9. This can be seen in Figure 4.4. These are operators of the same type as ω_2 and ω_3. They aggregate input events with information on the same day into one output event. RLS at these operators leads to the same result as RLS at ω_4 or ω_5, for the same reasons. This is the case because the level of aggregation of the input events is the same for ω_4 – ω_7. Therefore we lose the same amount of information when dropping one event. The resulting loss in stream quality after ω_8 and ω_9 is, as a result, the same.

ω_8-ω_9: ω_8 is parallel to ω_9 on a different branch of the Ω. Both operators send their output to ω_{10}. This can be seen in Figure 4.4.

These are operators of the same type as ω_2, ω_3, ω_6 and ω_7. They aggregate input events with information on the same day into one output event. At this point in Ω, the input events are at the highest level of aggregation. Thus we lose more information when dropping one event. This results in a higher degradation in the stream quality after ω_8 (when using RLS at ω_8) or ω_9 (when using RLS at ω_9).

ω_{10}-ω_{11}: These operators are located after ω_8 and ω_9 in Ω. (see Figure 4.4). Thus load shedding at these operators can not retroactively influence the stream quality after ω_8 and ω_9. The stream quality after ω_8 and ω_9 remains therefore at 100% if we use LS at these operators.

5.4.6 Taxi Scenario RLS: Stream Quality after Operator 10

In the following, we discuss the development of stream quality after operator 10 (Figure 5.9). The figure shows the stream quality for LS at one operator at a time for every tested drop probability.

ω_1: The source of the Ω, it produces all events from read in data files. The decline of the stream quality after ω_{10} is only significant for higher drop probabilities at ω_1. This is because ω_{10} receives heavily aggregated input events and then aggregates them even further. The number of events ω_1 sends that result in one output event in ω_{10} can vary. A small number of events send from ω_1, containing information from one day, can result in one output event. Adding a great number of additional events from the same day would result also in only one output.

There is only a small decrease in stream quality for higher drop probabilities at ω_1 (Figure 5.9).

ω_2-ω_3: ω_2 is parallel to ω_3 on a different branch of the Ω. ω_2 sends its output to ω_4, ω_3 sends it output to ω_5. This can be seen in Figure 4.4.

RLS at these operators leads to a similar result as RLS at ω_1, for the same reasons. For drop probabilities that are higher than 65% the result for load shedding at ω_2 and ω_3 is better than for RLS at ω_1. This is because Ω splits after ω_1. ω_2 and ω_3 each get half of ω_1's output events as input. Therefore RLS at ω_1 drop twice as many events than RLS at ω_2 or ω_3 with the same drop probability. As a result, RLS at ω_2 or ω_3 does not degrade the stream quality at ω_{10} as much as RLS at ω_1 does.

ω_4-ω_5: ω_4 is parallel to ω_5 on a different branch of the Ω. ω_4 sends its output to ω_6, ω_5 sends it output to ω_7. This can be seen in Figure 4.4. Again the same explanation as for RLS in ω_1, ω_2 or ω_3 applies here as well. But when using RLS in ω_4 or ω_5 the resulting loss in stream quality at ω_{10} is greater. The reason is that the input events at ω_4 and ω_5 are already much more aggregated than at the preceding operators. Therefore dropping a single event loses more information. That results in a higher degradation in the stream quality after ω_{10} compared to using RLS at ω_2 or ω_3.

ω_6-ω_7: ω_6 is parallel to ω_7 on a different branch of the Ω. ω_6 sends its output to ω_8, ω_7 sends it output to ω_9. This can be seen in Figure 4.4.

These are operators of the same type as ω_2 and ω_3. They aggregate input events, with information of the same day, into one output event. RLS at these operators leads to the same result as RLS at ω_4 or ω_5, for the same reasons. This is the case because the level of aggregation of the input events is the same for ω_4 – ω_7. Thus dropping one event loses the same amount of information. The loss in stream quality after ω_{10} is, as a result, the same.

ω_8-ω_9: ω_8 is parallel to ω_9 on a different branch of the Ω. Both operators send their output to ω_{10}. This can be seen in Figure 4.4.

These are operators of the same type as ω_2, ω_3, ω_6 and ω_7. They aggregate input events, with information of the same day, into one output event. At this point in Ω, the input events are at the highest level of aggregation. Therefore dropping one event loses more information. This results in a higher degradation in the stream quality after ω_{10} when using RLS at ω_8 or ω_9.

ω_{10}: This is a merge operator. It merges input events, representing information from the same day, into one. There is always a pair of two events that can only be merged with each other. Only after a successful match, it sends the resulting events to ω_{11}.

As we can see in Figure 5.9, RLS at ω_{10} leads to the greatest loss in stream quality. The reason is that an event shed at ω_{10} means the other event of that day cannot be merged any more. Therefore it can not be sent on. Thus shedding only half the input events could, in the worst case, lead to not finding a single output event. If we shed an event that has a counterpart that already was dropped, we lose no further output event. If an event is shed which has a counterpart that was not dropped, one more output event is lost. Therefore we have a high variance in stream quality when RLS is used at ω_{10}.

This is merge operator with restriction, the influence of RLS on the stream quality after ω_{10} is therefore as expected.

ω_{11}: ω_{11} is located after ω_{10} in the Ω (see Figure 4.4). Thus load shedding at this operator can not influence the stream quality after ω_{10}. The stream quality after ω_{10} remains therefore at 100% if we use LS at ω_{11}.

5.4.7 Taxi Scenario RLS: Accuracy of the End Result

The accuracy of the end result is the stream quality after operator 6. The development of the accuracy of end result can be seen in Figure 5.10. It shows the accuracy of end result for LS at one operator at a time for every tested drop probability. In the following, we explain the results seen in Figure 5.10.

ω_1: The source of Ω, it produces all events from two read in data files. ω_1 sends the information from one file to ω_2 and the information of the other file to ω_3. In both branches, events are aggregated. For all important attributes, we compute average values in these aggregated events.

RLS at ω_1 does only remove a small fraction of data in the aggregated events. As such it only greatly impacts the accuracy of end result at higher drop probabilities. At higher drop probabilities at ω_1 a great majority of events, that could later be merged, get dropped. This results in a greater loss of accuracy because the averaged values of the aggregated events are greatly altered. This does impact the pattern search at ω_{11} significantly. We can see this in Figure 5.10, where after 60% drop probability the line for LS in ω_1 has a much steeper slope than before.

ω_2-ω_3: ω_2 is parallel to ω_3 on a different branch of Ω. ω_2 sends its output to ω_4, ω_3 sends it output to ω_5. This can be seen in Figure 4.4.

These operators are aggregating input events that contain information on taxi rides on the same day, into one output event. In the majority of windows, we have only events of a single day and aggregate them into one output event.

We can see in Figure 5.10, that RLS at ω_2 does impact the accuracy of the end result the least of all load-shedding locations. The reason that RLS at ω_3 leads to a greater degradation in accuracy of the end result is data related.

The information gathered in the branch of ω_2, has greater differences between the individual averaged values than the averaged values on the branch of ω_3. As such pattern-finding at ω_{11} is more stable on data passing through ω_2 when using load shedding. The reason therefore is that small changes to the averaged attribute values are less likely to destroy a pattern. E.g. pattern 1-2-3 is less stable than 1-8-22.

Overall RLS an ω_2 or ω_3 leads to a more accurate end result than load shedding at ω_1. This is because after ω_1 the data splits and thus only half the number of events pass through each ω_2 and ω_3. Thus we lose only half as much information through RLS with the same drop probability at ω_2 or ω_3.

ω_4-ω_5: ω_4 is parallel to ω_5 on a different branch of Ω. ω_4 sends its output to ω_6, ω_5 sends it output to ω_7. This can be seen in Figure 4.4.

These operators compute average values for the aggregated input events they receive. Thereafter they send the events to the next operator.

We can see in Figure 5.10, that RLS at ω_4 does not impact the accuracy of the end result as greatly as RLS at ω_5. The reason that RLS at ω_5 leads to a greater degradation in accuracy of the end result, is data related.

The information gathered in the branch of ω_4, has greater differences between the individual averaged values than the averaged values on the branch of ω_5. As such pattern-finding at ω_{11} is more stable on data passing through ω_4 when using load shedding. The reason therefore is that small changes to the average values are less likely to destroy a pattern. E.g. pattern 1-2-3 is less stable than 1-8-22.

Overall RLS an ω_4 or ω_5 leads to a less accurate end result than load shedding at preceding operators. This is because after ω_2 and ω_3 the data is aggregated. For each dropped event at ω_4 or ω_5 the information of all the events that it was created from is also lost. Thus we lose more information through RLS at ω_4 or ω_5 compared to RLS at preceding operators in Ω with the same drop probability.

ω_6-ω_7: ω_6 is parallel to ω_7 on a different branch of Ω. ω_6 sends its output to ω_8, ω_7 sends it output to ω_9. This can be seen in Figure 4.4.

These are operators of the same type as ω_2 and ω_3. They aggregate input events with information from the same day into one output event. But in contrast to ω_2 and ω_3 we have events of different days in one window. Thus higher RLS drop probabilities lower the number of output events.

RLS at ω_6 or ω_7 has roughly the same impact on the accuracy of end the result. The reason that there is no difference between RLS at this operators is, that at this point great amounts of information are aggregated into every single event. Dropping even a small number of events can impact the aggregated values at ω_{11} greatly. It can destroy existing patterns or create new ones. Each event contains much more information compared to events at earlier operators. Therefore RLS here has a bigger impact on the accuracy of the end result than RLS at preceding operators in Ω. This can be seen in Figure 5.10.

ω_8-ω_9: ω_8 is parallel to ω_9 on a different branch of Ω. Both operators send their output to ω_{10}. This can be seen in Figure 4.4.

These are operators of the same type as ω_2, ω_3, ω_6 and ω_7. They aggregate input events with information on the same day into one output event. At this point in Ω, the input events are at the highest level of aggregation. Thus we lose more information when dropping one event. Therefore RLS here has a bigger impact on the accuracy of end result than RLS at preceding operators in Ω. This can be seen in Figure 5.10.

ω_{10}: This is a merge operator. It merges input events, representing information from the same day, into one. There is always a pair of two events that can only be merged with each other. Only after a successful match, it sends the resulting events to ω_{11}.

As we can see in Figure 5.10, RLS at ω_{10} leads to the greatest loss in accuracy of end result. The reason is that an event shed at ω_{10} is indirectly shed from all windows in ω_{11}. Hence RLS at ω_{10} does degrade the accuracy of end result more than RLS at ω_{11} does. Because one dropped event here means the loss of information on one whole day at ω_{11}. As can be seen below, dropping one event at ω_{11} is not losing the information on one whole day.

The result of RLS at ω_{10} also leads to the greatest variance in the end results. The reason is that RLS before ω_{10} loses only some data of a day. Load shedding at ω_{11} as mentioned, does not lose the information of one day as a whole. RLS at ω_{10} drops the greatest amount of information per event. Therefore it is impacting the accuracy of the end result the most.

This is amerge operator with restriction, the influence of RLS on the stream quality after ω_{10} is therefore as expected.

ω_{11}: The sink of the Ω. This operator searches for a pattern of two increases in a row in its attributes. The tendency for ω_{11} in Figure 5.10 is worse than a linear line $(m = -1, b = 100)$.

ω_{11} has a window size of five with a slide of one. The decision to shed an event is always done individually for each window. Therefore an event that is shed from one window can still remain in other windows. A pattern that we do not find in one window because of RLS, we can still detect in the next window. Because of this, we do not lose all information represented in an event by shedding a single event in one window. This is the reason why RLS in ω_{11} does not degrade the accuracy of end result as much as RLS in ω_{10} does.

This is a pattern-finding operator with restriction, the influence of RLS on the stream quality after ω_{10} is therefore as expected. We see that shedding whole input events can lower the accuracy of the end result very quickly. If we have, as in this case, merge operators before the pattern-finding operator and use LS on them, the resulting accuracy can be much better.

5.4.8 Taxi Scenario RLS: Latency

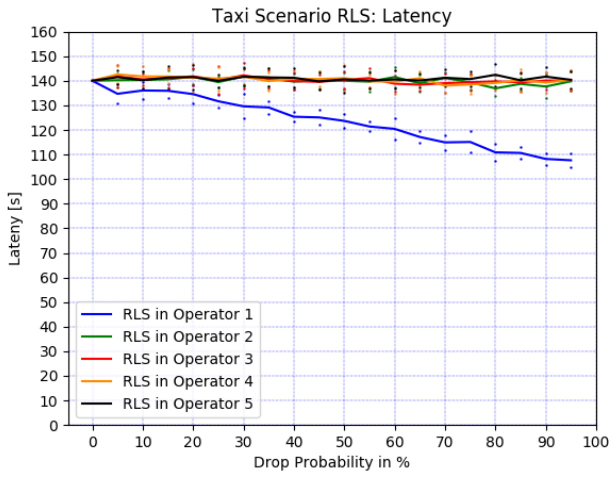
In the following, we examine the latency of the Taxi Scenario experiment for RLS. We examine each operator separately and use the same id as seen in Figure 4.4. The latency for each operator for every tested drop probability can be seen in Figure 5.11.

In this scenario the potential to reduce latency through load shedding is limited. The reading in of the data is responsible for a big part of the latency. Hence the actual processing of the events is responsible for only a small part of the overall latency. Because load shedding can only affect the number of events that are processed, it can only decrease this small chunk of latency.

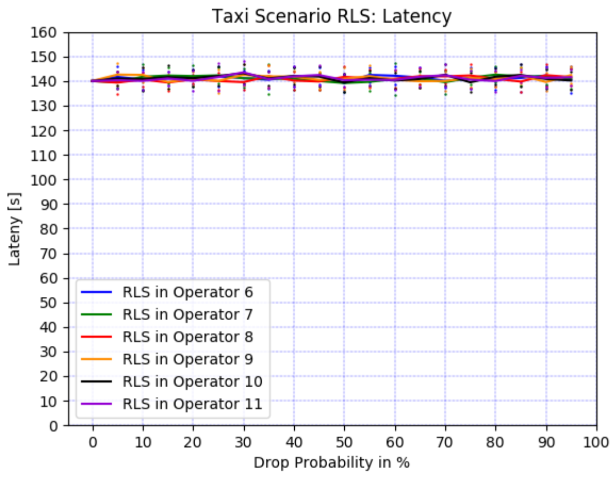
ω_1: The source of Ω. It generates all events here from read in data. Therefore a decline in the events sent from this operator would be assumed to decrease the overall latency linearly ($m = -1, b = 100$) with increasing drop probability. This is not the case here, because ω_1 reads the events in from data files. The reading in of data is responsible for the biggest part of the runtime. Therefore even high RLS drop probabilities yield only about 25% latency reduction for RLS in ω_1.

Still as expected, RLS here at the source of Ω leads to the highest reduction in overall latency.

ω_2-ω_9: At these operators, we aggregate information. To that purpose, we need to wait until a window is closed to send the output events to the next operator. The time we win through not having to process an event is lost through waiting for the next event. There is also a very small overhead for load shedding. As can be seen in Figure 5.11 RLS at these operators does not reduce the latency at all.



(a)



(b)

Figure 5.11: The latency in the Taxi Scenario when using random load shedding.

ω_{10}: At this operator, two halves of information concerning the taxi rides on the same day in NYC are merged. Only a small number of events arrive here. All other events were already aggregated into these input events. Thus ω_{10} spends most of its time waiting and only a very small fraction of its time processing events. The time we win through not having to process an event is lost through waiting for the next event. There is also a very small overhead for load shedding. RLS here does for that reason not impact the latency noticeably, as seen in Figure 5.11.

ω_{11}: ω_{11} searches for patterns in the aggregated data. Like ω_{10} this operator spends most of its time waiting and only a very small amount of time processing events. The time we win through not having to process an event, we lose again through waiting for the next event to arrive. There is also a very small overhead for load shedding.

Additionally ω_{11} is the sink of Ω. Therefore the majority of the processing is already done when an event arrives here. RLS here does for that reason not impact the latency noticeably, as seen in Figure 5.11.

Overall the reduction in latency that is possible through RLS, is compared to e.g. the Fire Scenario less pronounced. The reason is that the Taxi Scenario uses a lot of the runtime to read in the necessary data to generate events. In the Fire Scenario, events are randomly generated and additionally much leaner. The Fire Scenario uses a greater percentage of the runtime to process events at the operator. This time we can reduce through load shedding. But LS can not affect the time used reading in data files.

RLS in this scenario is most effective if used as early as possible in Ω, as seen when using RLS in ω_1. In fact for the above-mentioned reasons, RLS at ω_1 is the only option in Ω to significantly reduce the latency.

5.4.9 Taxi Scenario RLS: General Observations

The prediction for forwarding operators was accurate. ω_1, ω_4 and ω_5 reacts as expected with a linear decrease of stream quality.

When using RLS, aggregation type operators are more robust in their stream quality than other operator types. But it also depends on the number of events that they aggregate into one. If it is a big number of events, RLS results in no noticeable impact on the stream quality, as seen with ω_2 and ω_3.

This changes, if the number of events that they aggregate into one is smaller on average. The impact of RLS on the stream quality should increase in that case. But still, we can expect a better result than the linear ($m = -1, b = 100$) that the can see when using RLS on filter operators like ω_1. Only in the worst-case does the loss in stream quality through LS at this operator type equal the linear line ($m = -1, b = 100$). The worst-case would be that no event can be merged with another event. That would mean that the number of input events would equal the number of output events. In that case, the aggregation operator would act as a forwarding operator. It would show the same reaction to the usage of load shedding.

RLS is impacting the accuracy of end result in this scenario less if it happens early in Ω. Through the aggregation of data, events contain more information the further one traverses down the Ω. Thus RLS drops more information with each event. As such it impacts the aggregated values in which we search pattern in ω_{11} stronger

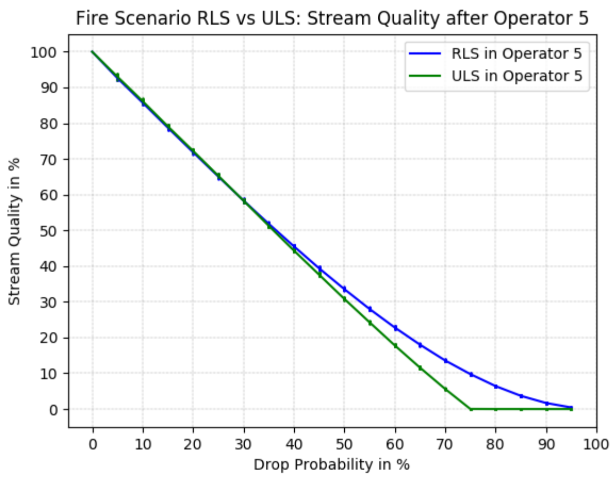
We also see again something we already saw in the other scenarios, about branching in Ω. If Ω branches out and splits the data among these branches, generally it leads to better results to shed load on one of the branches. In this scenario, this is only true for the first operators after the split. Through the aggregation of data, this advantage is negated for the following operators. The reason is, at later points in Ω we shed accumulated information of several primitive events if we shed a single aggregated event.

5.5 All Scenarios: Utility-based Load Shedding

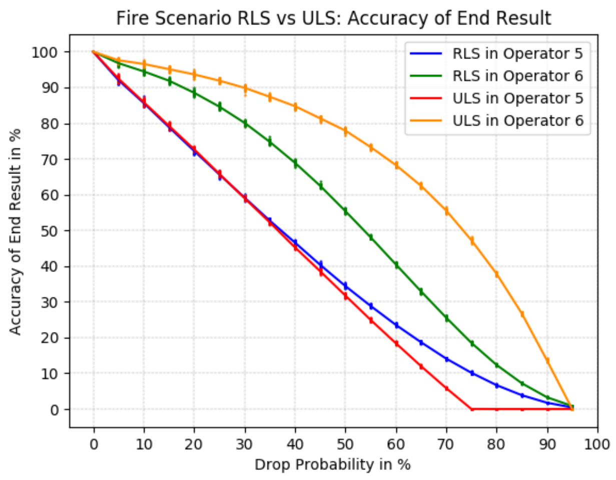
In the following, we examine the results of the experiment with ULS. Afterwards, we discuss the accuracy of the hypothesis. We address the operator as ω, with the id it has in the operator graph Ω of its scenario (Fire Scenario: Figure 4.2, Twitter Scenario: Figure 4.3, Taxi Scenario: Figure 4.4).

We give a brief overview of the inner workings of the operators that are affected by ULS. Afterwards, we explain the impact on the stream quality caused by load shedding at this operator. Lastly, we compare the hypothesis for this event type with the results.

The focus of this work is not the development of load shedders. Instead, we focus on how a loss in stream quality through load shedding propagates. Thus we did not let the load shedder learn the utility values. Instead, we computed the utility values based on the



(a)



(b)

Figure 5.12: Fire Scenario RLS vs ULS:
Figure 5.12a depicts the stream quality after operator 5.
Figure 5.12b depicts the accuracy of the end result.

probabilities of event types to contribute to a match. The utility values are always local for the operator itself. This means that we want to maximize the number of output events at this operator.

5.5.1 Fire Scenario ULS: Accuray of the End Result

The accuracy of end result is the stream quality after operator 6. Figure 5.12a gives a comparison for the resulting stream quality after ω_5 when using RLS and ULS at ω_5. Figure 5.12b gives a comparison for the accuracy of the end result, when we use RLS and ULS on ω_5 or ω_6. Both figures show the stream quality for LS at one operator at a time for every tested drop probability.

ω_1-ω_4: These operators are located before ω_5 in Ω. This can be seen in Figure 4.2. There is no meaningful way to use ULS with a local scope on these operators. Thus we did not use ULS on these operators.

ω_5: We tried several approaches to determine utility-values, that would allow us to outperform RLS. As can be seen in Figure 5.12a we did not succeed. In this case, which performed the best, did we drop 'B's with double the probability than 'A's, because 'A's that can contribute to an match are half the number of 'B's that can contribute. The overall number of 'A's and 'B's is equal, but not every event has a high enough value to contribute.

But important here is for the scope of this thesis, that we can see, that the loss in stream quality after ω_5 directly propagates to the accuracy of end result (see Figure 5.12).

ω_6: ω_6 is a propositional logic operator. We computed the utility values that we used to weight the random drop probability used in RLS as follows. Through using the upper cumulative distribution function we got the result, that 'D's had a 26 times higher utility than 'C's. To make sure that we drop the required number of events even for higher drop probabilities we need to alter the drop probabilities. This was necessary because otherwise, the number of remaining 'D's would be too high for higher drop probabilities. Too many events would remain in total. We multiply the base drop probability to reflect the utility for each event type. We multiply the probability for 'D's with $0,0795$ and for 'C's with $1,0597$. This represents a 13,333 times higher utility for D compared with C. E.g. we aim for a total drop rate of 10%. To achieve that we drop D with 0.795% and C with 10.597%. Because 'C' greatly outnumber 'D's we drop 10% of incoming events in total.

As we can see in Figure 5.12b ULS in ω_6 leads to an overall superior result compared to RLS at ω_6.

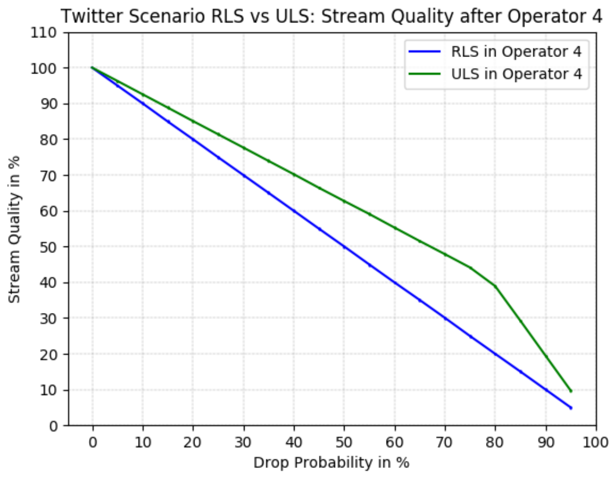
5.5.2 Twitter Scenario ULS: Accuray of the End Result

The accuracy of end result is the stream quality after operator 6. Figure 5.13a gives a comparison for the resulting stream quality after ω_4 when using RLS and ULS at ω_4. Figure 5.12a gives a comparison for the accuracy of the end result, when we use RLS and ULS on ω_5. Both figures show the stream quality for LS at one operator at a time for every tested drop probability.

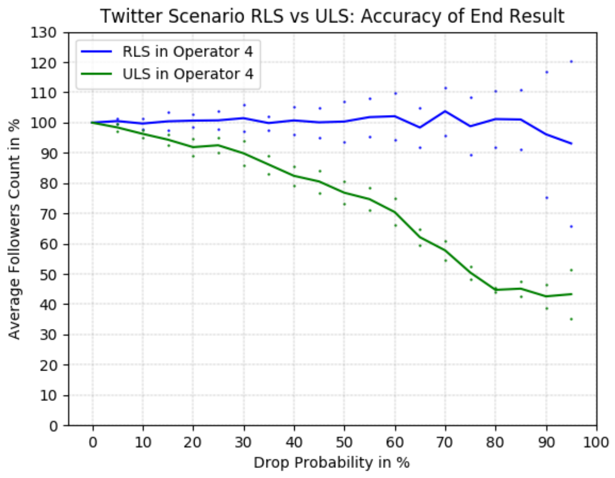
ω_1-ω_3: There is no meaningful way to use ULS with a local scope on these operators. Therefore no ULS was used on these operators.

ω_4: ω_4 is a filter operator. It sends all events to ω_5, which are responses to other Tweets. The utility for each event was determined after the following principle: We hypothesized that tweets of different character lengths would have a different proportion of events that were a reply. The result was, that events that have a character length from 15 to 60 have a probability of 37,88% to be a reply. The rest of the events have a probability of 14,49% of being a reply.

For drop probabilities up to 75% we therefore only dropped the events with low utility. We multiply the base drop probability to reflect the utility for each event type. We multiply the drop probability by 1.2688 for low utility events to compensate for not shedding high utility events. After a 75% target drop probability, we drop all events of low utility and start shedding high utility events. High utility events are shed with a drop probability that is necessary to reach the overall target drop rate.



(a)



(b)

Figure 5.13: Twitter Scenario RLS vs ULS:
Figure 5.13a depicts the stream quality after operator 4.
Figure 5.13b depicts the accuracy of the end result.

E.g. we aim for a total drop rate of 50%. To achieve that we drop low utility events with a probability of 63,44% and do not drop high utility events at all.

We can see this in Figure 5.13a that ULS up to 75% overall drop rate is linear, after this point it gets worse because more high utility events are shed. Overall ULS yields superior results compared to RLS when we look at the stream quality after ω_4.

But if we look at Figure 5.13b we see, that ULS at ω_4 also significantly lowers the accuracy of the end result.It seems that events that were designated as low utility locally in ω_4 are not low utility for the end result. They had a lesser probability to lead to an output event at ω_4, but they had Tweets among them that were necessary for a stable average value at ω_6.

Because the events there not dropped proportionally any more in ω_4, the accuracy of end result is degraded greatly. The variance in the end result for each run is lower when using ULS. But on average it is much worse than the result for RLS, which leads to a stable average result, with a greater variance for higher drop probabilities.

ω_5: For ω_5 RLS already delivers very good results. We could not find utility values that resulted in a greater accuracy of the end result.

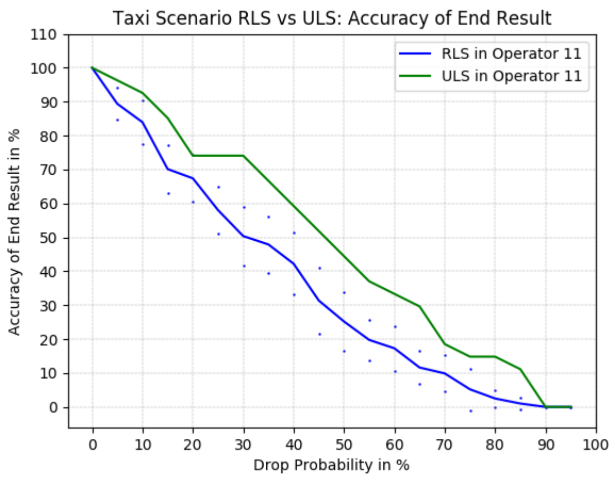


Figure 5.14: Taxi Scenario RLS vs ULS:
The accuracy of the end result.

5.5.3 Taxi Scenario ULS: Accuray of the End Result

The accuracy of end result is the stream quality after operator 11. Figure 5.14 gives a comparison for the accuracy of the end result, when we use RLS and ULS on ω_5. It shows the stream quality for LS at one operator at a time for every tested drop probability.

ω_1-ω_{10}: There is no meaningful way to use ULS with a local scope on these operators. Therefore no ULS was used on these operators.

ω_{11}: The sink of Ω. This operator searches for a pattern of two increases in a row in its attributes.

ω_{11} has a window size of five with a slide of one. The decision to shed an event is always done individually for each window. Therefore an event that we shed from one window can still remain in other windows. A pattern that we do not find in one window because of RLS, we can still detect in the next window. Because of this, we do not lose all information represented in an event by shedding a single event in one window.

We decided on the utility of an event similar to what a machine learning solution would do. For each input event, we analyzed the number of patterns that this event could participate in. With this information, we made a ranking of the utility of each input event. We then drop the required number of events for each drop probability, starting with the event of the lowest utility. Afterwards, we work the way up the utility ranking.

As we can see in Figure 5.14, the result for ULS is better than RLS at all drop probabilities up until 90% drop probability. At this point, too many events were dropped to still find a pattern. Because we use concrete utility values on a certain data set, there is no variance in the accuracy of end result for ULS. We required only one run of the scenario to get the necessary data.

5.5.4 All Scenarios ULS: Latency

All utility-based load shedder we wrote were more complex than a random load shedder. As such it is no wonder, that latency we measure was always a few percentage higher, when a ULS was used, compared to a RLS on the same operator with the same drop rate.

6 Conclusion

In the following, we present the new information we gained during the evaluation of the experiments. The results are presented for each operator type separately. For each operator type we examine how it reacts to losses in stream quality caused by Random Load Shedding (RLS) or Utility-based Load Shedding (ULS). We show, how changes in its input data stream propagate after each operator type.

If we speak of repairing the stream quality, we mean that the stream quality after the operator is higher than it was before.

For two input sources we assume, that the stream quality is the average of the stream quality of both sources, weighted by the total number of events sent per source.

6.1 Propositional Logic Operator (PLO)

In the case that we shed all input events before a PLO proportionally, e.g. with Random Load Shedding (RLS), the loss in stream quality after the PLO is linear ($m = -1, b = 100$) to the percentage of dropped events. It has thus the same stream quality that was also measured after its sources(s).

In next part, we discuss how Load Shedding (LS) affects a PLO if it is not done proportionally. For the following, let us assume that a PLO gets two types of input events. We need one event of both types to detect a match.

For load shedding at a PLO, it is important to know the composition of the input events.

If among the input events there is no limiting factor, using LS proportionally (e.g. RLS) at all event types leads to the best results.

But if there is a limiting factor among the input events, it is the best choice to drop the non-limiting factor preferably. We can only do this until the non-limiting factor, through LS, becomes the limiting factor itself. If we reach that point then the operator processes the limiting and non-limiting factors in equal numbers. From that point on we shed the events in such a way, that this equality in numbers is maintained.

If we have more than two event types, shedding the most numerous of the event types is preferable. Shedding only this event type, leads to the best result, as long as the event type is not reduced to the same numbers as the next most numerous event type. If that happens, we begin with shedding the other event types. The order is from the most numerous event types left to the limiting factor.

The potential to use LS and still achieve a very accurate result depends on how many times the non-limiting factor outnumbers the limiting factor. E.g. if an operator wants to find a match $A \wedge B \rightarrow C$ and a third of all input events are 'A's and the rest 'B's, then the limiting factor A is outnumbered by a factor of 2.

In the following, we assume that there are only two incoming event types at the PLO. The same results can also be applied to a PLO with more than two incoming event types. But in that case, we would talk about how many times the most numerous event type outnumbers the second most numerous event type and so on for all event types.

6.1.1 PLO: Load shedding at the non-limiting factor

In the following, we discuss how targeting the non-limiting factor for load shedding before the PLO influences the stream quality after the PLO.

If the LS is done at the non-limiting factor the loss of stream quality does not directly propagate after the PLO. The PLO can repair the stream quality to a significant degree.

There are generally spare events of the non-limiting event type in a window. We can drop a certain number of them, before we start to greatly impact the number of matches found. The result is better than proportionally shedding of all event types, e.g. through RLS. The higher the factor by which the non-limiting factor outnumbers the limiting factor, the better the result is, if we compare it with RLS with the same drop rate.

The degree by which the PLO can repair the stream quality also rises with the factor by which the non-limiting factor outnumbers the limiting factor. We can see this in our experiment, where LS at the non-limiting factor, that outnumbered the limiting factor by a factor of 23, leads to a greatly improved stream quality after the PLO.

6.1.2 PLO: Load shedding at the limiting factor

In the following, we discuss how targeting the limiting factor for load shedding before the PLO influences the stream quality after the PLO.

Load shedding at the limiting factor decreases the stream quality. The degree, by which the stream quality sinks, increases, the more the non-limiting factor outnumbers the limiting factor. We can see this in our experiment in the Fire Scenario.

6.1.3 PLO: Summary

If there is a limiting input event type dropping the non-limiting event type yields the best result in terms of stream quality. We can only do this until the non-limiting factor, through LS, becomes the limiting factor itself. If we reach that point then the operator processes the limiting and non-limiting factors in equal numbers. From that point on we shed the events in such a way, that this equality in numbers is maintained.

The higher the degree by which the non-limiting factor outnumbers the limiting factor, the better is the stream quality after the PLO. The potential to repair the stream quality is also higher, the higher the degree by which the non-limiting factor outnumbers the limiting factor.

If there is no limiting factor, random load shedding leads to the best results.

6.2 Pattern-finding Operator

Load shedding that negatively influences the attributes of an event, but does not shed the event itself, is preferable to shedding input events completely for this type of operator. We can see this in the Taxi Scenario with LS at the merging operators. This is at least the case if the patterns are relatively stable. E.g. the successive increase 1-2-3 is not as stable under attribute altering LS than the pattern 1-8-23. That is the case because the difference between the values is smaller and therefore the pattern is more likely to be broken if the attributes are altered even slightly.

For unstable patterns, negatively influencing the attributes could lead to many false positives and false negatives. This could reduce the stream quality as much as shedding whole input events. But for the testing in this work, we counted every found pattern as a valid detected complex event. In the Taxi Scenario, the patterns were very stable and the window sizes small, so there are few false positives.

In general, if we do not want to find false positives, every reduction in stream quality before this operator type reduces the stream quality afterwards even more. But if we do accept false positives as valid, and have greater window sizes, the resulting stream quality could become unpredictable without data related knowledge.

6.3 Average-building Operator

We did not take the number of output events as the criterion for the average-building operator. This number would remain the same even for high drop probabilities. Instead we took the accuracy of the average as criterion for the stream quality.

If we proportionally shed input events, the computed average value remains on average 100% accurate. It does not matter if this is achieved through LS at the average-building operator itself or the preceding operators.

If the operator graph branches or has multiple sources, we should not use load shedding before the data paths merge. Alternatively, we should do it on both branches in the same ratio. If we don't do it in the same ratio, shedding on the branches can alter the resulting average value, depending on the data. The divergence in the average becomes increasingly stronger with higher drop probabilities.

In summary, we should do LS for this operator type in a way, that we drop input events in the same ratio, e.g. through RLS. This leads to a very good and stable result. On average the stream quality in the Twitter Scenario is roughly 100% for all drop probabilities. At a 95% drop probability, we have a standard deviation that is only up to 20%. For lower drop probabilities the standard deviation is much lower.

This operator type (or preceding operators) is a very good choice for LS if done in the same ratio for all input event types.

For the above mentioned reasons we have a great potential to repair the stream quality at this operator type, even if the input has a very low stream quality.

6.4 Merge Operator

In this work, we have defined two types of merge operators. One type merges all events in a window that can be merged, before sending the merged and non-merged events on. The second type only sends on the successfully merged events.

In the following, we discuss both these types separately.

6.4.1 Merge Operator: Merge if possible

The impact of LS on the stream quality depends on how many output events there are per window (assuming big window sizes). If we e.g. have a window of 100 events that we can all merge into one output event, even dropping 99 events results in the same number of output events. And the number of output events is how we measure the stream quality.

In comparison to this, if we have a window of 100 events and only two events can on average merge with each other we could potentially have 50 output events. If we drop 99 events, in that case, we would see a drastic loss in stream quality.

If the number of output events is very low, then LS at this or preceding operators will not influence the stream quality, i.e. the number of output events. But the attributes of those merged events could still be affected. Especially, in the case where the merged events have attributes that depend on aggregated data like e.g. averaged values.

For a low number of output events per window, this operator has the potential to improve the stream quality to a high degree. But the content of the attributes of the merged events could get increasingly inaccurate for higher drop probabilities.

If the number of output events per window is higher, the loss in stream quality will slowly move towards a linear line ($m = -1, b = 100$). The more output events per window, the closer the loss in stream quality gets to this line. This is because the operator becomes more similar to an operator that just forwards events. In this extreme case, we would have a loss in stream quality represented by that linear line ($m = -1, b = 100$). This is the lower bound for stream quality for load shedding at this operator type.

6.4.2 Merge Operator: Only forward if merged successfully

The best case for this operator type looks nearly the same as the best case for the merge operator without restriction. If we e.g. have a window of 100 events that we can all merge into one output event, even dropping 98 events results in the same number of output events.

But if we have a window of 100 events and only two events can on average merge with each other we could potentially have 50 output events. But even dropping only 50 of those events, could in the worst case lead to the generation of zero output events. We can limit this by using utility-based load shedding. For each event we shed directly, other events are most likely shed indirectly. That happens, if through the shedding of a different event they can not be merged any longer. Thus they can also not be sent on. If we also can shed this indirectly dropped events directly at preceding operators, the resulting stream quality could improve.

The loss in stream quality becomes less severe per dropped event, if there are more potential merge partner per event. The reason is that we have less indirectly dropped events. There is a higher probability for events of the same type to still find a merge partner.

6.5 Filter Operator

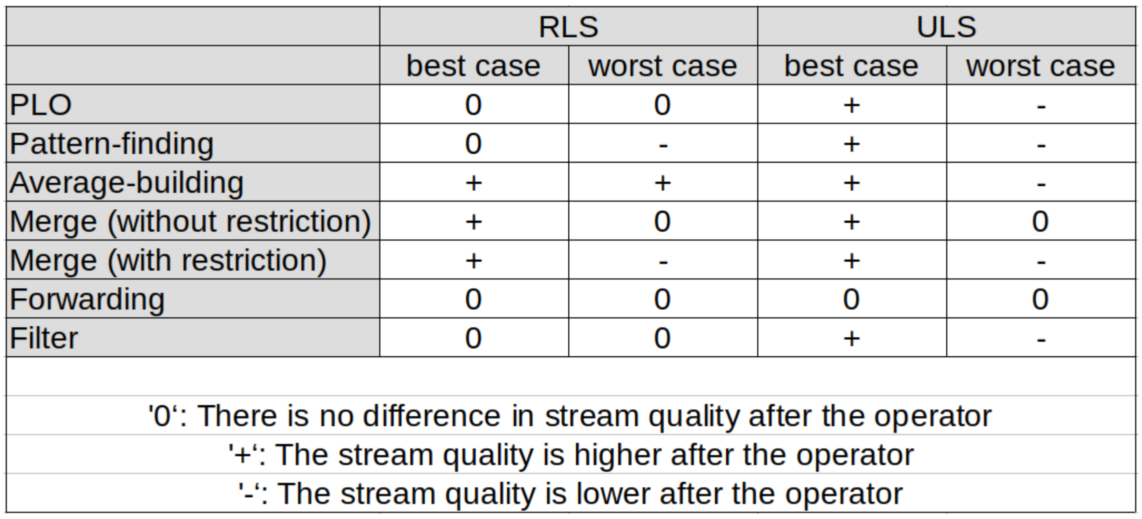
Filter operators propagate the stream quality directly to the following operator. When using RLS, the loss in stream quality before the filter operator equals the loss in stream quality after the filter operator.

We have seen that when using RLS, this operator type can not improve the stream quality.

ULS can improve the stream quality, if we choose good utility values. In the best case, the stream quality could even be repaired up to 100%.

6.6 Forwarding Operator

Forwarding operators can not improve the stream quality. It forwards every input event it receives, after it potentially did some operation on it, like a computation. Therefore we have a linear dependency between the number of output events and the number of input events. A forwarding operator can therefore not compensate a loss in stream quality that happened before it.



	RLS		ULS	
	best case	worst case	best case	worst case
PLO	0	0	+	-
Pattern-finding	0	-	+	-
Average-building	+	+	+	-
Merge (without restriction)	+	0	+	0
Merge (with restriction)	+	-	+	-
Forwarding	0	0	0	0
Filter	0	0	+	-

'0‘: There is no difference in stream quality after the operator
'+‘: The stream quality is higher after the operator
'-‘: The stream quality is lower after the operator

Figure 6.1: Overview of the how the operator types propagate losses in stream quality.

6.7 Overview

In the following, we give a brief overview of how the different operator types react to load shedding. In Figure 6.1 we assume that an operator has one source. For two sources we assumed, that the stream quality would be the average of the stream quality of both sources, weighted by the total number of events sent per source. That would equal the stream quality of one source, which does the same as both of these sources combined. E.g. shedding at only one of the two sources could be simulated by using ULS at a single source.

We further assume that when using ULS, we select good utility values. That means utility values that would lead to better results than RLS.

For the merge operators it is dependent on the data and the window size if we have the best case, the worst case or something in between. For all other operator the worst case for ULS should never happen, if we assume that we choose good utility values.

For the pattern-finding operator we assumed, that false positives are not counted as valid matches. Otherwise RLS could, in the best case, improve the stream quality.

7 Summary and Outlook

At the beginning of this work, we examined several publications on the topic of load shedding. Most of these publications discussed how to make a good load shedder for CEP or stream processing systems. Others gave a general overview of the topic of load shedding in CEP or stream processing systems.

But no publication we could find examined, how the inaccuracies in the data stream, caused by load shedding, propagate through the operator graph. Still, the related works gave a good starting point for the topic of load shedding. They gave us the knowledge we needed to set up our experiment and evaluate its results.

For the practical part of this thesis, we modified an existing Java Framework. Then we used it for our experiments. For the experiments, we created three different scenarios. They all use different operator types, a different topology, and different data sets. For each of the scenarios, the experiment followed the same pattern. We did load shedding on each operator in the topology, always one at a time. To analyze the output data of the experiments we created several Python scripts.

In the evaluation, we examined how load shedding influenced the stream quality after the operators. Further, we examined how the resulting inaccuracies propagated through the operator graph. We examined the stream quality after each operator in the topology, which input we influenced at some point through load shedding.

One general result of the evaluation is, that the earlier in the topology we shed load, the better is the reduction of the overall latency. But the resulting stream quality and the accuracy of the end result are often not optimal. We want to find the optimal trade-off between high accuracy and low latency. But the spot to shed load to achieve this optimal trade-off is operator graph specific. As such we cannot give a general recommendation.

In the following, we give a brief overview of the results for each operator type and how they propagate a prior loss in stream quality.

For Propositional Logic Operators (PLO) the result was, that if there is a limiting factor among the input events, one should always preferably shed the events, that are non-limiting. We can only do this up to the point where the non-limited factor becomes limited itself. After that point, we need to start shedding the original limiting factor as well. If we shed a non-limiting factor prior to the PLO, then the PLO can improve the stream quality over this connection. This means that the stream quality after the PLO is higher than it was after the input operator in question. The improvement is stronger, the more outnumbered the limiting factor is. The stronger the limiting factor is outnumbered, the more events we can shed at the PLO or before, while still having a very accurate result. If there are more than two events types needed for a match, we should shed starting with the most common event type and than work our way down to the limiting factor.

A filter operator will propagate the loss in accuracy before the operator directly if we use random load shedding. This means that the stream quality before and after the operator is the same. For utility-based load shedding, the filter operator can potentially repair the stream quality to some degree. This is the case if prior to the filter operator we preferably dropped events, that could not pass the filter.

Merge operators, that merge events together if certain conditions apply, give a very accurate result when using load shedding on this operator or prior to it. The result is most stable if the operator can merge many events in a window into one. If on average the operator merges a high number of events into one, it can repair the stream quality to a high degree. If the operator can merge no events and sends on all events single, the result of load shedding is like seen in the filter/forwarding operators.

The next type of merge operator has the restriction, that it only forwards events if there was a successful merge. Load shedding leads to a potentially great reduction in stream quality after this operator. The loss in stream quality is great, if the operator can on average only merge a small number of events. But it can, for the best case that many events can merge into one, be as stable the merge operator without restriction.

The pattern-finding operator has a very high loss in accuracy if we shed whole events. If we have merging operators on the path to the pattern-finding operator, load shedding at these leads to much better results. The reason is, that if we have relatively stable patterns, load shedding that only influences the attributes of an event, leads to a much more accurate result. This kind of load shedding is far less likely to change a stable pattern.

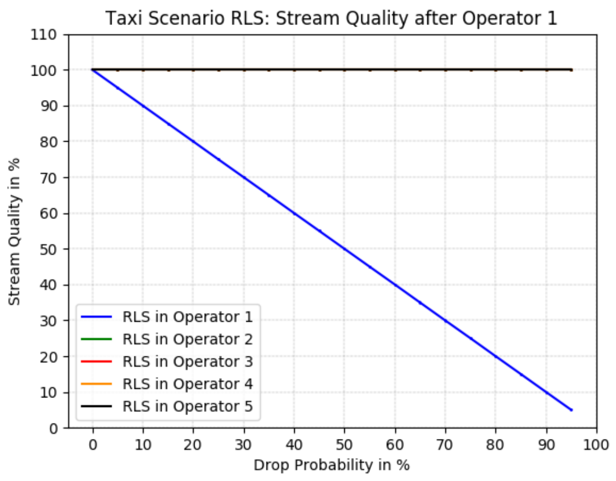
The average-building operator is very stable when we use random load shedding. If we shed load in all branches, that lead to this operator proportionally, we get a very stable result. It remains on average at 100% accuracy of the result, with only small variance in the individual runs for higher drop probabilities. As such the average-building operator has great potential to repair the stream quality.

Outlook

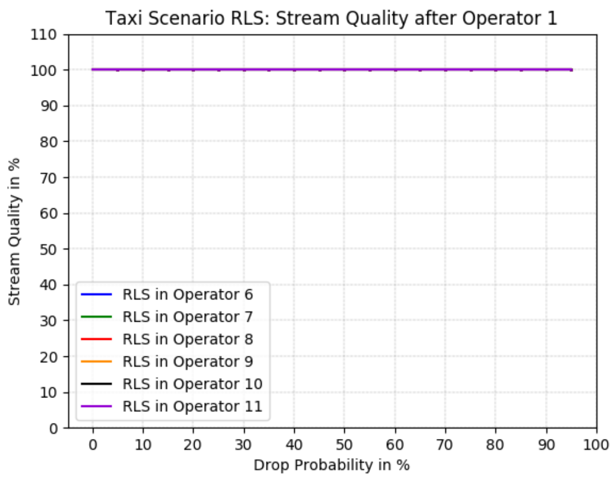
In this thesis, we have applied load shedding at one operator at a time. The next step would be to test, how load shedding at more than one operator at a time would affect the stream quality. It would be interesting to see, if the guidelines for load shedding in a CEP system, that we have found in this thesis, still apply in that case. Furthermore, maybe new rules could be found, that optimize the reduction in latency and minimize the loss of stream quality.

The operator types used in the experiment could also be tested in different typologies and with different data sets. This could be used to confirm the results of this thesis.

8 Attachment

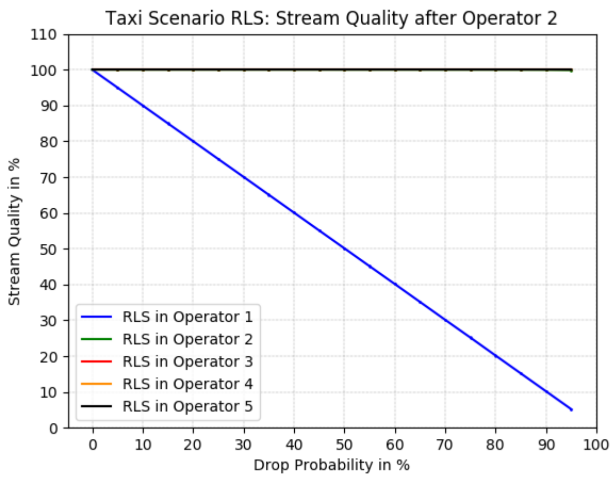


(a)

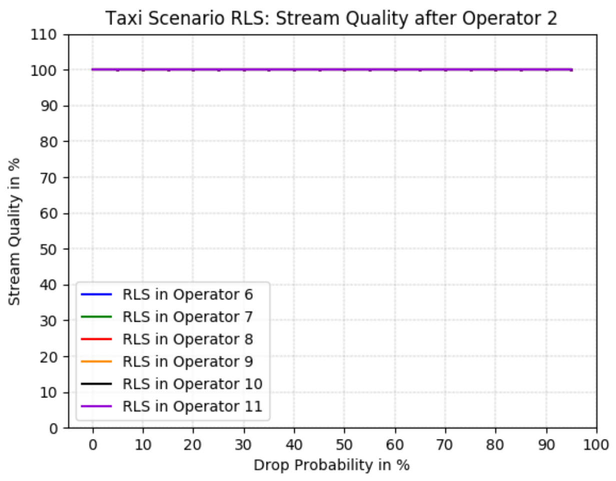


(b)

Figure 8.1: Taxi Scenario: This figures depict the stream quality after operator 1.

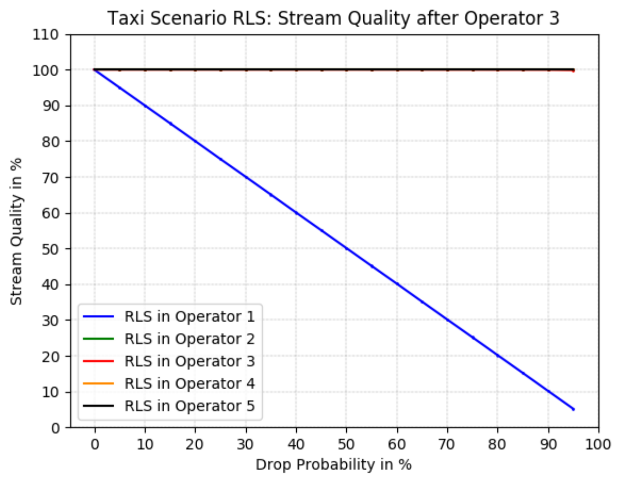


(a)

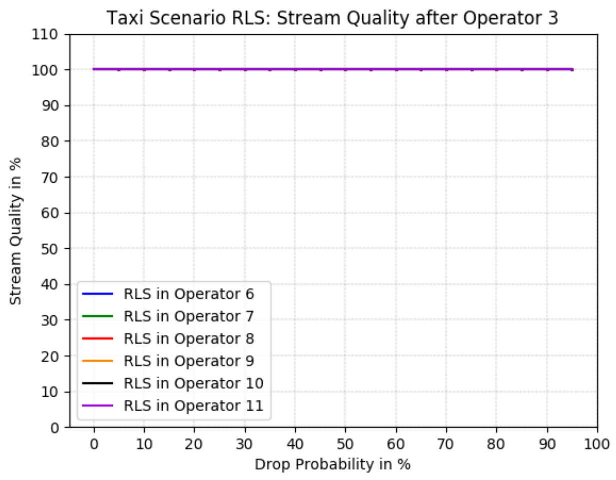


(b)

Figure 8.2: Taxi Scenario: This figures depict the stream quality after operator 2.

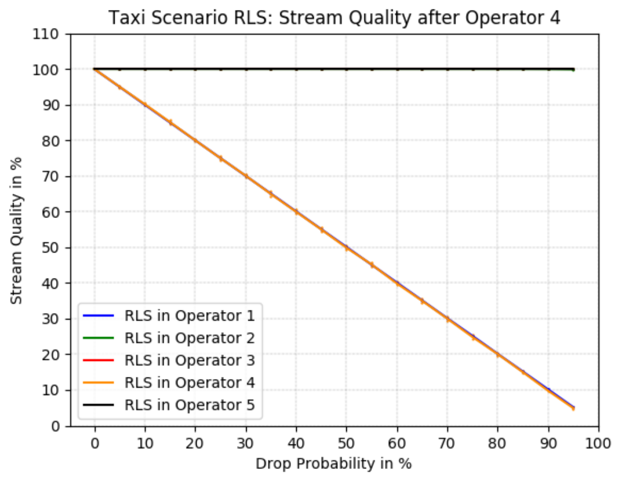


(a)

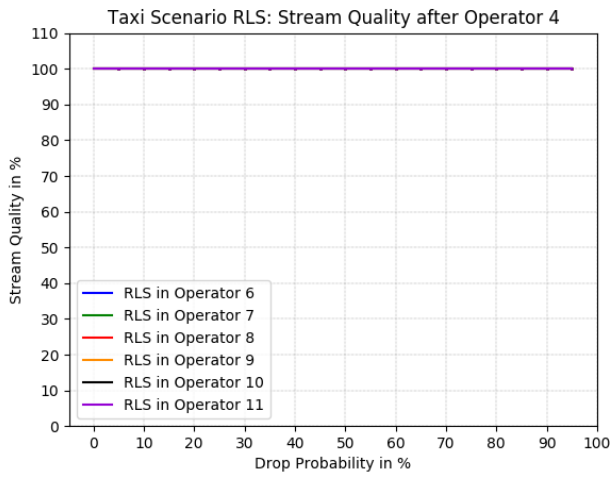


(b)

Figure 8.3: Taxi Scenario: This figures depict the stream quality after operator 3.

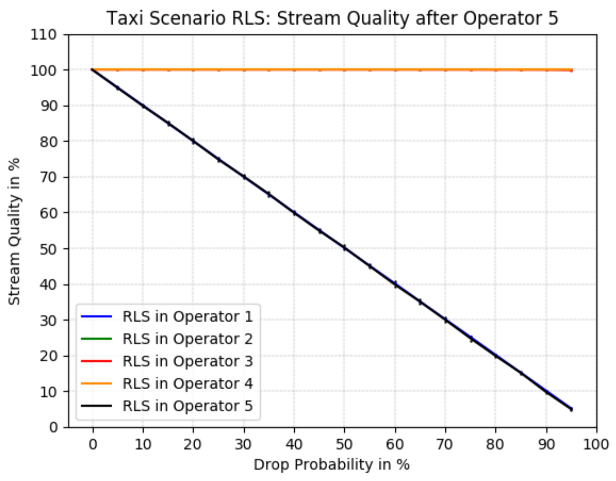


(a)

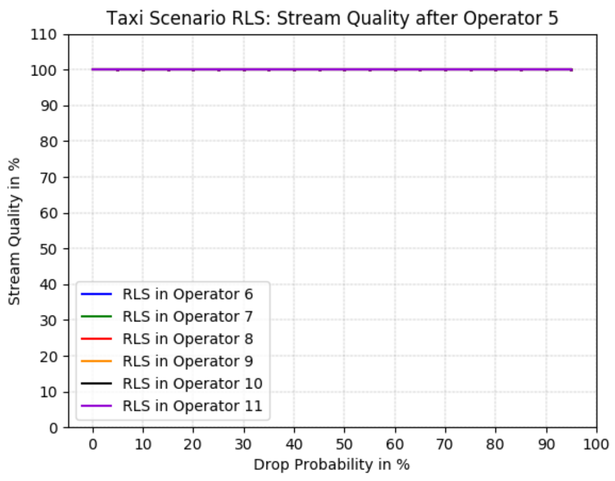


(b)

Figure 8.4: Taxi Scenario: This figures depict the stream quality after operator 4.

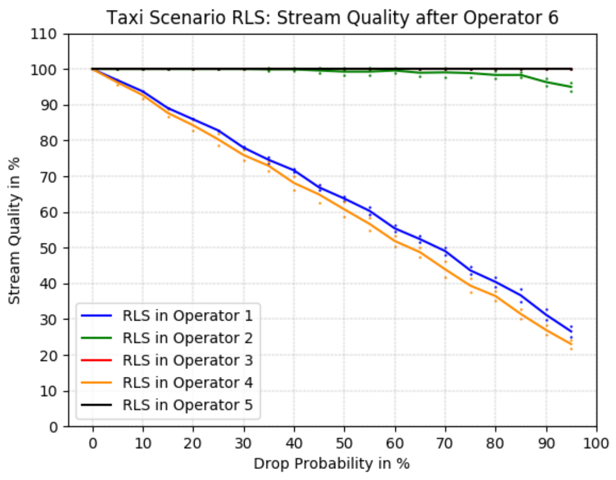


(a)

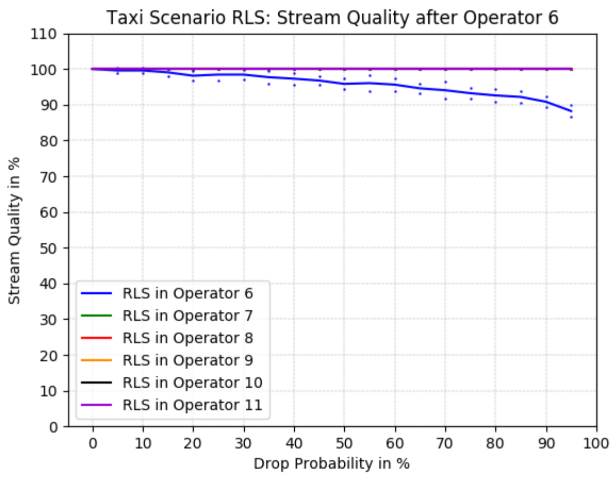


(b)

Figure 8.5: Taxi Scenario: This figures depict the stream quality after operator 5.

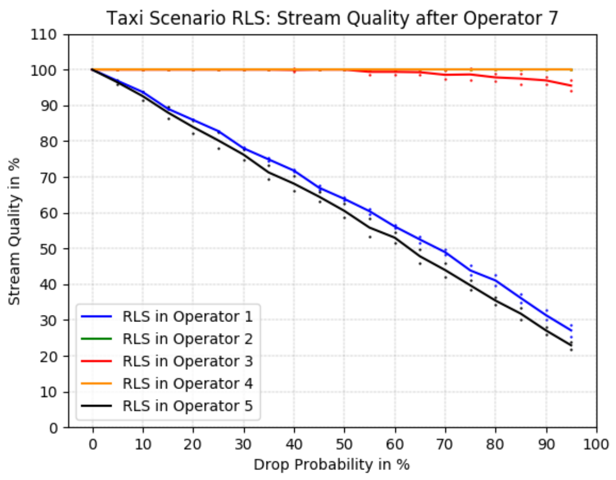


(a)

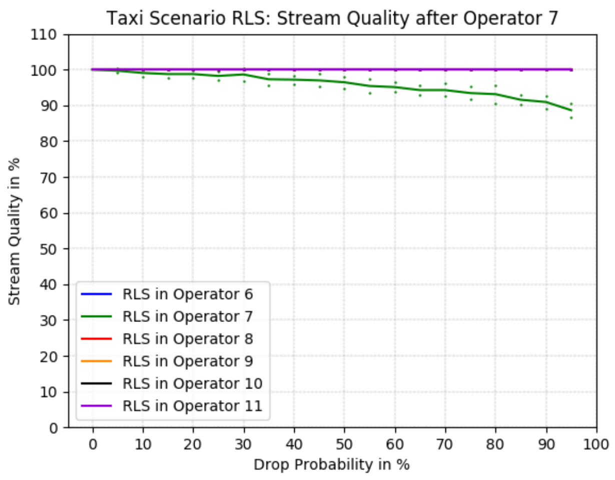


(b)

Figure 8.6: Taxi Scenario: This figures depict the stream quality after operator 6.

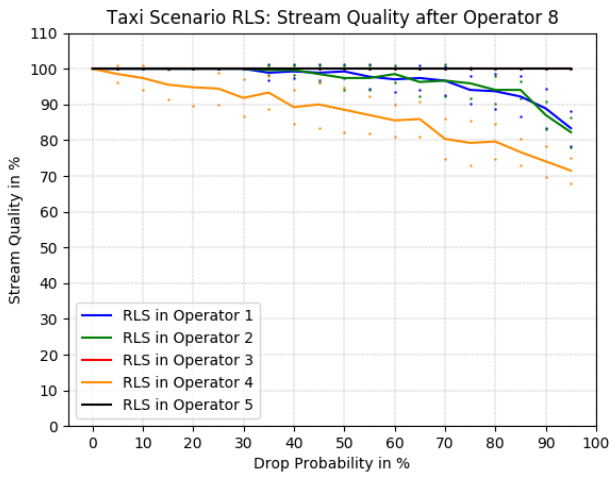


(a)

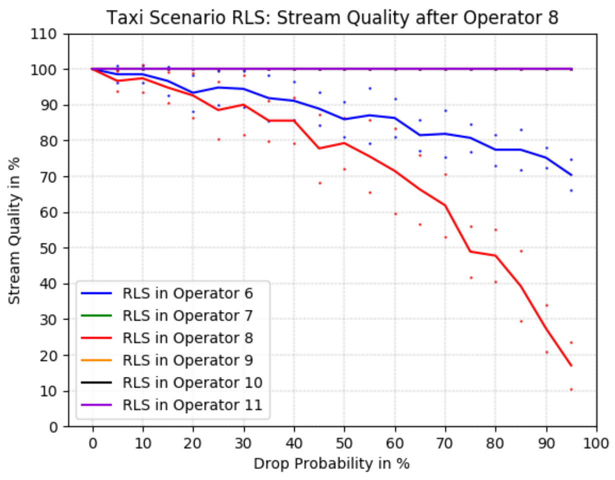


(b)

Figure 8.7: Taxi Scenario: This figures depict the stream quality after operator 7.

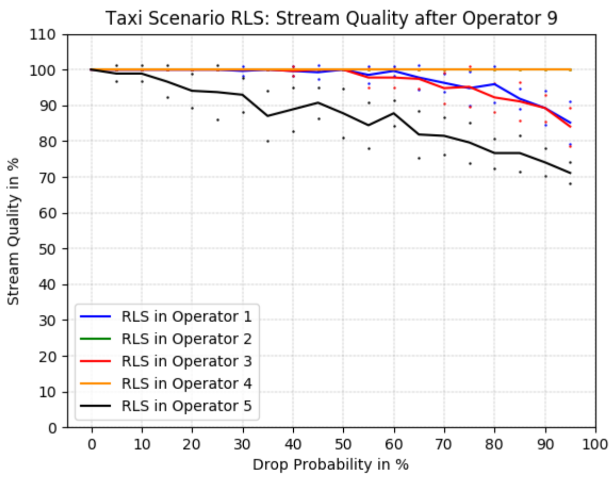


(a)

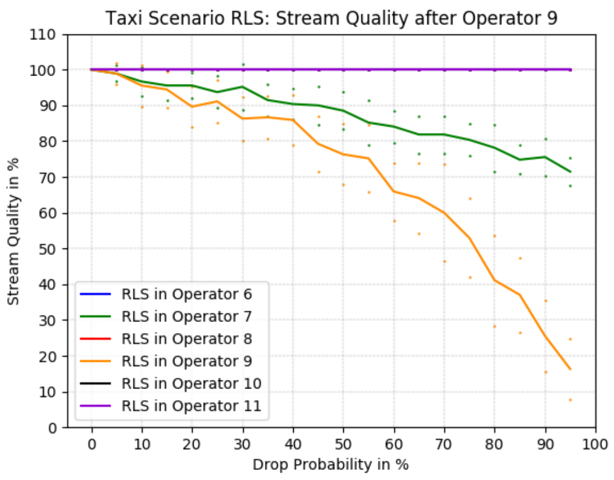


(b)

Figure 8.8: Taxi Scenario: This figures depict the stream quality after operator 8.

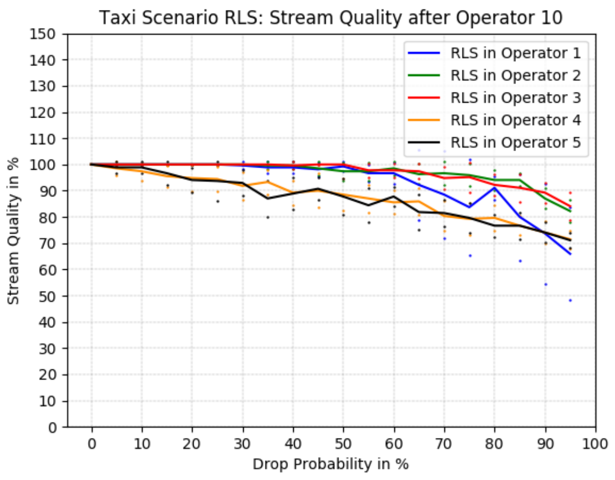


(a)

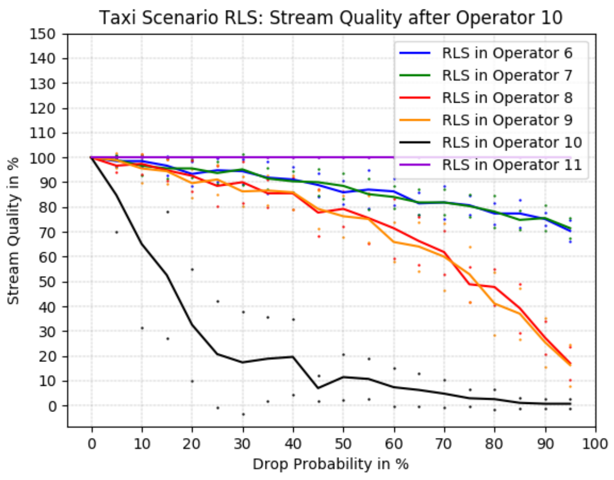


(b)

Figure 8.9: Taxi Scenario: This figures depict the stream quality after operator 9.

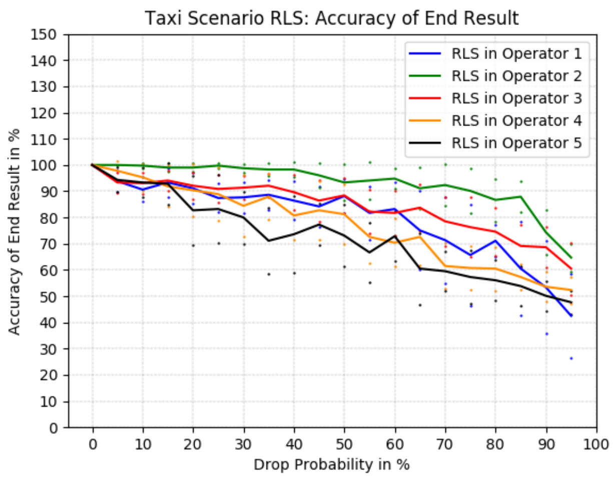


(a)

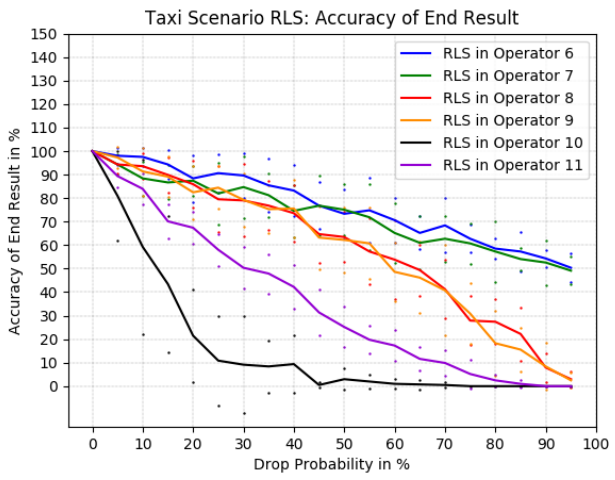


(b)

Figure 8.10: Taxi Scenario: This figures depict the stream quality after operator 10.



(a)



(b)

Figure 8.11: Taxi Scenario: This figures depict the accuracy of the end result.

Bibliography

[HBN13] Y. He, S. Barman, J. F. Naughton. “On Load Shedding in Complex Event Processing.” In: *CoRR* abs/1312.4283 (2013). arXiv: 1312.4283. URL: http://arxiv.org/abs/1312.4283 (cit. on p. 19).

[KFSP16] E. Kalyvianaki, M. Fiscato, T. Salonidis, P. Pietzuch. “THEMIS: Fairness in Federated Stream Processing Under Overload.” In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: ACM, 2016, pp. 541–553. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2882943. URL: http://doi.acm.org/10.1145/2882903.2882943 (cit. on p. 21).

[KLC18] N. R. Katsipoulakis, A. Labrinidis, P. K. Chrysanthis. “Concept-Driven Load Shedding: Reducing Size and Error of Voluminous and Variable Data Streams.” In: *2018 IEEE International Conference on Big Data (Big Data)*. Dec. 2018, pp. 418–427. DOI: 10.1109/BigData.2018.8622265 (cit. on p. 21).

[Raj16] G. B. Raj Jain Arjan Durresi. *Throughput Fairness Index: An Explaination*. 2016. URL: https://www.cse.wustl.edu/~jain/atmf/ftp/af_fair.pdf (cit. on p. 22).

[RBR19] H. Röger, S. Bhowmik, K. Rothermel. “Combining It All: Cost Minimal and Low-latency Stream Processing Across Distributed Heterogeneous Infrastructures.” In: *Proceedings of the 20th International Middleware Conference*. Middleware ’19. Davis, CA, USA: ACM, 2019, pp. 255–267. ISBN: 978-1-4503-7009-7. DOI: 10.1145/3361525.3361551. URL: http://doi.acm.org/10.1145/3361525.3361551 (cit. on p. 11).

[RM19] H. Röger, R. Mayer. “A Comprehensive Survey on Parallelization and Elasticity in Stream Processing.” In: *ACM Comput. Surv.* 52.2 (Apr. 2019), 36:1–36:37. ISSN: 0360-0300. DOI: 10.1145/3303849. URL: http://doi.acm.org/10.1145/3303849 (cit. on p. 11).

[SBFR19] A. Slo, S. Bhowmik, A. Flaig, K. Rothermel. “pSPICE: Partial Match Shedding for Complex Event Processing.” Englisch. In: *Proceedings of the 2019 IEEE International Conference on Big Data (BigData ’19); Los Angeles, CA, USA 9 - 12 December, 2019*. IEEE, Dezember 2019, pp. 1–11. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2019-26&engl=0 (cit. on pp. 12, 25).

[SBR19] A. Slo, S. Bhowmik, K. Rothermel. "eSPICE: Probabilistic Load Shedding from Input Event Streams in Complex Event Processing." In: *Proceedings of the 20th International Middleware Conference*. Middleware '19. Davis, CA, USA: ACM, 2019, pp. 215–227. ISBN: 978-1-4503-7009-7. DOI: 10.1145/3361525.3361548. URL: http://doi.acm.org/10.1145/3361525.3361548 (cit. on pp. 12, 20, 25).

[Tat02] N. Tatbul. "QoS-Driven Load Shedding on Data Streams." In: *XML-Based Data Management and Multimedia Engineering — EDBT 2002 Workshops*. Ed. by A. B. Chaudhri, R. Unland, C. Djeraba, W. Lindner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 566–576. ISBN: 978-3-540-36128-2 (cit. on p. 23).

[Tat07] E. N. Tatbul. "Load Shedding Techniques for Data Stream Management Systems." AAI3272068. PhD thesis. Providence, RI, USA, 2007. ISBN: 978-0-549-11942-5 (cit. on p. 23).

[TÇZ+03] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, M. Stonebraker. "Load Shedding in a Data Stream Manager." In: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*. VLDB '03. Berlin, Germany: VLDB Endowment, 2003, pp. 309–320. ISBN: 0-12-722442-4. URL: http://dl.acm.org/citation.cfm?id=1315451.1315479 (cit. on p. 23).

[TÇZ07] N. Tatbul, U. Çetintemel, S. Zdonik. "Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing." In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB '07. Vienna, Austria: VLDB Endowment, 2007, pp. 159–170. ISBN: 978-1-59593-649-3. URL: http://dl.acm.org/citation.cfm?id=1325851.1325873 (cit. on p. 22).

[Twi19] TwitterInc. *Introduction to Tweet JSON*. 2019. URL: https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/intro-to-tweet-json (cit. on p. 29).

[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 '06. Seoul, Korea: VLDB Endowment, 2006, pp. 799–810. URL: http://dl.acm.org/citation.cfm?id=1182635.1164196 (cit. on p. 23).

[Who14] C. Whong. *FOILing NYC's Taxi Trip Data*. 2014. URL: https://chriswhong.com/open-data/foil_nyc_taxi/ (cit. on p. 30).

[ZXLT10] H. (Zhao, C. H. Xia, Z. Liu, D. Towsley. "A Unified Modeling Framework for Distributed Resource Allocation of General Fork and Join Processing Networks." In: *SIGMETRICS Perform. Eval. Rev.* 38.1 (June 2010), pp. 299–310. ISSN: 0163-5999. DOI: 10.1145/1811099.1811073. URL: http://doi.acm.org/10.1145/1811099.1811073 (cit. on p. 22).

All links were last followed on December 17, 2019.

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