University of Zurich[UZH]

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Implementation of Algorithms and Heuristics to Optimize a Data Generation and Preparation System for Credit Card Fraud Detection

Dominik Bünzli
Zürich, Switzerland
Student ID: 14-707-111

Supervisor: Eder Scheid, Muriel Franco, Matthias Schwalbe
(Viseca Card Services SA)
Date of Submission: April 15, 2021

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Zusammenfassung

Durch die fortschreitende Digitalisierung, das kontinuierliche Wachstum des Online-Handels sowie der zunehmenden Akzeptanz von mobilen und kontaktlosen Bezahlmöglichkeiten werden Transaktionen immer häufiger mit Kredit- und Debitkarten abgewickelt. Dieses Wachstum hat jedoch auch seine Schattenseiten. Wo viel Geld im Spiel ist, sind auch Betrüger nicht fern. In der Studie von [1] wird erwähnt, dass der weltweite jährliche Verlust durch Betrug von 28 Milliarden Dollar innerhalb von 5 Jahren auf über 35 Milliarden ansteigen wird. Um diesem Trend entgegenzuwirken, investieren Kreditkartenherausgeber vermehrt in die Erkennung und Prävention von Kreditkartenbetrug. In der Industrie sind aktuell hauptsächlich regelbasierte Systeme in Betrieb [2]. Der Trend zeigt aber stark in Richtung von Machine Learning Modellen. Diese bieten eine vielzahl an Vorteilen gegenüber statischen Regelwerken. Ein Machine Learning Model benötigt jedoch eine umfassende und anspruchsvolle Datenaufbereitung. Dies wird erschwert durch den Umstand, dass jede Kartenbenutzung innerhalb weniger Millisekunden bewertet werden muss. Traditionelle Systeme stossen bei dieser Aufbereitungsarbeit an ihre Grenzen. Abhilfe könnte hierbei die Streaming Technologie schaffen. Diese Arbeit beschäftigt sich entsprechend mit dem Design und der Implementierung von Algorithmen und Heuristiken zur Optimierung eines Streaming Systems zur Erkennung und Prävention von Kreditkartenbetrug. Es werden verschiedene Optimierungsstrategien eingeführt um die Performance eines vorliegenden Systems zu erhöhen. Es findet dabei eine Unterteilung in State, Input/Output (I/O) und algorithmische Optimierung statt. Nach bestem Wissen des Authors werden mit den kaskadierenden Fensteraggregationen sowie den kontinuierlich gleitenden Fenstern zwei neue Optimierungsmöglichkeiten eingeführt. Durch die implementierten Anpassungen konnte der Durchsatz des Streamingjobs von wenigen 100 auf 40'000 Events pro Sekunde verbessert werden. Ein verstopfen des Jobs oder aufblähen des Status findet zudem nicht mehr statt. Durch die Reduzierung der benötigten Operatoren konnte zudem die Latenz minimiert werden.

Abstract

Due to advancing digitalization, the continuous growth of e-commerce and the increasing acceptance of mobile and contactless payment methods, an increasing number of purchases are being made with credit and debit cards [3]. However, this growth has its downsides. Where there is a lot of money at stake, fraudsters are not far away. The study conducted by [1] mentions that the global annual loss of 28 billion due to fraud will rise to over 35 billion within 5 years. To counter this trend, credit card issuers are investing heavily in credit card fraud detection and prevention systems. In the industry, mainly rule-based systems [2] are currently in operation. There is a strong trend towards machine learning models as they offer a number of advantages over static rules. However, a machine learning model requires extensive and sophisticated data preparation. This is further complicated by the fact that each card usage has to be evaluated within a few milliseconds. Traditional systems reach their limits in this preparation task. Streaming technology could provide a remedy. Accordingly, this thesis deals with the design and implementation of algorithms and heuristics to optimize a streaming system for credit card fraud detection and prevention. Different optimization strategies are introduced to increase the performance of a given system. Thereby, a subdivision into operational state, input/output (I/O) and algorithmic optimizations takes place. To the best of the author's knowledge, the cascading window aggregation and the continuous sliding window algorithm are introduced as two new optimization approaches. With the implemented adjustments, the throughput of the streaming job could be improved from a few 100 to 40'000 events per second. Furthermore, the job is no longer clogged and the status does not longer get bloated. By reducing the number of operators required, the latency could also be significantly reduced.

Acknowledgments

I would like to thank many people for the opportunity to do this very interesting and educational work. First of all, I would like to thank the Viseca Card Services SA, namely the Head Business Analytics Services - Herbert Bucheli, Head Risk Analytics - Dr. Viton Vitanis, Head Datawarehouse Development - Pascal Simon for all their input and support. I would also like to thank Prof. Dr. Stiller - Head of the Communication Systems Group at the University of Zurich as well as Eder Scheid for the opportunity to write this highly interesting paper under their guidance and supervision. Their continuous support and feedback helped a lot for the successfull completion of this thesis. My greatest thanks goes to Matthias Schwalbe. I am immensely grateful for the knowledge he shared with me. Matthias has always taken time for my questions and has helped me with any uncertainties with his advice. His way of passing on knowledge is really impressive and testifies to a great personality. It was a pleasure to work with him during this time. I would also like to thank my family, who supported me during my studies. Without them, the whole journey would not have been possible. Many thanks for that.

Contents

Chapter 1

Introduction

According to [1], the current global loss due to fraud amounts to 28 billion in 2018 and is predicted to rise to 35 billion within 5 years. It is even forecasted to reach up to 40 billion within 10 years. A study by [5] explains that in the US alone, the number of fraud reports jumped from approximately 17'000 in 2015 Q1 to 45'000 in 2020 Q1. This equates to a total increase of 164%. As stated by [3], approximately 530 million transactions have been carried out with Swiss credit cards in 2020. Compared to 2005, the number of transactions has increased fivefold. Furthermore, the revenue has doubled in the process to 47 billion, and the use of mobile and contactless payments is driving growth in credit card usage even more. When comparing the global loss to the revenue in Switzerland, it becomes obvious why credit card fraud detection and prevention is such an important topic, and the need for efficient fraud prevention systems is evident. Nevertheless, the implementation and operation of such systems is a complex and demanding task. The algorithms used are often computationally intensive and require a large amount of data or human knowledge to work efficiently. The whole process is complicated by the fact that when a card is used, a decision with a financial impact has to be made within a few milliseconds, whether the use is legitimate or not. Large amounts of data in machine learning models are increasingly being used as a basis for this decision-making. The processing of such data volumes is in itself a great challenge which is exacerbated even more so within the given time limits. Therefore, this thesis deals with the design and implementation of algorithms and heuristics for the optimization of a feature generation system for fraud detection and prevention. The next Section is used to provide an in-depth introduction to the problem at hand.

1.1 Problem Description[1]

As outlined in [6], credit card issuers put considerable effort into preventing fraud in order to minimize their losses as well as the inconvenience caused to the affected cardholders. Systems regularly used in practice are mostly based on rules [2], which require frequent

[1]This thesis is a continuation of the topic addressed in the master's project [6]. Thus, the problem description was adopted.

updates and constant maintenance from human operators. In recent years, issuers have been investing in developing machine learning systems [1] that adapt to changing fraud patterns and assist fraud analysts to identify fraud faster. The corresponding machine learning models are often complex and attribute their success to both straightforward and sophisticated features that need to be calculated for each incoming transaction. For example, such features include:

- The time difference between transactions,
- The average amount spent in the past 30 days,
- The ratio between current amount and previous amount
- The number of distinct fraud cases (cards that had fraud) at an establishments point of sale.

In contrast, sophisticated features are the ratio of the number of cards with at least one fraud authorization at a specific establishment to the number of cards with at least one non-fraud authorization at the same establishment in the last 30 days. Into further consideration falls the ratio of the skewness of the amount spent after the current transaction to the skewness before the current transaction in the last 30 days. The current machine learning model implemented for the Fraud Scoring algorithm at Viseca [7], is based on a data preparation system performed on an SQL server once every hour. The decision to rely on SQL was made based on the existing infrastructure and has several significant implications:

- Every time a subset of the features is calculated, regardless of the complexity of the computation, the SQL engine has to access the same records several times. In addition, most of the SQL queries employed on the calculation have a complexity of $O(M * N)$, where M is the number of days the training table needs to be calculated for and N is the number of days that are considered as history for the features.
- The data preparation cannot be performed incrementally due to the need of creating and maintaining the data state. This requires significant investments in time and resources. Moreover, due to the sheer amount of data, maintaining this state results in a fragile system, which must be avoided.
- The fact that it is not possible to prepare the data and score transactions in (near) real-time significantly reduces the benefit of using a machine learning model to prevent fraud.

However, these disadvantages of the current configuration can be mitigated if the data preparation is performed based on a stateful streaming model. In such a model, each of these features being calculated (even the ones based on a "distinct" operation) can be updated incrementally with each incoming transaction; therefore, reducing the latency of the system and increasing its potential to prevent fraud. If potentially fraudulent transactions in real-time are rejected, it is expected that applying such a streaming model could prevent almost twice as much fraud, such as when the scores are calculated every hour and only the subsequent fraudulent transactions are blocked.

1.2 Task Description

The main goal of this thesis is to study and optimize an existing data preparation system for fraud prevention (*cf.* Fig. 1.1) in terms of latency, jitter, and throughput. It is important to note that for legal reasons, no work on customer data will be performed for this master thesis. Instead, the test data generator developed in a previous work [6] will be used. This data generator allows to model general interactions and statistical correlations of production data and does not require the use of real-world data, which avoids privacy issues. The development of the existing fraud prevention streaming system initially started in the master project [6] and was refined before the start of this thesis. It serves as a baseline to compare the different optimization strategies introduced herein.

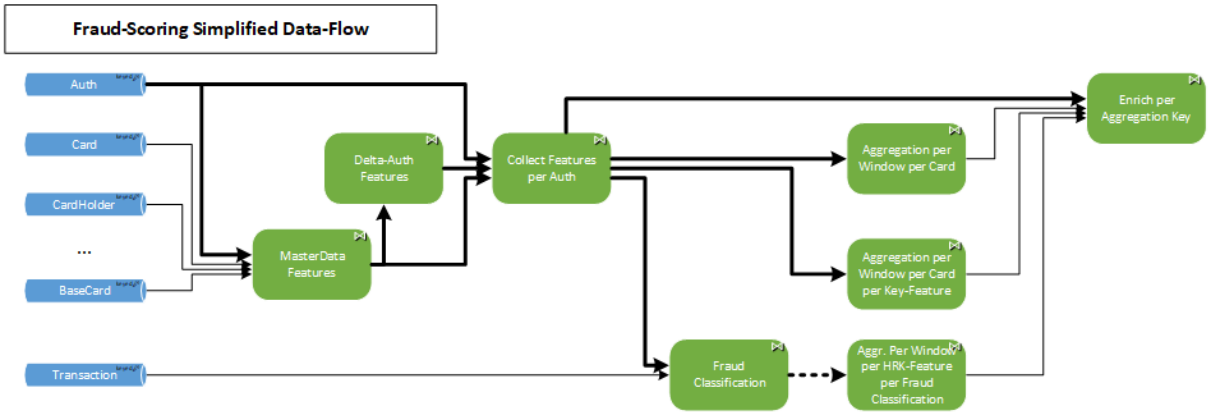

Figure 1.1: Dataflow Diagram of the Existing Data Preparation System

The system depicted in Fig. 1.1 is used to calculate several hundred features from a continuous stream of authorizations and transactions. Subsequently, they are used in a pre-existing machine learning model to detect fraudulent card usages. Apache Flink [8] is used as the framework for the data preparation, which has inherent constraints that set the boundaries for this thesis. The following two points are considered important in this context: (i) **State Backends** and (ii) **State Primitives**. The first is that Flink only offers Java-Heap or RocksDB [9] based state backends for storing intermediate state, where only one state backend can be used at the same time. Both backends have a similar, yet different semantic and timing behaviour. The second is regarding to Flink's state management, which only offers the following so-called state primitives: *single typed values*, *list of typed values*, and *Maps from key type to value type*. Therefore, the state backends and state primitives restrict the choice of pre-existing aggregation algorithms that can be used.

The thesis is split into four different stages. In a first stage, a literature research has to be performed in order to list credit card fraud detection and prevention approaches. Furthermore, algorithms and works that optimize streaming frameworks and adhere to the aforementioned constraints have to be covered. It should be noted that algorithms to optimize streaming frameworks exist (*e.g.,* [10, 11]), but the solutions are not directly applicable regarding the given constraints and data characteristics.

In a second stage, the current implementation of the system has to be analyzed in order to obtain reference values for the system's performance (*e.g.,* latency, jitter, and throughput). A way has to be found to measure the systems performance without solely relying on pre-defined metrics. For the evaluation, the same configurations as for the baseline implementation should be used. This is necessary to gather reliable results.

In a third stage, the design and implementation of algorithms to optimize the current performance of the data preparation and feature generation system are addressed. These optimization strategies should consider the state of the art methods listed in the first stage of this thesis. Furthermore, not only optimization algorithms, but heuristics have to be proposed and implemented considering the existing constraints.

Finally, in the the last stage the evaluation of the optimization strategies with contrast to the performance of the baseline implementation have to be carried out. The effects of the different algorithms in terms of latency, jitter and throughput will be discussed as to whether the values are appropriate for real-world use. The results will be discussed concerning the achievement of the thesis goals.

1.3 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 describes the necessary technologies and terms employed in this thesis, offering a deepened insight into streaming systems and the used stream processing framework Apache Flink. Afterwards, Chapter 3 discusses the state of the art methodologies on related works and concepts. Chapter 4 then details the design of the optimization strategies and their algorithms. It is complemented by Chapter 5, where the implementation of the optimization strategies are discussed. Next, the evaluation of the implementation compared to the existing data preparation system for fraud prevention is presented in Chapter 6. Finally, Chapter 7 concludes this thesis and presents future work.

Chapter 2

Background

This Chapter provides background information on the present work. First, an insight into credit card fraud and its prevention is given. Second, the introduction of data streaming, stream processing as well as Apache Flink follows. Finally, the baseline implementation of the Fraud Scoring feature calculation pipeline is explained and a look is taken at test data generation.

2.1 Credit Card Fraud

As stated in the introduction, credit card fraud is an increasingly relevant topic in a further digitizing world. In the Swiss criminal code, the term fraud is defined in article 146 as follows: *"Any person who with a view to securing an unlawful gain for himself or another wilfully induces an erroneous belief in another person by false pretences or concealment of the truth, or wilfully reinforces an erroneous belief, and thus causes that person to act to the prejudice of his/her or another financial interests."* This definition is further deepened by article 148, which specifically deals with cheque and credit card fraud. The following lines contain information that underlines the need for fraud prevention. *"Any person who with a view to obtaining services of a financial value and although incapable of making or unwilling to make payment uses a cheque card or credit card or similar means of payment that has been entrusted to him by the issuer thereof and thus causes loss to the issuer, is liable, provided the issuer and the contracting enterprise have taken reasonable measures in order to prevent the abuse of the card."* In other words, credit card fraud is defined as unrightful use of credit card by another entity than the owner. This use can be either physical (*e.g.,* theft or skimming) or online.

Of particular interest is the passage stating that the issuer and the contracting enterprise have taken sufficient measures to counteract such a fraud. The card issuer is responsible for card-present-authorizations (*i.e.,* card is physically used at the payment terminal) as well as card-not-present-authorizations usages (*i.e.,* card is used online) with activated two-factor authentication. However, if the card is used physically and a fraudster has obtained the PIN through a breach of due diligence on the part of the customer, then

the customer is liable. In case of an online transaction, where the merchant does not use two-factor authentication, then the merchant is liable.

While there is some understanding of how a breach of due diligence can occur, it is somewhat incomprehensible why a merchant would not rely on two-factor authentication. The reason for this behavior is described in [1], where the introduction of the two-factored authentication leads to a loss of revenue of up to 19%. This is due to the fact that customers sometimes forget their password or are distracted by other things such as a crying baby or a ringing doorbell. Thus, they do not complete the purchase. Other reasons mentioned in [1] are bad user interfaces, which give the users the feeling of being led to an insecure website. In addition, an overall poor user experience and the annoyance of additional authentication are mentioned. Since merchants are interested in more sales, it is worthwhile for larger merchants to take the risk and invest more in their own fraud prevention systems. However, the main burden of fraud prevention lies with the issuer. This explains the increasing interest in this area and why investments are being made in new technologies for this field.

Subsequently, the systems currently in use and their potential successors are examined in more detail in the following Section 2.2.

2.2 Fraud Prevention Systems in Industry

If one searches the Internet for fraud prevention and detection systems, most of the systems listed refer to machine learning models [12, 13, 14, 15]. This is understandable due to the long existing efficacy of such systems. It is also a selling point to keep up with the times and offer systems that use up-to-date technologies. However, the current state-of-art method in the industry is different. Many of the fraud prevention systems currently in use have grown historically and the operationalization of new machine learning models is, as so often, a difficult task. In addition, there is a time criticality of the response time. As a customer should not wait long for the response of the fraud prevention system, a response within a few milliseconds is prescribed by the schemes.

As stated by [2], the most common fraud prevention and detection systems today are still rule-based engines. Such systems consist of a set of manually defined rules, which are applied to the card usage. This means that each time a card is used, the rules are applied and a decision is made whether to allow or deny the card usage. If a customer uses a card in a restaurant with an amount that is within the normal range for this merchant, the card will most likely be accepted. However, if for instance three transactions are made with the same card at the same merchant within a short period of time, possibly with a conspicuous value, they will be rejected.

It is apparent that a large number of these rules must exist in order to obtain a high level of fraud coverage. This results in a large amount of personnel and manual work. The rules need to be maintained, adjusted and new patterns discovered. Additionally, the employees working in the fraud department are often supported by data scientists

with reports and other insights into the data. Thus, increasing the overall costs for fraud prevention even more.

Due to the disadvantages mentioned, the use of machine learning models is becoming increasingly popular. The topic of credit card fraud prevention and machine learning has been a subject in science for a long time as well as in numerous papers [12, 13, 14, 15]. These relevant works are reviewed in more detail in Chapter 3. Also, potential parallels to this work, especially the data preparation and feature calculation, will be highlighted.

2.2.1 Fraud Prevention Prototype

One project of the cooperating company of this thesis, which dealt with the further development of fraud prevention implemented a machine learning model with great effort. Their internal studies have shown that the new system can detect up to 200% more fraudulent card usages than the existing rule set. The new Fraud Scoring system assigns a score to incoming authorizations indicating the probability of fraud. Although this approach is promising, many challenges arise. The data preparation for the training of the model currently requires four days for the data volume of one month. In addition, the effective scoring can only be performed every hour where a large amount of data has to be included for this. Thus, it is in contrast to the maximum required response time of a few milliseconds. The data streaming technology introduced in Section 2.3 could help to mitigate this issue. However, the requirements for the system to be implemented are not trivial. Currently, the machine learning model is based on a feature set with approximately 1485 attributes. This set is depicted in Fig. 2.1 and further broken down into the following three scenarios:

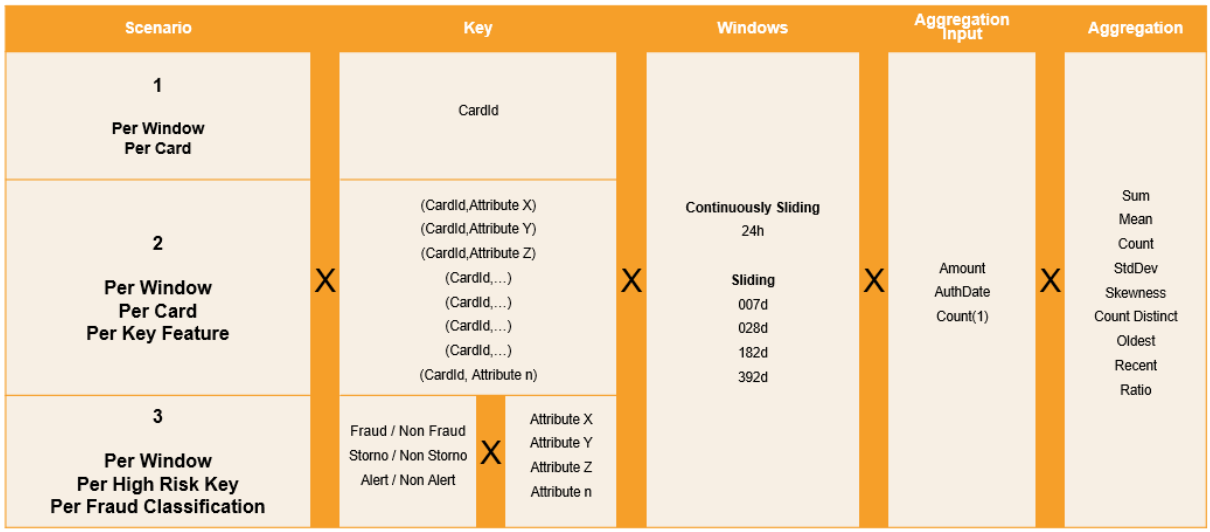

Figure 2.1: Required Features for the Machine Learning Model

- **Scenario 1:** Aggregations are performed per card. For example, how much a card has spent in the last 24 hours as well as the past 7, 28 or 392 days.

- **Scenario 2:** Aggregations are performed per card and another key attribute. For example, it should be calculated how often a card was used for a certain Merchant Category Code (MCC) or how much a card has spent in a certain region. The keys consist of tuples, which are depicted in the *key* column of the second scenario in Fig. 2.1. Currently, it contains eight combinations of the card id and another attribute. Later in this thesis, these different modes of aggregation will be called *key domains*.
- **Scenario 3:** Aggregations are performed for keys which are prone to have a high impact on the model, independent of the card id. These keys are defined by data scientists as **High Risk Keys** (HRK), where an example would be the country of the card usage. Interesting for the HRKs, is the further classification into *fraud / non fraud*, *storno / non storno* and *alert / non alert*. For example, this allows to differentiate how many of the card usages in a country or with a merchant were fraudulent or not.

The three scenarios are further divided into 5 different lengths of interest. These time frames will, from now on, be related to as windows. The windows are presented in more detail in Section 2.4.5. In each case, the windows should be daily sliding windows of 7, 28, 182 and 392 days length. This means that every day there should be a window that contains the data from the corresponding time span. In addition, a continuously sliding 24-hour window has to be implemented. This window contains all events of the last 24 hours at any point in time.

Finally, the nine aggregations visualized in Fig. 2.1 are performed on the windows. The feature set is composed as presented in Table 2.1 and amounts to a total of 1485 features which have to be calculated concurrently.

	Key Domains	**Window Types**	**Aggregation Functions**
Scenario 1	1	5	9
Scenario 2	8	5	9
Scenario 3	(6 Classifications * 4 Keys)	5	9

Table 2.1: Number of Features to be Calculated

The goal of the envisioned system is to reduce the time needed for calculations from one hour to just a few milliseconds per event. With traditional, relational means, this cannot be achieved with justifiable effort. However, the streaming technology is promising to solve this issue and subsequently, is introduced in Section 2.3.

2.3 Data Streaming and Stream Processing

Data streaming, or event stream processing, is described by [16] as the continuous flow and processing of data from various sources. These sources send the data simultaneously and

in small sizes. Using stream processing technology, streams can be processed, stored and analyzed in real time. Streaming data covers a wide range of applications. For example, it can be various logs, e-commerce purchases, in-game activities, social media information, financial information, or in the case of this thesis, credit card transactions.

[17] provides a simple analogy to understand the term streaming. Analogous to water flowing through a river, different streams come from different sources, at different speeds and in different volumes. However, they eventually flow into the same continuous stream.

The data can be divided into bounded or unbounded streams according to [8], which will be explained in the next subsection.

2.3.1 Bounded and Unbounded Data

Unbounded Data has a start, but no defined end. Thus, there is no finite data set over which an operation (*e.g.,* aggregation) could be performed. Therefore, events must be processed continuously after they enter the system. By definition, it is not possible to wait for all data to arrive to get a final result, as at no point in time will all the data be available. In order to make statements about the completeness of the results of such streams, these events often have to be processed in a specific order, as mentioned by [8]. For example, this order can be the time at which the event happened. An example of such streams is depicted in Fig. 2.2.

Bounded Data has a defined start and end as visualized in Fig. 2.2. These streams are processed by getting all the data of a set before doing any calculations. An ordered reading is not necessary, because the data in a bounded data set can always be sorted. The processing of bounded streams is also called batch processing.

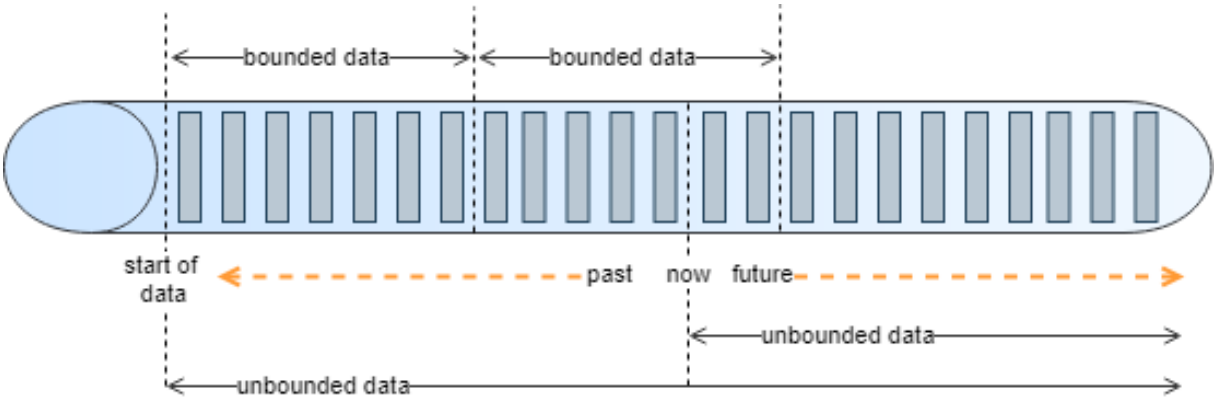

Figure 2.2: Visualization of Bounded and Unbounded Streams [8]

Bounded and unbounded data requires different approaches for their processing and have different use cases. [16] provides an interesting table (*cf.* Table 2.2), which compares the individual characteristics.

	Bounded Data	**Unbounded Data**
Data Scope	Queries Over All Data in the Dataset	Queries Over a Window
Data Size	Large Batches	Single Events or Micro Batches
Performance	Minutes to Hours	Seconds or Milliseconds

Table 2.2: Difference Between Bounded and Unbounded Streams

In summary, this represents the difference between traditional batch processing and real-time, continuous processing of data. The following distinction can be made.

- In **streaming**, the arrival of new events triggers calculations independent of the consumer of the results.
- In **batching**, the consumer triggers computation when results are needed and thereby selects the end of the bounded data. The calculation can be repeated with new intervals.

In the progressively digitalized world, data is delivered at an ever-increasing rate, in different formats, sizes, and from different sources. Consequently, existing data processing solutions are thus starting to reach their limits. In most cases, it is no longer sufficient to wait until all of the data is available and process it as a group of transactions. Rather, it is becoming increasingly important to analyze and evaluate data in real time to gain insight into the business as quickly as possible. Also, users are no longer accustomed to waiting minutes or days for a report, as the data must first be aggregated in the desired batch. Thus, streaming seems to be a viable candidate for various use cases that rely on real-time data. The next subsection will now take a look at the benefits and challenges arising from data streaming.

2.3.2 Benefits and Challenges

As stated by [17], two main functionalities are required for applications that work with streams: storage and processing. Storage, on the one hand, must be able to efficiently manage a large amount of events to ensure sequentiality and consistency. Processing, on the other hand, must be able to access the storage, consume the data, analyze it and perform calculations on it. These requirements bring additional challenges to such a system.

- **Scalability**: For example, when the number sensors added to a system increase, or more and more online transactions are made due to digitalization, the amount of data increases exponentially. Another scalability example pertains to the generation of large data logs. In case of a fault, these data logs are delivered at a rate of gigabytes instead of kilobytes per second. If the consuming streaming application is not built to scale, it becomes slower and slower until it is no longer usable.

- **Ordering**: In most cases, ordering is a non-trivial problem problem, resulting from different timestamps and clocks in the generating devices. In addition, varying receiving times complicate the process even more due to interferences in the network. Streaming applications must therefore be aware of the assumptions of Atomicity, Consistency, Isolation and Durability (ACID). An example for this is the difference whether an account is funded before or after money is deducted from it.
- **Consistency and Durability**: This is related to issues related to accessing a data set at any point in time, which may have already been modified in a different location. For example, data durability issues are important when working with streams in the cloud.
- **Fault Tolerance and Data Guarantees**: Here, these considerations must be taken into account when working with any distributed system, such as a stream processing system. It relates to how the system handles an interruption and how operations can be resumed without losing or duplicating data.

Fortunately, frameworks exist to help with this multitude of challenges. These tools aid users to design and build data streaming applications. One such framework is Apache Flink, which is used in the course of this thesis and is introduced below.

2.4 Apache Flink

Apache Flink is described by [8] as a framework and distributed processing engine for stateful computations over unbounded and bounded data streams, as discussed in the previous Chapter.

As a distributed system, Flink requires computational resources to run the applications. The architecture of Flink is depicted in Fig. 2.3. According to [8], Flink can be set up with common cluster resource managers such as Hadoop [18], Apache Mesos [19] and Kubernetes [20]. However, it is possible to run it as a stand-alone cluster. When a Flink application is deployed, the required resources are automatically identified based on the configuration and requested from the resource manager. In case one container fails, it will be replaced by a new one. It is designed to run stateful streaming applications of large scale. Flink applications are divided into a number of tasks, which are executed concurrently in a cluster. Thus, a potentially unlimited number of CPUs, memory, disk and network Input / Output (I/O) are supported. It features an asynchronous and incremental Checkpointing algorithm that has minimal impact on processing latency and guarantees exactly-once state consistency. The types of applications that a stream processing framework can support are defined by three components that need to be controlled. These are streams, state and time, which are explained in more detail in the following subsection.

2.4.1 Managing Streams, State, Time

According to [8], the types of applications that can be covered by a stream processing framework are determined by how well each dimension's stream, state and time is covered.

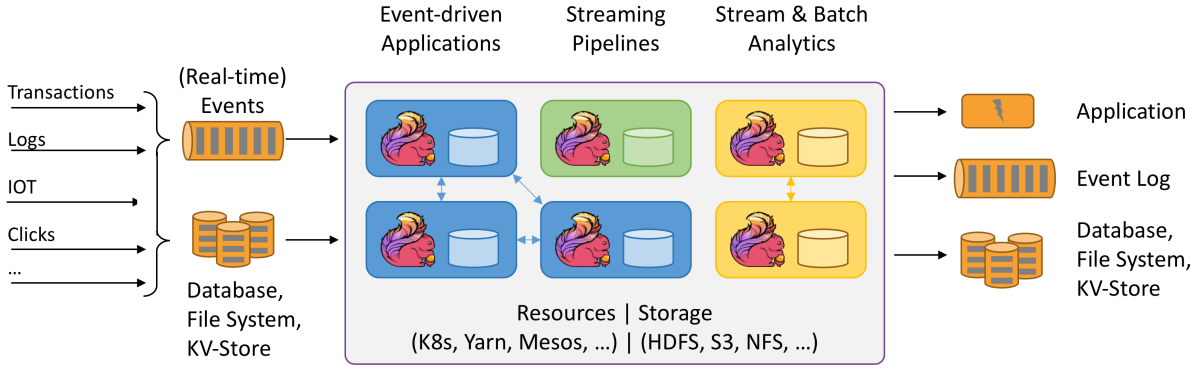

Figure 2.3: Flink Architecture as Shown by [8]

The following paragraphs elaborate on these and explain how Flink handles them.

Streams are a basic aspect of a stream processing framework. A stream can have different characteristics that require different processing approaches. Flink is a very flexible tool that enables the processing of bounded and unbounded data. It is possible to process the events in near-real-time or to access stored resources.

Any non-trivial streaming application will need to work with **state** at some point. For example, intermediate results might need to be cached in order to be accessed again at a later time. Flink provides a variety of functions to meet these requirements described below.

- **Multiple State Primitives**: Flink provides different state primitives for different application areas. These are atomic values, lists, and maps. The developer must choose the optimal structure.
- **State Backends**: Intermediate state is handled by pluggable state backends. One can choose between a Java-heap or file-based backend, as well as an implementation of RocksDB [21]. Additionally, Flink allows for the implementation of custom state backends.
- **Exactly-Once State Consistency**: By using the checkpointing and recovery algorithm, it is possible to guarantee the consistency of the application in case of a failure, if all components in use support it. Other consistency guaranties are *at least once*, and *none*.
- **Very Large State**: By storing state outside of the memory and due to the asynchronous checkpointing, it is possible to handle several petabytes of state (*i.e.,* for RocksDB state backend).
- **Scalable Applications**: Here, distributing state to multiple workers ensures a high degree of scalability.

Time is considered as the third dimension. According to [8], most streams have an inherent time pattern, as the events were generated at a specific time. In addition, many of

the common stream computations are based on time, such as time-windowing, sessioning, and time-based joins. In Flink, a differentiation between event and processing time is made. Event time is the time at which the event was generated, and the processing time is the time at which the event is processed. The time related functionalities are:

- **Event-Time Mode**: If the events are computed with event-time semantics, then accurate and consistent results can be promised, regardless if recorded or real-time events are used.
- **Watermark Support**: Watermarks [22] are used to make statements about the progress of time in event-time applications. They are described as a mechanism to find a trade-off between latency and completeness of results.
- **Late Data Handling**: Due to various environmental influences (*e.g.,* different networks), it may happen that a result is output before all relevant events have arrived. Flink allows users to handle such late events separately, *e.g.,* to update side outputs or previous results.
- **Processing-Time Mode**: Flink also allows users to handle events based on processing-time semantics. For example, this mode can be useful when there are strict low latency requirements and full correctness of results is not required.

To handle these three dimensions, Flink offers a layered Application Programming Interface (API) that allows different levels of abstraction. It represents a trade-off between conciseness and expressiveness.

2.4.2 Layered APIs

Fig. 2.4 visualizes the three API levels, which are examined in more detail in the following Section.

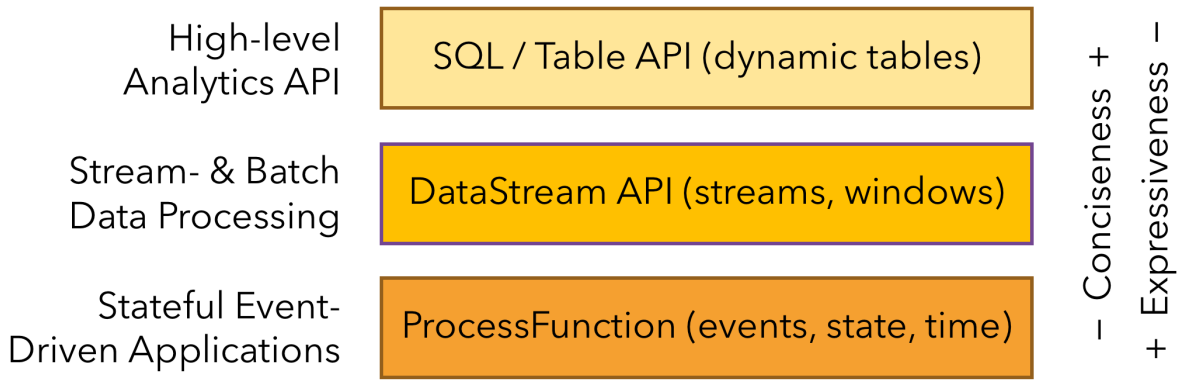

Figure 2.4: Layered APIs as Shown by [8]

The most expressive API level is that of the **ProcessFunction**. It allows one to manage individual events from one or two input streams. Furthermore, a fine granular access

to the state and time dimensions is possible. A `ProcessFunction` can modify its state arbitrarily and register timers which trigger a callback at the given time.

The **DataStream API** provides primitives for a variety of common stream processing operations. These include windowing, transformations or enriching events by querying external data stores. Example functions are `map()`, `reduce()`, and `aggregate()`.

At the highest level of abstraction is the **SQL and Table API**. Both APIs are unified for bounded and unbounded data and return the same results. According to [8], the SQL and Table API is designed to simplify the definition of data analytics, data pipelining and Extract, Transform, Load (ETL) applications.

This work will mainly focus on the first two layers, as a high degree of flexibility is needed for the planned optimizations. Flink places another important emphasis on the operationalization of stream processing. Accordingly, the next subsection describes how these topics have been covered.

2.4.3 Handling Operations Challenges

Many streaming applications are designed to run continuously for extended periods of time with minimal downtime. To enable this, a streaming framework must provide error recovery and monitoring of running jobs.

It is imperative that a distributed stream processor such as Flink can **recover from errors and failures** to enable 24/7 operation. It must not happen that in case of an error the whole history of events has to be rolled up again. It must also be ensured that the state remains consistent. To make this possible, Flink offers the so-called Checkpoints of application state. Since the state can potentially cover several terabytes, the Checkpoints allow asynchronous and incremental creation. Flink enables both end-to-end exactly-once semantics, which means that in the event of an error, data is only written out once. Additionally, Flink integrates with the most popular cluster managers and offers a high availability mode based on Apache ZooKeeper [23].

Another topic within streaming applications is **maintenance and updating**. When bugs need to be fixed or new features are implemented, it must be ensured that the state can be migrated. For this purpose Flink offers Savepoints, which are very similar to Checkpoints, but are not automatically generated or deleted when the application is stopped. A Savepoints can be used to start a state-compatible application and initialize the state. The use cases covered by this are manifold. With the help of the Savepoints, an application can be further developed or supplemented with fixes. A version upgrade can be carried out, the parallelism of an application can be changed, test scenarios can be run through or simple archiving of the Savepoints can be established. The differences between Check- and Savepoints mentioned in [24] are presented in Table 2.4.3. Checkpoints are mainly used as a recovery and failover mechanism for possible job failures in Apache Flink. On the other hand, Savepoints are used as a manual backup for restarting or continuing a Flink job. Checkpoints are automatically triggered by Flink and can be

	Checkpoint	Savepoint
Size	Small	Large
Incremental	Yes	No
Periodic	Yes	No
Application Upgrade	No	Yes
Change of Parallelism	No	Yes
Objective	Recover/Failover Mechanism	Backup For Restart or Continuation
Goal	Quick Restore of Data	Data Portability

Table 2.3: Difference Between Check- And Savepoints

created incrementally. However, the creation of Savepoints must be forced manually by the user.

Furthermore, an important aspect is the **runtime monitoring** of the Flink cluster. It allows users to get an insight into the system and the running jobs. Flink offers a dashboard (web User Interface (UI)) which provides functions for inspection, monitoring and debugging of running applications. Flink also implements the SLF4J [25] logging interface and additionally, provides an enormously powerful means to create custom metrics. The metrics can, in turn, be exported to various reporters such as JMX [26], Prometheus [27] or SLF4J [25]. Last but not least, Flink offers an API interface, which allows for the addition of new applications, the creation of a Savepoint and the running and termination of applications. By now, a lot has been said about the architecture of Flink, but the actual processing of the data is handled by so-called transformations. These will be looked at in the next Section.

2.4.4 Transformations

Transformations allow the user of the framework to transform one or more data streams into a new one. According to [28], a distinction is made between basic transformations (*e.g.,* `Map`, `FlatMap`, `Filter`), keyed-stream transformations (*e.g.,* `KeyBy`, `Reduce`) and multi-stream transformations (*e.g.,* `Union`, `Connect`, `CoMap` and `CoFlatMap`). The most important operators for the following chapters and the knowledge required are explained in more detail in the following paragraphs.

The **Map** transformation (*cf.* Listing 2.1) is used to apply an operation to the single, unrelated events of a stream. The transformation takes one input and generates one output. The input and output type do not have to be the same.

```
1 val environment = StreamExecutionEnvironment.createLocalEnvironment()
2 val input : DataStream[Int] = environment.fromElements(1, 2, 3)
3 val result : DataStream[String] = input.map(x => x.toString())
```

Listing 2.1: Map Transformation That Converts a Digit Into a String

The **FlatMap** transformation is similar to the `map` functionality, but for each incoming event, it returns zero, one or more output events. It represents a generalization of a `filter` and `map` operation. An example that is often used according to [28] is the division of a sentence by spaces as presented in Listing 2.2. As a result, each word is output as a separate record.

```
1 val environment = StreamExecutionEnvironment.createLocalEnvironment()
2 val input : DataStream[Int] = environment.fromElements("This is a
    sentence", "This is also a sentence")
3 val result : DataStream[String] = input.flatMap(x => x.split(" "))
```

Listing 2.2: FlatMap Transformation That Splits a Sentence

Filter is used to evaluate events in a stream based on a condition, followed by dropping or forwarding the records accordingly. If the condition evaluates to false, the event will be dropped and if true, then forwarded. Calls to the `filter` function return a `DataStream` of the same type. This behaviour is presented in Listing 2.3.

```
1 val environment = StreamExecutionEnvironment.createLocalEnvironment()
2 val input : DataStream[Int] = environment.fromElements(1, 2, 3)
3 val result : DataStream[Int] = input.filter(x => x != 2)
```

Listing 2.3: Filter Transformation That Filters a Stream Based on a Boolean Condition

KeyBy is used to divide a stream into logical, disjoint partitions, each containing the records of the consistent key set. It is the basis for horizontal scaling and comparable with `group by` in relational systems. The main goal is to process groups of records that share a defined property, the key. The output of such a transformation is a `KeyedStream`. State-dependent operations on a `KeyedStream` operate in the context of the corresponding key. Thus, all events with the same key access the same state. Elements with the same key are consistently processed by the same parallel subtask of the subsequent operator. Events with different keys can also be handled by the same parallel subtask, but the keyed state is still distinguished based on the event key. The definition of a key is shown on the next lines.

```
1 val environment = StreamExecutionEnvironment.createLocalEnvironment()
2 val input : DataStream[(String,Int)] = environment.fromElements(("A",1),
    ("B",2), ("C",3))
3 val result : KeyedStream[(String,Int),String] = input.keyBy(x => x._1)
```

Listing 2.4: KeyBy Transformation With Key Extraction

Reduce is used to apply a reduction function (*cf.* algebraic semigroup operation) to a `KeyedStream`. Each incoming event is combined with the currently reduced value. The type of the stream is not changed. The reduce function keeps a state for each processed key. Since this state is never cleaned up, it is important to use this function only on bounded key domains. In the example presented in Listing 2.5, the stream is keyed by the category, and then the numbers are summed up.

```
1 val environment = StreamExecutionEnvironment.createLocalEnvironment()
2 val input : DataStream[(String,Int)] =
    environment.fromElements(("Category1",1), ("Category2",2),
    ("Category1",3))
3 val keyed = input.keyBy(0)
4 val result : DataStream[(String,Int)] = keyed.reduce((x,y) => (x._1, x._2
    + y._2))
```

Listing 2.5: Summation of Values by Category

Union combines two or more streams of the same type and produces a new DataStream, which contains all events of the incoming streams. No specific order of the events is established, since the forwarding is done according to the FIFO principle. Duplicate elimination does not take place. Everything that arrives is forwarded.

Connect, CoMap and CoFlatMap if two streams are combined that do not have the same type, the `Connect` function has to be used. As a result a `ConnectedStream` object is returned. This object provides a `map` and `flatMap` transformation, which requires an implementation of the `CoMap` or `CoFlatMap` interface. These are typed on the respective input of the first and second stream, as well as the output. The interface defines two `map` (`map1()` and `map2()`) or `flatMap` (`flatMap1()` and `flatMap2()`) functions for the input streams. If a record is delivered on the first or second stream, the corresponding function is called. The order in which the functions are called cannot be controlled. The methods are called as soon as an event is available.

In this Section, the most important transformations for the thesis at hand have been explained. Next, the handling of time-based operators is explained.

2.4.5 Time-Based Operators

In Section 2.4, the high-level differences between event and processing time were introduced. The following Section further details these concepts.

Processing time determines the current time of the data stream based on the system clock of the executing machine. This leads to non-deterministic results as the content of the windows depends on the speed at which the records arrive. Nevertheless, this setting allows very low latencies as operators do not have to wait for Watermarks.

When using event time, the operators determine the current model time from the inherent information of the events themselves. Each event has an assigned event time, which determines when the event took place. The logical time of the system is defined by so-called Watermarks, which indicate up to which point in time all events were received with a high certainty. Event time guarantees deterministic results, since the result is not based on how fast the stream was read or processed.

The third time domain is the ingestion time, which is a mixture of event and processing time, and is called for completeness reasons. The processing time of the source operator

is added as event time to each record read and a watermark is generated automatically. However, this mode has no significant advantages over event time since it does not provide deterministic results and has similar performance to event time.

In this section, Watermarks were mentioned. They indicate the current progress of time in the system. In the following lines, this concept will be elaborated upon further.

Watermarks

As mentioned by [28], Watermarks are used to balance the latency and completeness of the results. Specifically, they control how long to wait for events before starting a computation. In operators based on event time, Watermarks are used to compute a point in time when all relevant events have been ingested (with a high degree of confidence), allowing the operator to then proceed forward. Unfortunately, this does not work perfectly in reality. Otherwise, no delayed events could occur. Therefore, the Watermarks must be generated based on heuristics. They help to generate an upper bound for the delay of the events. However, heuristics always contain errors, which then lead to inaccurate Watermarks. These, in turn, lead to delayed results or an unnecessary increase in latency. If the Watermarks are far behind the timestamp of the processed event, the latency is increased. In this case, a result could have been produced earlier, but it had to wait for the watermark. Furthermore, the size of the intermediate state also increases, since more data must be buffered. However, a high completeness of the data can be assumed. If the Watermarks are selected very closely to the arriving events, the computations are triggered faster. However, inaccurate results can occur because not all relevant events have arrived. This distinction is a fundamental characteristic of streaming systems in contrast to batch systems, which assume that all data is available. Flink offers many ways to handle the challenges that arise from these differences, including window triggers, process functions and the handling of late events, which will be explained in the next sections.

Process Functions

Process functions are low-level transformations of the DataStream API. They can be used when more control over functionality, time and Watermarks is needed. They allow to register timers that will be triggered at a defined time in the future. In addition, side outputs can be set up, which can emit records to different output streams. Process functions are usually used to implement special logic when the existing windows and transformations are not sufficient.

Flink offers eight different process function types. These are the `ProcessFunction`, `KeyedProcessFunction`, `CoProcessFunction`, `ProcessJoinFunction`, `BroadcastProcessFunction`, `KeyedBroadcastProcessFunction`, `ProcessWindowFunction` and `ProcessAllWindowFunction`. As the name suggests, each of these is used in a different context.

This Section will mainly focus on the `KeyedProcessFunction` and the `CoProcessFunction`, as they are the most used ones in the present thesis.

The `KeyedProcessFunction` is applied to a keyed stream, returning zero, one or more outputs for each input. The `KeyedProcessFunction` provides the user with the following two functions (except `open()` for initialization, `close()` for cleanup, and `getRuntimeContext()` for access to runtime objects such as the metric groups (*cf.* Section 2.4.8):

`processElement(v : IN, ctx: Context, out: Collector[OUT]` is called for each incoming event. The collector then offers the possibility to emit a record further down the stream to the next operator. The context allows the user to access the key as well as the timer service.

`onTimer(timestamp: Long, ctx: OnTimerContext, out: Collector[OUT]` is the callback that is called when a previously registered timer is due. The context again provides the capabilities of the `processElement` function, but also provides access to the time domain. Again, the collector can be used to push a record to the next operator.

CoProcessFunction

For some cases, it may be necessary to have multiple inputs to a process function. For this purpose, Flink offers the `CoProcessFunctions`. Similar to a `CoMapFunction`, two transformation methods are provided for each input, which are `processElement1` and `processElement2`. In addition, the Context object again provides access to the timer service and, thus, the possibility to register timers. The `CoProcessFunction` will play an important role in the later course of this work to achieve a common output with multiple operators that have a different output behaviour.

Window Operators

Window operators are an essential part of a streaming application. They enable one to perform transformations on bounded intervals (*i.e.,* based on either count or time) of unbounded streams (*e.g.,* aggregations). To create such a window operator, two components are needed:

- A **Window Assigner** controls the membership of an event to an arbitrary number of windows and produces a windowed stream.
- A **Window Function** is needed to process the elements which are assigned to the window.

Listing 2.6 presents the steps needed to use a window assigner and a subsequent `WindowFunction`.

```
1 stream
2     .keyBy(...) //optional
3     .window(...) // add window assigner
4     .reduce(...) // aggregate(...) // process(...)
```

Listing 2.6: Specification of a Window Assigner and Function

Window Assigners

The window assigners are used to assign events to the corresponding windows. A distinction is made between count, tumbling, sliding and session windows. The focus for this work is on the tumbling and sliding windows, which will be examined in more detail in the following paragraphs. Count windows contain a fixed number of events in the order in which they arrive. A session window assigner then groups the incoming events into non-overlapping activity windows of varying size. The intervals between the windows are determined by periods of inactivity.

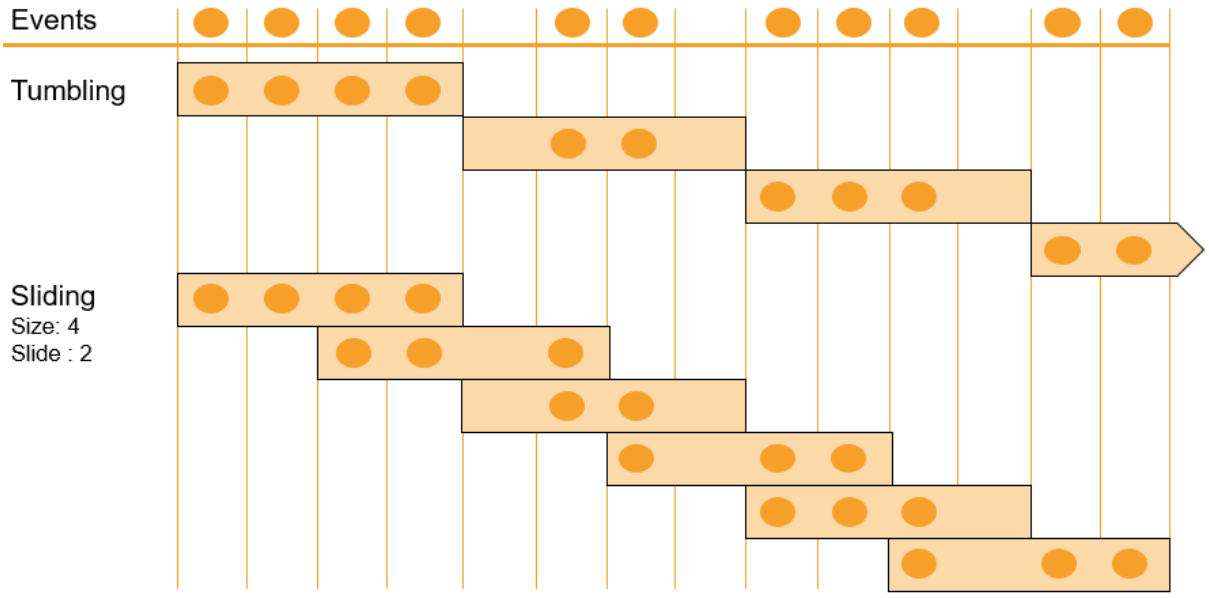

Figure 2.5: Visualization of Tumbling and Sliding Windows

Fig. 2.5 visualizes the difference between tumbling and sliding windows. Tumbling windows are non-overlapping windows with a fixed size. An event is therefore always added to exactly one window. Sliding windows are windows that have a fixed size, but slide at a specified slide interval. If the slide interval is smaller than the size, the windows overlap and an event can be added to one or more windows. If the slide is larger than the size, it is possible that an event is not assigned to any window and is lost. Sliding windows provide a *smoother* aggregation over the data because there is no jumping from one distinct set of data to the next as in tumbling windows.

Window Functions

The window functions are used to perform aggregations on a windowed stream. It is distinguished between incremental and non incremental aggregations.

The **incremental aggregations** are executed directly when an event is calculated for a window. They contain an accumulated value as window state. Therefore, they are efficient in terms of memory. The two incremental aggregation functions are `ReduceFunction` and `AggregateFunction`. The latter is a complication of the former and offers more possibilities to control the aggregation. While the `ReduceFunction` takes two values of the same type and combines them, the interface of the `AggregateFunction` determines the input and output type, as well as the aggregator itself.

The interface of the `AggregateFunction` contains four different functions, which are described in Listing 2.7.

```
1  public interface AggregateFunction<IN, ACC, OUT> extends Function,
       Serializable {
2
3      /*Used to create a new accumulator, reflecting an empty aggregate*/
4          ACC createAccumulator();
5
6      //Adds a new event to an already existing accumulator
7          ACC add(IN value, ACC accumulator);
8
9      //Returns the result from an accumulator
10         OUT getResult(ACC accumulator);
11
12     //Merges two accumulators a and b and returns a combined version of
           both
13         ACC merge(ACC a, ACC b);
14 }
```

Listing 2.7: Interface of an `AggregateFunction`

Sometimes it is necessary to access all elements of a window to perform more complex operations on them. For this purpose, incremental aggregations cannot be used. The `ProcessWindowFunction` offers a `process()` function that is called with the corresponding key as well as the context and the list of elements currently associated with the window. The context is similar to the known process functions and allows for accession of the window's meta data. The `ProcessWindowFunction` is a powerful tool, but it must be used with caution. By keeping all events, the state becomes larger than with incremental aggregations. If the underlying problem can be solved by an incremental aggregation, but for some reason also access to the meta information of the window is needed, a combination of the two functions can be used. For this, the reduce or aggregate() function can be given to the `ProcessWindowFunction` as a second parameter. Subsequently, each event is aggregated incrementally and only a single value is passed to the window function.

Trigger and Evictor

Triggers and Evictors further contribute to the customization of the windowing. A `Trigger` is used to determine when the window is ready to be processed by a window function. Each `WindowAssigner` has a default `Trigger` assigned. If this `Trigger` does not cover all requirements, a custom one can be defined. The `Trigger` interface offers five different functions.

- `onElement` will be called for every event that is added to a window
- `onEventTime` is called when an event time timer fires
- `onProcessingTime` is called when a processing time timer fires
- `onMerge` is called when a merge operation for a session window is performed. As session windows are not used in the present work, this topic will not be covered in more detail.
- `clear` is used to perform any actions when a window is removed

The first three functions are used to determine how to further proceed when called. This is done by returning the `TriggerResult`, which can be one of the following:

- `CONTINUE` does nothing
- `FIRE` is used to trigger a computation
- `PURGE` is used to clear the elements in a window
- `FIRE_AND_PURGE` is used to fire the calculation and then purge the window content

Triggers have access to both state and timers. Thus, a complex logic can be set up to determine when a trigger action has to be performed. For example, these include firing as soon as a certain threshold has been reached or a number of events for a key have been delivered in a predefined time frame.

Evictors are used to remove events *after* a `Trigger` and before/after the window function is executed. The two functions `evictBefore` and `evictAfter` receive as parameters the list of all events belonging to the window, the current number of elements in the pane, the window itself and an `EvictorContext`. The `EvictorContext` offers the possibility to obtain the current processing time, the current watermark and allows accession of the metrics. In comparison to the trigger, no state or timer service is available. Two important aspects to mention are the following: As a list is passed as a parameter, the original order of the events cannot be inferred from their placement. Thus, removing the first or last element from the list does not delete the first or last event of the window. Additionally, as mentioned by [8], specifying an `Evictor` prevents any pre-aggregation. This is due to the fact that all elements of a window have to be passed to the `Evictor` before any calculations.

Handling Late Events

Another important aspect when it comes to time management is the handling of delayed data that arrives out-of-order or after its period of interest. For example, an event can arrive after the calculation that it should have contributed to. This is due to the trade-off between result completeness and latency by the Watermarks. In this case, it must be decided how to deal with such late events. The record can either be dropped, redirected to a separate stream or the old result can be updated and re-emitted. The default way is to drop the events and accept the inaccuracy of the results. This is the default behavior. By redirecting the results, they can be included again at a later time by a backfilling process. Updating the results is certainly the most difficult option. This means that already finished windows must be stored in the state and must be accessed again. Also the operators further down in the flow must be able to handle updates. For the handling of these late events the `allowedLateness` function is available in the event time mode. This way, a horizon can be specified how long the window should be kept in the state.

In the last Section, the most important aspects of time-based operators for this thesis were introduced. Thus, the next Section will focus on the next critical topic, the management of the state.

2.4.6 State Management

Most streaming applications are considered stateful. They continuously read and write state, which is present in form of collected events in windows, intermediate aggregations or other data. In general, all data managed by a subtask and used to calculate the results of a function belong to the subtask's state. According to [8], the state can be defined as a local or instance variable, which is consumed by the business logic of the task.

When a task receives an input, it can read and update its state during the processing of the input and calculate the result. In Flink, a distinction is made between operator state and keyed state, which will be discussed in the next paragraphs.

The **Operator State** is related to an operator subtask. This means that all records handled by the same operator subtask can access the same state across all processed keys. It is not possible to access this state from another task of the same or different operator. Operator state provides the user with three state primitives:

- **ListState** contains the state as a list.
- **UnionListState** contains the state also as a list, but differs in the way it handles a failure or a restart from a Savepoint. With the `UnionListState`, the complete list is distributed to all tasks, and the tasks individually decide afterwards which part of it they need.
- **BroadcastState** is needed for the special case as to when the sate of each subtask of an operator is the same.

The operator state will be used in this thesis for the switch between the Java-Heap and RocksDB state backend introduced in Section 4.2.2. It enables the storage of data in-memory, while staying able to benefit from the Check- and Savepoint algorithm.

The **Keyed State** is handled in the context of a key given by the input stream. This means that all entries with the same key access the same state. Keyed State is understood as a key value map structure that is sharded across all parallel tasks of an operator using the key. It offers the following state primitives:

- **ValueState** is used to store a single value of a specific type. Complex data structures can also be stored as values.
- **ListState** contains a list of typed values per key.
- **MapState** contains a key-value map per key and maps from a key type to a value type.

For the present work, most of the state will be handled as keyed state. Particularly, the `MapState` will be of central importance due to the way it is implemented in RocksDB.

In Flink, the state is stored with a key that is divided into three levels. The first level is the grouping by the actual key which was provided by the user in the `keyBy` statement. The second level is optional and describes the namespace that reflects the windows. In this case, the timestamp of the window is taken as the key component. The third level describes the state primitives. For example, this includes the key of the `MapState` structure. Fast access to the state is of central importance for a stream processing application. How the state is stored, accessed and maintained is handled by the pluggable state backends. This thesis will mainly focus on storing the state on the Java-Heap and in RocksDB. RocksDB is a highly optimized key-value store which uses the memory for caching and the hard-disk for long-term storage.

Using the Java-Heap state backend provides very fast access to the state. Yet, the storage space is limited by the available Heap memory. The RocksDB state backend provides slower access, but the state can grow extremely large. A speciality that will play an important role for this thesis is the underlying implementation of the state primitives in the RocksDB state backend. An inherently given characteristic of the RocksDB state backend is that the `MapState` provides a canonical ordering of the stored elements. For instance, this ordering is not given when using a Heap structure. The reason for this is that the `MapState` implementation by RocksDB relies on so called column-families [29]. They can be seen as a column store index. This canonical ordering allows for retrieval of an iterator from a `MapState`, and as soon as the first element is found, the following entries are read with constant complexity. This is due to the fact that only the next element in a sorted list needs to be returned. For the other state backends and memory structures, this property does not hold.

Nearly all important base concepts for this thesis regarding Flink have been introduced. With the help of the transformations, windows, triggers, evictors, various process functions and the state management, a first pipeline can be created. However, what is still missing is the transformation of a program into an execution plan. Accordingly, the next Section deals with this topic.

2.4.7 From a Program to an Execution Plan

An important aspect of the Flink job execution process is the transformation of a program into a physical execution plan. Depending on various parameters such as the data size or the numbers of machines in the cluster, the Flink optimizer automatically generates an execution strategy for the job. The program passes through four different stages. First, the program is transformed into a `StreamGraph`, then from a `StreamGraph` to a `JobGraph`, from a `JobGraph` to an `ExecutionGraph` and finally to a physical execution plan. However, describing each step in detail would go beyond the scope of this paper. Therefore, this Section focuses on an introduction of the individual transformations which are described by [30] and depicted in Fig. 2.6. Because of later importance, the `JobGraph` and the logical execution plan will be described in more detail.

In a first step, the topologically sorted list of transformations (*i.e.,* program) is transformed into a `StreamGraph`. The transformation starts with the source nodes. A `StreamNode` is generated from each transform operation and a `StreamEdge` from each connection between two `StreamNodes`. Together, the `StreamNodes` and `StreamEdges` form a directed acyclic graph (DAG).

The second step involves converting a `StreamGraph` to a `JobGraph`. The conversion into a `JobGraph` is concerned with combining the individual operators into tasks. The operators are traversed from the source nodes to find those that can be nested. If operators cannot be nested, a separate `JobVertex` is created for both of them. The up- and downstream vertexes are then connected by a `JobEdge` to form a DAG at `JobVertex` level. This nesting of the individual operators is called task chaining. To chain a task, the following conditions shown in 2.8 must be satisfied.

```
1  return downStreamVertex.getInEdges().size() == 1
2  && outOperator != null && headOperator != null
3  && outOperator.getChainingStrategy() == ChainingStrategy.ALWAYS
4  && (
5       headOperator.getChainingStrategy() == ChainingStrategy.HEAD
6       || headOperator.getChainingStrategy() == ChainingStrategy.ALWAYS
7  )
8  && (edge.getPartitioner() instanceof ForwardPartitioner)
9  && upStreamVertex.getParallelism() == downStreamVertex.getParallelism()
10 && streamGraph.isChainingEnabled();
```

Listing 2.8: IsChainable Conditions

The downstream node may only have one input and the upstream and downstream node may not be null. The downstream node needs to have the chainable policy `ALWAYS` and the downstream node *ALWAYS* or *HEAD*. In addition, there must be a forward partitioner and the upstream and downstream nodes must have the same parallelism. Finally, and most fundamentally, chaining itself must be allowed. Chaining allows the co-location of successive transformations in a single task. By omitting the de/serialization and sending of data between several threads the performance can be optimized.

In a third step, the nodes are sorted starting from the source nodes. An `ExecutionJobVertex` is created for each `JobVertex` and an `IntermediateResult` is created for each

`IntermediateDataSet` of the `JobVertex`. This result is used to set up an upstream-downstream dependency to form a DAG at the `ExecutionJobVertex` level. This DAG is the `ExecutionGraph`.

Finally, the last step is to convert the `ExecutionGraph` into a **physical** execution plan, which is then serialized in order to be transferred to the execution cluster (*i.e.,* task managers).

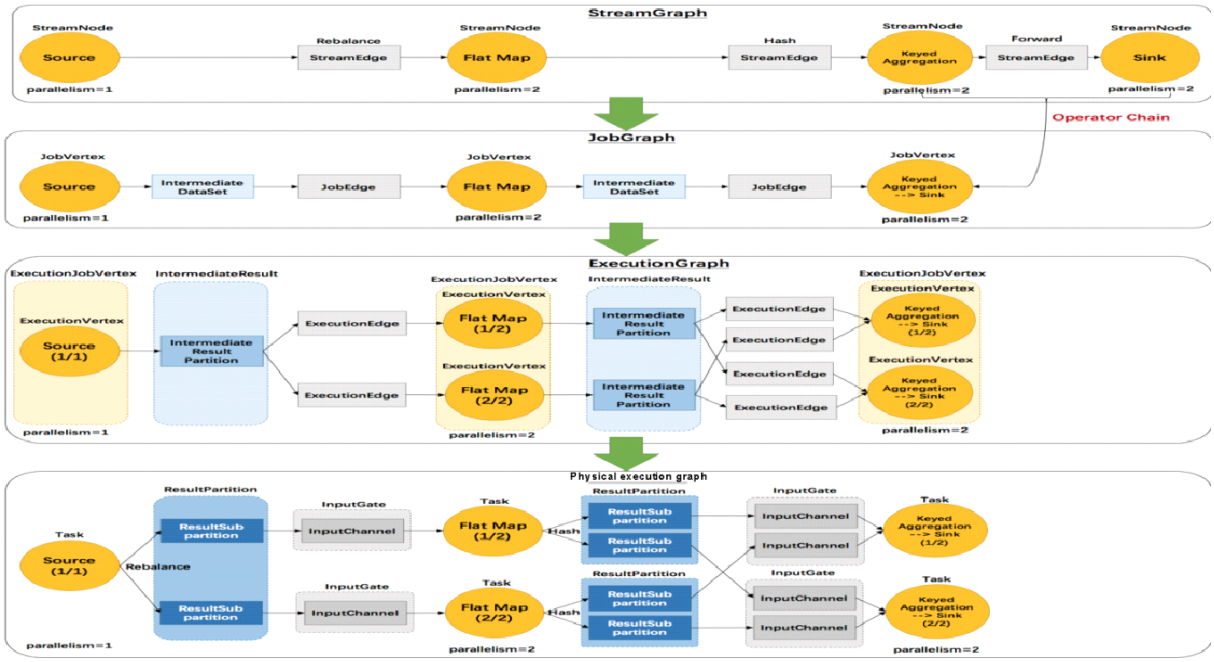

Figure 2.6: Transformation of a Program Into an Execution Plan Visualized by [30]

As visualized in Fig. 2.6, reading such a plan becomes increasingly complex with each level. However, for a developer only a high-level overview of the individual operators (`StreamGraph`) or a view on the summarised tasks (`JobGraph`) is important. When a high-level overview is required, the function env.getExecutionPlan() can be called. It returns a JavaScript Object Notation (JSON) representation of the `StreamGraph`, which represents the logical execution plan as depicted in Fig. 2.7. It contains the operators, their parallelism, the links between the operators as well as the partitioners. The partitioners distinguish between forward, hash and rebalance. Each having different properties for processing. In the following section, the forward and hash partitioners are discussed in more detail. These are of significant importance for the thesis.

In general, the partitioners are used to partition the incoming data and define how data is distributed across parallel task instances. Fig. 2.8 visualizes the difference between a hash and a forward partitioner.

- When using a **forward partitioner**, all data consumed by one of the parallel instances of an operator is passed to the exact same parallel instance of the next operator. Therefore, the same degree of parallelism is expected.

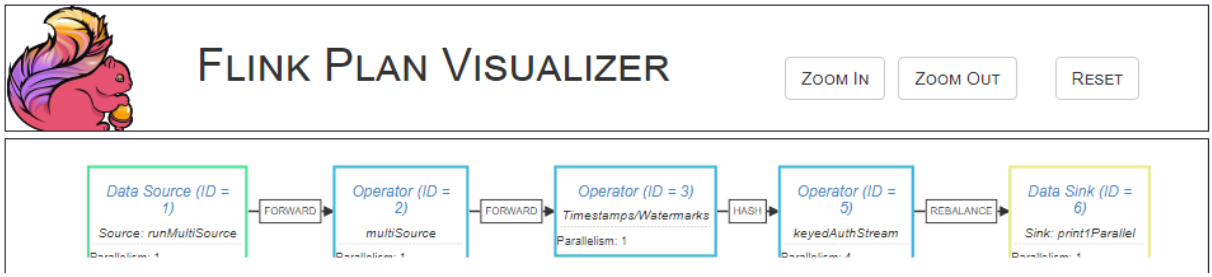

Figure 2.7: Visualization of a Simple Flink Execution Plan

- When using a **hash partitioner**, a hash is calculated based on the key attribute for each entry. Afterwards, the entries are divided among the available parallel instances of the following operator. As already mentioned before, such a partitioner is for instance generated by a `keyBy`.
- A **rebalance partitioner** is either generated by an explicit usage of `rebalance`, or when the parallelism changes from one operator to the next. The data is then rebalanced with a round-robin scheme [31]. This operation can be used to counteract data skew.

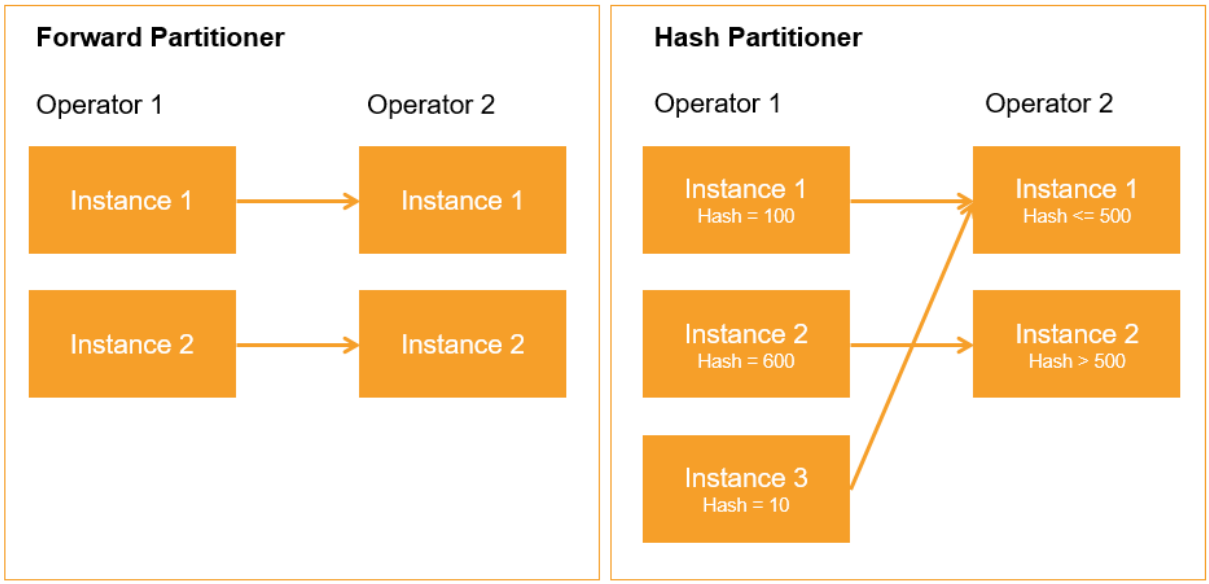

Figure 2.8: Difference Between a Forward and Hash Partitioner

When an overview of the combined operators in the tasks is required, the web UI of Flink can be consulted. When the job is running, a representation of the `JobGraph` will be displayed. Compared to the `StreamGraph`, it no longer shows all operators individually, but the tasks. Instead, the chained operators are displayed in combined form as depicted in Fig. 2.9.

The last Section described how a Flink program is transformed into a specific physical execution plan. Inevitably, the question arises as to how the performance of such a

Figure 2.9: Job Graph of the Example Shown in 2.7

complex system can be measured. For this purpose, the metrics implemented by Flink can be used. They allow for the integration of own measurement logic into the system.

2.4.8 Metrics

Metrics are used in Flink to get a more accurate insight into the current state of the job, as well as the cluster. It is not realistic to analyze the task log in real time on the cluster, which makes efficient monitoring even important. Flink offers the possibility to monitor the current status of the system using metrics. There are four different metric types:

- A **Counter** is a simple counter which can be incremented or decremented using `inc()` or `dec()` function calls.
- A **Gauge** stores and retrieves a value of any type.
- A **Meter** measures the average throughput. The `markEvent` function can be used to mark the appearance of an event.
- A **Histogram** measures the distribution of long values.

According to [32], metrics are organized in Flink in groups of a multi-layered structure. The metric is uniquely determined by its name and the assigned group. Furthermore, Flink allows to create additional subgroups and thus, to extend the layered structure. A distinction is made between the predefined system and user-defined metrics. The system metrics include measurements of the Java Virtual Machine (JVM) on master or worker level. For example, these include heap memory usage or buffer pool utilization. In addition, there are metrics which measure the number of active threads and garbage collector information. The predefined metrics measure the latency of an event through the system. In the current version, `LatencyMarkers` are passed through the system. When they reach the end of the pipeline, the measured latency is reported. The problem is that the computational steps in the operators are not considered. Only the pure I/O latency is measured, since the `LatencyMarkers` are simply passed through the pipeline. Furthermore, the latency markers cannot overtake normal events, so the latency is only delayed when events are buffered in front of an operator.

Another system metric is the cluster monitoring. For example, it shows the timestamp of the last Checkpoint. This is critical because if the Checkpointing is not working, a job has to start from a state far in the past after a restart.

Lastly, the RocksDB metrics are important to mention. They allow a detailed insight into the state backend. For example, the number of active keys can be displayed, which allows approximate size estimations.

The user defined metrics refer to the DataStream API and allow to implement an own measurement logic. To initialize such a metric, the `RuntimeContext` of a function implementing the `RichFunction` interface must be accessed. The `RuntimeContext` provides access to the metric interface. Using the `getMetricGroup` function, the metric group can be obtained. By using these groups, a developer can then either add new subgroups or define own metric types.

Now that metrics can be created, the question arises as to how they are taken up by the monitoring systems. There are three different methods to access the metrics. They can either be accessed in the web UI, a rest interface or the metrics reporter. The latter is the most commonly used method, which will also be used in this thesis. The first two methods obtain their metrics by accessing a central node, which pre-aggregates the data. The metric reporter accesses the nodes individually. It reports raw data, which can be processed with better performance. The non-centralized architecture is also free of the problems that a centralized node entails (*e.g.,* insufficient memory).

In this section, the most important components of Flink for the given context were introduced. The next step is to describe the baseline, which was implemented before the master thesis and should serve as a comparison for the elaborated optimization strategies.

2.5 Fraud Scoring Baseline Implementation

The baseline implementation implementation was a first attempt to fulfil the requirements posed by the Fraud Scoring system. Its development started before the master thesis and the initial goal was to calculate the required 1485 features with a latency of a few milliseconds per event. The required throughput and latency could never be achieved. Thus, the thesis at hand uses this baseline implementation to evaluate various optimization strategies and guidelines to leverage the overall performance of a streaming job. The weaknesses of the baseline implementation will be discussed in detail in the following Section.

The general architecture of the baseline is visualized in Fig. 2.10. On an abstract level, a subdivision is made into the individual scenarios. First, the events are distributed into the key domains (*i.e.,* blue boxes). Inside of the key domains, a split into the different aggregation types is carried out. For the baseline implementation, an statistical momentum (*e.g.,* sum, mean, standard deviation, skewness), a unique count and an oldest / recent aggregate function is required. These aggregation types (*i.e.,* red boxes) are then again applied to the individual windows (*i.e.,* green boxes). The following window types are required:

- The **Continuously Sliding Windows** should contain at any given point in time the records of the last 24 hours. From now on, they will be referred to as D0 or day zero windows
- The 7, 28, 182 and 392 days **Sliding Windows** should contain the data from the given time span and slide by one day.

The different windows of the aggregation functions are then combined and joined together across the different scenario domains(*i.e.,* orange boxes). This join operation is carried out using the split/join pattern [11] mentioned in Section 3.2.

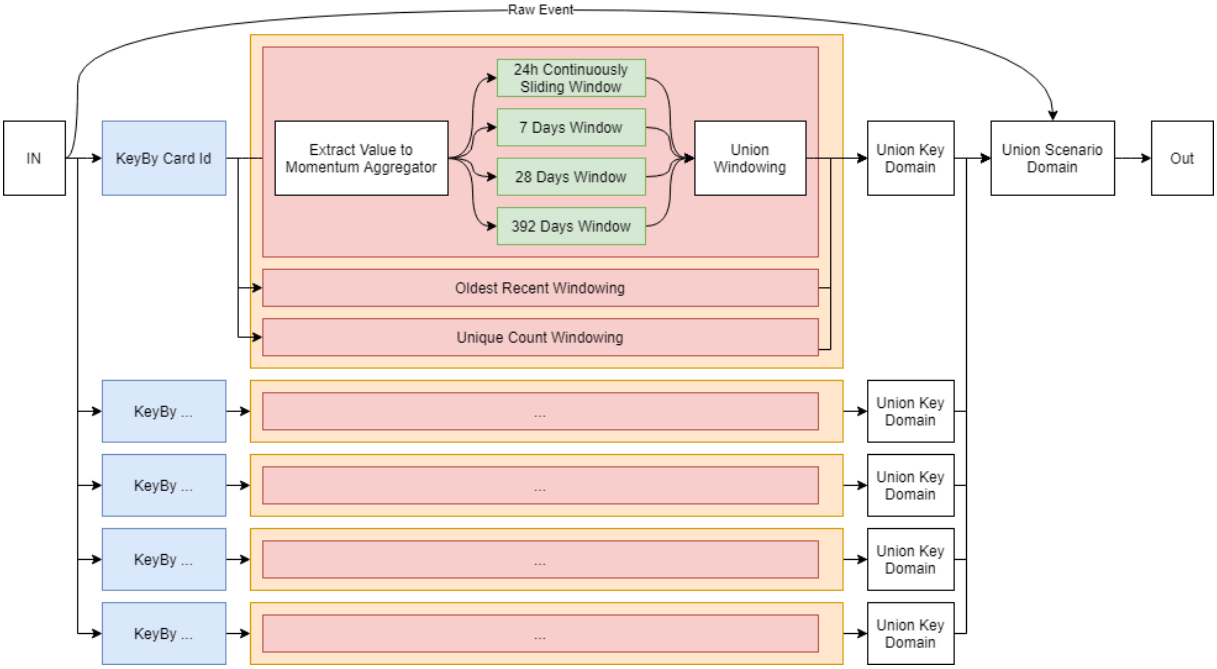

Figure 2.10: Abstract Baseline Architecture

In the following sections, some aspects of the baseline implementation are discussed in more detail. Namely the general windowing strategy and the continuously sliding windows with their inherent performance issues. As a third point, the de/serialization overhead between the tasks of the job will be highlighted.

2.5.1 Windowing Strategy

In general, a separate sliding window was used for each feature to be calculated. This means that a 7-day, 28-day, 182-day, or 392-day sliding window was created for each aggregation. What sounds logical has a variety of negative implications. This means that, in the worst case, a new 392-day window is opened every day for each key. If a new event comes into the pipeline, it will be stored by the window assigner into the corresponding window. This means that per feature 392 windows are open in the current implementation. Additionally, the events need to be stored in every window for 392-days which leads to an $O(n^2)$ complexity. In terms of operational state, this is clearly

not optimal. Due to the fact that with RocksDB a file based state backend is used, state accesses have to be executed with restraint. The *card id* domain (*i.e.,* scenario 1) can be estimated by the maximum number of cards. However, the subdivision by card and another attribute in scenario 2 complicates reliable estimations. The estimation of scenario 3 is more evident, since an upper limit is given for the required attributes. The following list presents the approximate expected number of keys per scenario.

- **Scenario 1** The key by card id domain is certainly the one that can be most easily covered with an upperbound estimation. The balance sheet of the company [33] shows that 1.7 million cards were in circulation at the end of 2020. This number will certainly increase in the next few years, which means that the number of cards can be expected to reach around 2 millions.
- **Scenario 2** For data protection reasons, no exact details about the attributes used for the fraud model can be provided. However, these are attributes partly have a high range of values. This range in combination with the maximum number of cards results in a very high potential number of keys. The two most decisive categories contain about 2.5 million and 7.5 million unique values, respectively. Furthermore, both numbers are increasing. This gives a clear indications of the dimensions.
- **Scenario 3** The upper limit can be estimated by taking the upper limit (10'000'000) of the used attributes times the potential classifications (6). Again, no precise information about the attributes used may be given. However, an upper limit of approx. 60'000'000 keys is expected.

The potential first level key space which has to be covered is large. However, the key space as well as the data for individual keys is limited by the maximum number of events per year. The approximate number of annual transactions can be calculated by the annual Swiss Payment Monitor [3]. According to [3], 530 million credit card transactions were made in Switzerland in 2019. The cooperating company is one of the three big card issuer in Switzerland. The whole market currently consists of seven players. So, roughly, it is assumed that the three have a share of 7/8 of the market. Divided among three providers, this makes a volume of 154 million transactions per year. The volume of cashless payments is considered to be increasing due to the broader acceptance. It can therefore be estimated that there will be between 150 and 200 million transactions per year. With this number, new estimations regarding the expected key space can be made. Fig. 2.11 provides an overview of how each scenario generates keys per incoming event. It can be seen that for each event 9 keys are generated based on the card id and in 8 cases a second attribute. In the third scenario, the key space is also based on the incoming events, but is capped by other factors. For example, there are only 195 countries in the world. Therefore, there would be only 195 potential keys for this attribute, multiplied by the number of classifications.

By using the maximum number of cards as well as relying on the approximate number of attributes to expect, an estimate can be made about scenario 1 and 3. Scenario 2 is difficult to estimate because of the potential combinations of cards and attributes. Assuming each transaction of a card differs from the others in all attributes, 8 new keys

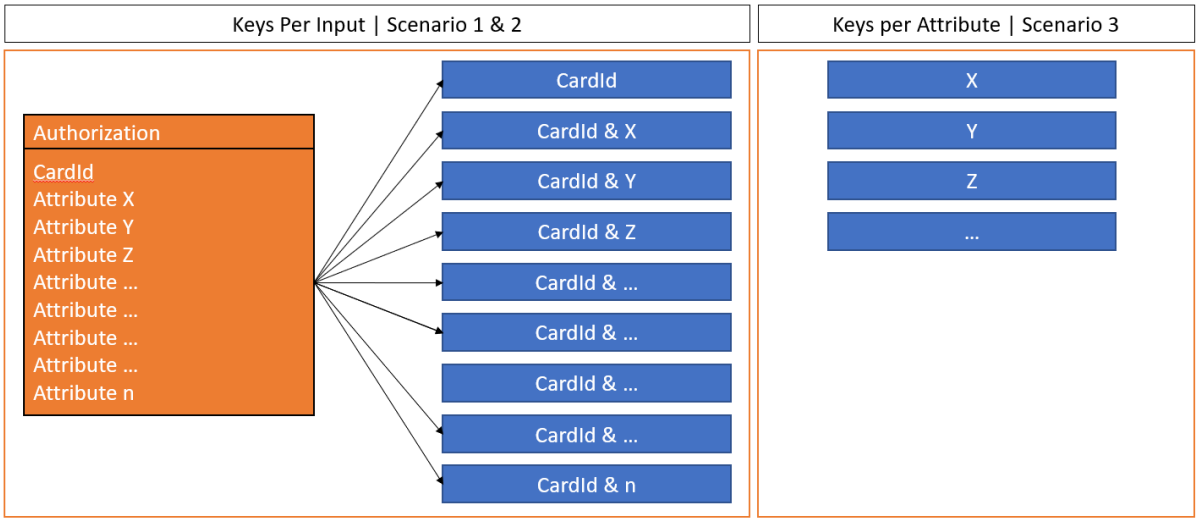

Figure 2.11: Splitting of Key Space Depending on Input and Categorical Set

are generated per card usage. Extrapolated to the transaction volume, this results in a number of 1.6 billion keys. Thus, the keys of scenario 1 and 3 are of minor importance compared to the maximum number of keys in scenario 2. The challenge with a large key space is, that with a growing number of entries also the read and write times grow. Not only do many keys have to be stored, but for scenario 3, also the amount of data per key becomes a problem. For example, an aggregation by region has to be performed. Lets assume there are 20 major regions. With roughly 400'000 transactions per day, each day 20'000 events have to be processed per region. When considering windows with a length of 392 days, the issue gets evident. This topic also applies to the continuously sliding windows, which will be discussed in the next Section.

2.5.2 Continuously Sliding Windows

The second challenge of the baseline implementation refers to the continuously sliding windows. These windows must always contain the aggregated values of the last "x" hours at any time with a millisecond precision. In the specific use case at hand, the window must always contain the events of the last 24 hours for each key. These windows are also known as day zero or D0 windows. This requirement implies that all records belonging to this time frame must be buffered in the window, as the records have to be removed from it when their validity period is exceeded. Flink offers a few ways to solve this issue. However, the two presented variants contain some stumbling blocks regarding efficiency. One possibility is to use triggers. First, the trigger function `onEvent()` fires the process function. Then a timer is registered in the trigger which in turn fires as soon as the event becomes invalid. The `ProcessFunction` is then called again with the corresponding event. Unfortunately, the `ProcessFunction` lacks the knowledge whether this event was received already and thus is to be removed, or it was sent the first time and needs to be added to the state. This information is not passed on by the trigger. The `processEvent` function of the `ProcessFunction` is just called again. This requires for every record that is received

to check the state of the `ProcessFunction` whether the event is already present or not. If it is present, the value must be subtracted and removed. Otherwise, it needs to be added to the state. Thus, for each received event all stored records in the corresponding state are queried and aggregated, which results in an $O(n^2)$ complexity, where n is the number of events per window. Another possibility is to use an `Evictor`, which is used to remove events from a window. However, the interface of the `Evictor` specifies that all records are kept and passed to the `ProcessFunction` in a list. Due to the properties of a list, it does not contain any actual ordering or efficient search function. If a value has to be removed, the entire list must be iterated through. This logically also leads to the fact that the entire list must be re-aggregated for each addition or removal operation. This again results in an $O(n^2)$ complexity. Potential solutions to the problem are presented in Chapter 4.

2.5.3 Combination of Different Aggregations

Another problem is the efficient merging of windows with different output behaviour due to their window length. For instance, continuously sliding and daily sliding windows have to be merged. The difficulty is that the continuously sliding windows have one output per event. However, the daily sliding windows have one output per day. In addition, the window aggregations have to be pivoted and enriched with the original event. A potential solution for this is the split join pattern which is introduced in Section 3.2 and critically analyzed in Chapter 4.

2.5.4 Shuffle Operations Between Operators

When looking at the execution plan of the baseline implementation, one thing that gets noticed is the large number of shuffle operations represented by the hash partitioners. As explained in Section 2.4.7, these shuffle operations generate I/O by exchanging data between different machines of a cluster. Therefore, it is important to reduce the usage of these operations as much as possible. In some cases, sending data across the network is not necessary and only leads to computational overhead. If operations are carried out on the same key in a pipeline, the data could stay on the same node until it is keyed by another attribute or the parallelism changes. Additionally, the chaining of the individual operators will play an important role in the later phase of the analysis.

The problems listed above are not exhaustive, but they are potentially responsible for a large part of the bad performance of the streaming job. In a first phase, reliable measurements have to be carried out to confirm the suspicions. Afterwards, optimization strategies to eliminate the problems have to be proposed.

The fields for optimization can therefore be roughly divided into three distinct fields.

The first topic of interest is state management. Here, the cardinalities of the keys and the characteristics of the state backends are examined. Possible stumbling blocks are the access times of RocksDB or the storage capacity of the Java-Heap state backend.

The second category is concerned with optimizations regarding the I/O. The term I/O is used to describe the time of data transfer and how long data is buffered between the individual steps in a streaming pipeline. For example, this includes the de/serialization of data records between individual tasks during shuffle operations. Another possibility is the optimization of the actual business logic and the algorithms used. Here, especially the windowing and aggregation algorithms are of interest.

To test the baseline, it is important to have events with a similar statistic distribution profile to production conditions. It should enable setting up tests that are as close as possible to reality. Yet, due to data protection regulations, no real production data may be used. The next Chapter deals accordingly with how this test data was generated.

2.6 Data Generation

Due to the data protection regulation, no data in production may be used for this work. To remedy this problem, a data generator was implemented in the previous master project [6]. It allows to generate test data based on statistical distributions, which were modeled in advance. The data generator works completely deterministic and is able to export the data either as an Avro file [34] or to write it directly into a Kafka topic. The Avro files contain the data serialized in a compact binary format.

The generated data should be as close as possible to the statistical distributions found in the productive systems. This is important for the validity of the tests. A streaming system behaves differently when confronted with large amounts of data and a possible critical state is only reached after a certain amount of data. Another important criterion is the achievement of a **steady state** of the aggregations. This term describes the moment at which values are continuously calculated to and from the windows. For a window containing 392 days, this is logically only possible after 392-days. Once this state is reached, the system operates in a mode that comes closest to production conditions and thus allows to make statement about the performance of the system under real conditions.

The data generation supports two different time modes. Events can be generated either in real-time or in full-speed mode. If the real-time mode is selected, the events are produced at the actual speed specified by the model. If it is defined that statistically every 3 seconds an event happens, this rate can be seen effectively in the data. In full speed mode the events are produced as fast as possible and there is no effective delay (*i.e.,* the model (=event) time progresses faster than the processing time, however the rates in term of event time are still the same). Thus, the model time progresses independently of the time. The data generator also offers a combined mode in which the events are generated at full speed up to a certain point. After this point the mode changes to real-time. For the tests in this thesis the mode is set to fullspeed to generate a larger amount of data. After a certain point, the data should be generated in real time to simulate a running operation. In the model, 2'000'000 customers and 5'000'000 merchants will independently carry out their transactions according to their preferences and locations. These events are then delayed by a `ProcessFunction` to simulate possible network delays and to bring

the events out of the correct order. Finally, the events are written to the corresponding Kafka Topic of the Viseca Streaming Data Platform (SDP).

This Section concludes Chapter 2. It served to provide the necessary background information for the thesis and to introduce the stream processing framework Apache Flink. Next, related work and concepts are introduced in Chapter 3. The gathered insights are then used to design possible optimization strategies in Chapter 4.

Chapter 3

Related Work

A literature review was conducted to identify possible solutions to the problems motivating this work and to highlight potential developments. The first step is to look at credit card fraud prevention systems that use streaming systems. Especially the number of features to be calculated and the amount of data used will be of interest. Next, the split join pattern for Flink introduced by [11] will be looked at more closely. Another important point discussed in this Chapter is the slicing of windows which will be used for implementing the continuously sliding windows as well as other optimizations regarding windowing. As a last point, the parameter tuning of RocksDB as described by [35, 4, 21] is shown.

3.1 Credit Card Fraud Prevention Approaches

In literature, various works [12, 13, 14] address credit card fraud prevention systems in combination with machine learning. What is mostly missing is the feature engineering process. Implementations that rely on data streaming are also rare. The following two approaches describe fraud prevention systems that use streaming methods for their data processing. The paper of [15] works with Apache Spark [36], whereas the paper by [11] implements its solution by using Apache Flink [8].

3.1.1 SCARFF: A Scalable Framework For Streaming Credit Card Fraud Detection With Spark

[15] present a realistic and scalable fraud detection system. The system is designed as an open source platform and enables the analysis and processing of streaming data to generate reliable fraud alerts in near real time. The work deals with 5 core topics. First, the design, implementation and testing of an open source solution developed with state-of-the-art components of the Apache environment is described. According to [15], the architecture is able to handle the whole process of ingestion, streaming, feature engineering, storage and classification. Second, a scalable learning solution is implemented which performs the classification of the computed features. The third topic is the most interesting one for

the present work. The design of an on-line feature engineering functionality. Fourth, an assessment of the implemented system follows. The scalability, performance and precision of the approach are evaluated. For the tests, 8 million transactions of 1.9 million cards are evaluated. The last contribution of the work is the virtualization of the entire workflow as a Docker [37] container to make the implemented solution completely reproducible.

The architecture of the envisioned solution is depicted in Fig. 3.1. Since the focus of this work is on optimizing a feature engineering pipeline, this topic will be discussed in more detail in the following paragraph. The areas dealing with machine learning are left to the interested reader.

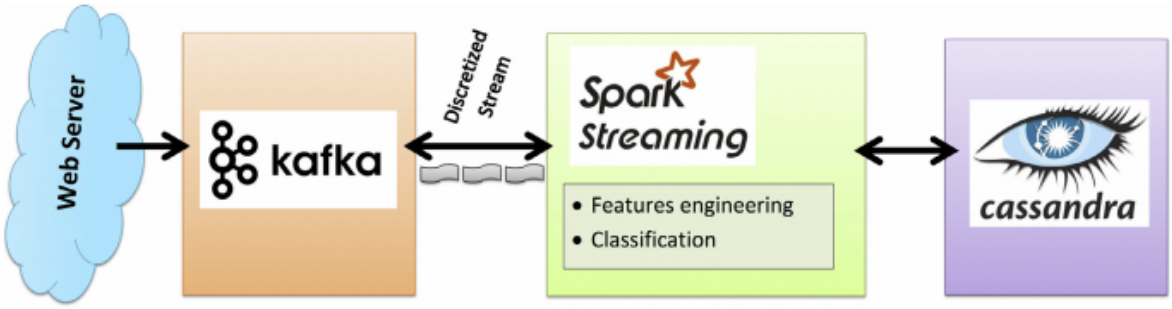

Figure 3.1: SCARFF Architecture as Introduced by [15]

For the implementation of the streaming analytics functionalities, the authors use Apache Spark [36]. Apache Spark is an in-memory map-reduce implementation that automatically distributes the computations among the available resources. An important key aspect of Apache Spark is the possibility to run batch and streaming analysis on the same platform. The data is divided into so-called Resilient Distributed Datasets (RDDs). Due to their distributed nature, the RDDs are able to recover automatically from a node failure. By splitting the data into RDDs, a mini batching is performed. This leads to a higher latency compared to Flink (focus on streaming and not efficient batching) which is one reason why the latter was chosen for the implementation in this thesis.

The streaming analytics engine is used to implement five different functionalities. First, Spark Streaming is used to ensure the basic flow of data. Secondly, data pre-processing is to be ensured. This step is mainly concerned with handling missing values and coding categorical values. The categorical values are converted into a numeric value. It is used describes the predefined probability with which the category is associated with a fraudulent transaction. These probabilities are calculated from historical data and are stored in a dictionary. [15] refer to the good experiences of the industry partner regarding the efficiency of such a pre-processing. The chosen method is a kind of cascade generalization approach [38] in which advanced, naive classifiers are used as input for more powerful classifiers. To counter the the possibility of concept drift in the procedure, they suggest updating the dictionary whenever a new batch of labels is received. The third area comprises feature engineering. Here, the required aggregates are to be calculated using the historical data in the Cassandra database [39]. These include the maximum, minimum, count and average of relevant numerical values such as the transaction amount. As a reference to their strategy for aggregating the relevant features, the authors refer to

the approach presented in [40]. Fourth, the online classification of incoming transactions based on the latest classification model is discussed. Once the classification is done, the system updates a dashboard that shows a priority list of transaction alerts based on the calculated risk. Finally, the last Section of the work is dedicated to storing the data and aggregates in a Cassandra table. This table is then periodically used for training the machine learning engine.

The experiments were subsequently performed on a cluster of 10 machines, each with 24 cores and 80GB RAM. A data set containing transactions from the last 40 days was used as a test. It includes slightly more than 8 million transactions from almost 2 million cardholders. The individual records each consist of 18 descriptive features and a fraud / non-fraud label. The feature engineering step generates 17 additional features from weekly windows. The Spark batch duration was set to 240 seconds and the data rate to 100 transactions per second. The tests were performed with 25, 35 and 45 Spark executors respectively. With a batch duration of 240 seconds and a rate of 100 transactions per second, this results in 24000 data records in one batch.

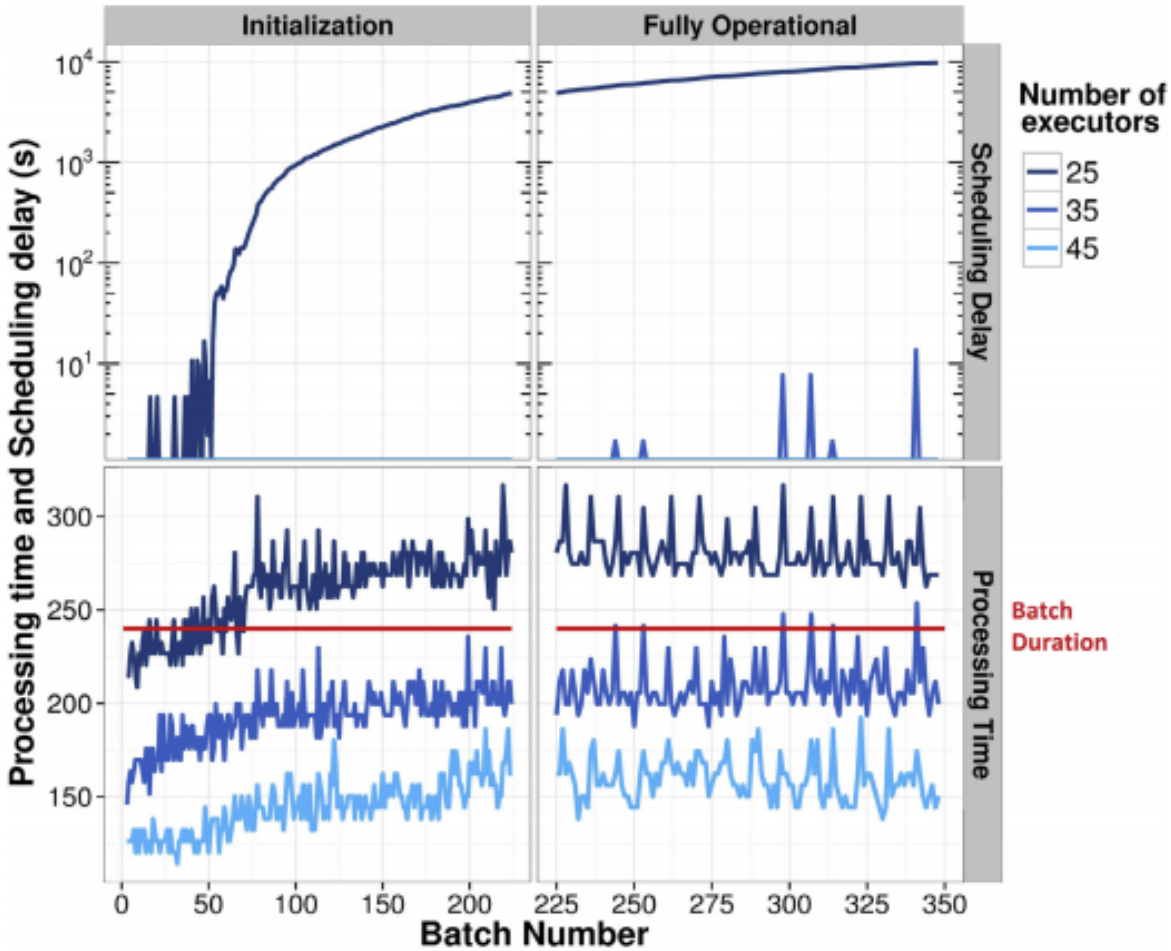

Figure 3.2: SCARFF Evaluation by [15]

An interesting characteristic is the splitting of the operations depicted in Fig. 3.2 into an initialization and fully operational part. This approach will also play an important

role in the present work. The streaming system first has to warm up and reach a steady state in which not only values have to be added to the state, but also values get removed. Up to this point, a continuous slowdown of the processing time will be noticeable, as the overall amount of data to process increases. As can be seen from the graph in Fig. 3.2, the processing time for 25 executors is slower than the batch time. This leads to an accumulation of data to be processed, which in turn can lead to a failing of the application as more and more data accumulates. When increasing the executors to 35 and 45, the performance is increased and no pile up should occur.

However, as noted by [15], increasing the number of executors beyond a certain amount does not significantly increase performance. As explained in the paper, this is due to the map-reduce process, which becomes too expensive after a certain point. It is stated that the effort of shuffling the data between the executors outweighs the gain of the divide and conquer approach. Internal parameter tuning is mentioned as a possible optimization approach. For example, it is possible to increase the batch duration time. However, this can lead to two potential problems. First, the algorithm is designed in such a way that a batch can only access the previous values at a time. If there are two transactions of the same card in a batch, the information of the first transaction cannot be used for the second transaction. This reduces the precision of the feature calculation. The second problem is related to a possible delay of the alerts that are generated due to the longer batch duration. However, this can be neglected due to the manual intervention of an agent in the fraud case. Finally, the authors conclude that their solution has shown to work robustly up to 200 transactions per second, which is an advance compared to the 2.4 transactions per second of the existing system. Also, the 200 transactions per second is not a hard upper limit, it can be increased by parameter adjustment. The precision of the classification was subsequently found to be 0.24, which means that for every 100 alerts reported, 24 were correct. The authors identified the interaction of various tools in a platform as well as the internal parameters, which have a major influence on the performance of the overall solution, as a problem. For the present work, processing the live data set in batches is certainly not an optimal solution. Results have to be delivered in a few milliseconds, which contradicts the rather large batch duration. Also, the required calculations as well as the window lengths and data sets are much more complex and larger in size. Thus, the performance of the presented approach is certainly improvable and not directly applicable for the needs of the present work. Nevertheless, the separation between a bootstrap and a live phase could be taken over as an idea for future work. This way, the initial roll-up of the history could be done in a batch mode. Afterwards, the results would be picked up by a live streaming job and the calculations would continue from this initial state.

3.1.2 StreamING Machine Learning Models: How ING Adds Fraud Detection Models at Runtime with Apache Flink

The work presented by [11] was developed in collaboration with the ING Bank [41]. The ING Bank has over 36 million customers in over 40 countries. 9 million of these customers are located in the Netherlands where ING handles up to 1 million transactions per day. [11] state that fraudsters are using increasingly sophisticated strategies to obtain money.

The emergence of such new strategies pose an increasing amount of problems for rule-based systems. These systems only work if one knows which fraud patterns to expect in the first place. Possible new patterns can be overlooked. An example of such an attack is the attack of a group called **Carbanak** [42], which caused Automated Teller Machines (ATM) to dispense money at a set time. The money could then be collected by accomplices. This shows how important it is for financial institutions to have risk systems in place that are able to react immediately to previously unknown threats. [11] identified three goals that their application should fulfill. First, it should be possible to support a range of machine learning models. It should support rule based alerting as well as scoring based machine learning models. Second, the architecture should be built in such a way that it can be deployed in different countries and departments. Even if the underlying infrastructure differs. Finally, business users should be able to make changes to the models and update them. No downtime or re-deployments should be necessary to implement new rules.

The **support for a range of machine learning models** is achieved by separating them into an online and offline domain. As depicted in Fig. 3.3, analysts work with tools such as Knime [43] and Apache Spark [36] to create fraud detection models. These models are then streamed into Flink and used to score events in real time. To transfer the models between the off- and online environment the Persistent Model Markup Language (PMML) [44] is used. It is described by [11] as an XML based format which allows to store machine leaning models. It is a de-facto standard and is offered as an export format by many tools that data scientists use for model training. In the online environment, these models are then streamed as Kafka events into Flink control streams and broadcast to a model scoring operator. In the scoring operator these models are parsed into objects and stored in the state. The feature sets are then passed to the models and the result is output to a stream.

Figure 3.3: On- and Offline Environment as Shown by [11]

The **support for deployment to different environments** is ensured with the Kafka-in and Kafka-out architecture which is depicted in Fig. 3.4. This means that the input comes from various sources such as the banking website, the banking app or other applications. These events are consumed by the Flink job and transformed into features. These features

then serve as input for the model. The problem here is that event schemes change and the Flink job cannot be redeployed for every change. As a solution to this problem, a second Flink job was added. According to [11], the so-called pre-processor has three requirements that it should cover.

- Events should be filtered out which are not needed for the feature extraction.
- Multiple raw events are aggregated. An example is the aggregation of web clicks to a single business event.
- It consolidates events from all input topics with different schemes into a single business event topic with a common scheme.

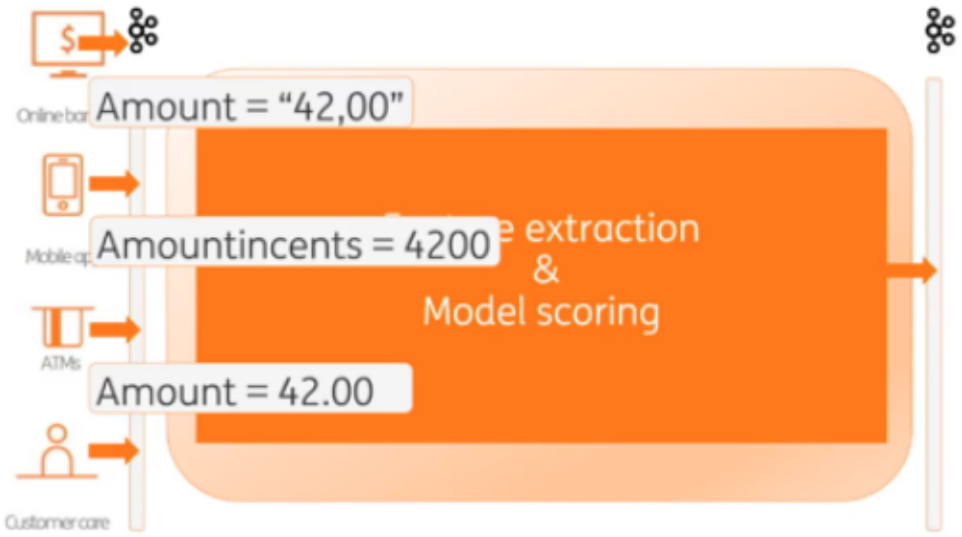

Figure 3.4: Decoupling of Pre-Processing and Feature Extraction as Shown by [11]

The goal of **allowing instant changes without downtime** is to enable normal analysts to fix potential vulnerabilities in rule-based models without having to redeploy. However, it is not only necessary to be able to deploy new PMML models. Feature extraction must also be user definable. As a solution for this a Domain Specific Language (DSL) [45] was introduced. It allows a user to describe the feature extraction process and to deploy it together with the PMML file. The DSL offers the option to defined which information must be stored by the event. Derived from this, it is also determined which key is extracted. Furthermore, it enables to define how the feature is calculated based on the key and the event and how the mapping between the feature set and the model is established.

The last step to be clarified is the ingestion of the models and definitions into the correct operators. [11] describes the process as follows. First, users save their files to the version control system. Then they are pulled by a process and pushed into a specified Kafka topic. From there they are picked up by an operator, parsed and forwarded via broadcast to the other relevant operators.

The approach presented includes some exciting concepts. The separation into an on and offline environment will also be relevant for the present work at a later stage. The decoupling of modeling and more complex development work makes sense from a business perspective. Furthermore, the second and third topic contains important inputs to be considered for the present project. The magnitude of the processed data is certainly larger in the ING case, although it is not disclosed how many calculated features are ultimately involved. However, this is to be expected due to the business secrecy. Next, a closer look will be taken at the split join pattern also included in [11]. It will find an immediate application in the present work.

3.2 Flink Split Join Pattern

In 2017, [41] introduced a split join pattern for Flink in the work described in 3.1.2. [11] describes the problem that an event in Flink can potentially contain multiple foreign keys, such as customer number, card number, or account number, which should be aggregated independently but merged back together at the end. The same is true for different aggregation or window types that are different but should form a single output record. Another use case is the combination of the aggregated values with the initial event at the end of a pipeline. The problem is that in Flink the so-called keyed stream is based on a single key. Thus, information can only be enriched based on a single key. The solution elaborated by [11] is to extract the different keys from the event and create an individual record for each key. Based on the extracted values of the different keys they end up on the nodes which contain the state of the corresponding keys. The problem is that the different records of the same event are now on different nodes of the cluster. However, the desired state would be that they are all on the same node to subsequently assemble the feature set. [11] proposes to initially assign a random event id to each event and attach it as payload. Once all aggregations are done on the foreign keys, the events are keyed again based on the initially assigned event id. Since all formerly split elements of the event now end up on the same node again, they can be combined. This process is visualized in Fig. 3.5. When splitting, the streams are divided into different sub-streams. The subsequent join operator then waits until a result has been received from all inputs. Finally, it forwards the combined record as output.

The pattern works without issues for smaller areas in a job where all extracted keys approximately have the same duration until a record is emitted. However, if the ranges are extended, it can happen that events pile up in the subsequent join operator which can lead to a congestion. In the present work this was especially the case when different window types were combined. It could happen that in the worst case an event to be joined had to wait for the maximum window length of 392 days. Another important topic for the present work was the window slicing introduced by [10]. It introduces a two-stage algorithm for windowing that relies on combining small slices into bigger time-frames.

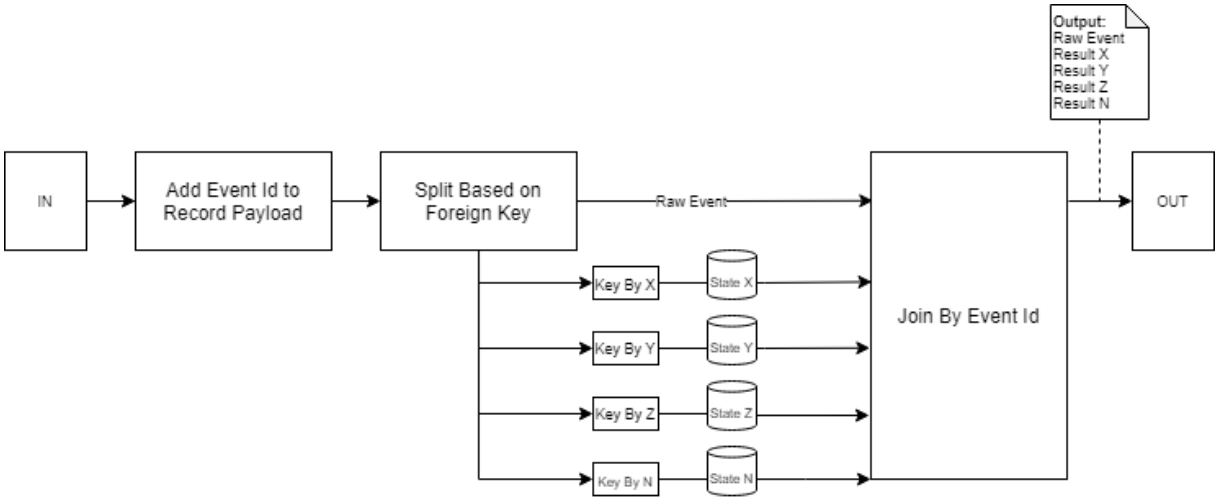

Figure 3.5: Split Join Pattern as Introduced by [11]

3.3 Window Slicing

As pointed out in the previous sections, the work of [10] and [46] highlights that overlapping or concurrent aggregations of windows lead to inefficiencies due to redundant computations. This has also been noted by [47, 48, 49] and corresponding issues have been created for the Flink developers. Accordingly, this work aims to provide a general solution that not only improves performance but also improves applicability to window types, aggregation functions, and out-of-order processing. [10] emphasize that the database community is working on aggregation techniques for overlapping windows [50, 51]. The focus is on calculating partial aggregations for overlapping parts and then pulling them together. According to [10] these techniques are not widely used for two reasons. First, the literature for streaming window aggregation is fragmented and second, each technique has different assumptions and limitations. As a consequence, it is not clear to researchers and practitioners under which conditions which technique can be applied.

Therefore, the envisioned solution should be applicable to different types of aggregation workloads. At the same time, the operator should be as efficient as specific technologies that support only selected workloads. As a first contribution, the paper provides a workload characterization of existing specialized window aggregation types. These characteristics are:

- window types (*e.g.,* sliding, session, tumbling)
- windowing measures (*e.g.,* time or tuple-count)
- aggregate functions (*e.g.,* associative, holistic)
- stream order

The classification of existing techniques based on their concepts and use cases is achieved through an extensive literature research.

The second contribution of the paper is related to stream slicing. It was first introduced by [46] and describes the splitting of a stream into smaller parts. The so-called slices. These partial aggregates are then shared by all queries and window types, allowing more efficient storage and processing of windows. The general idea of the stream slicing introduced by [46] is visualized in Fig. 3.6. It shows the division of a stream into different slices, which are then combined into windows. Individual slices can be reused for different windows.

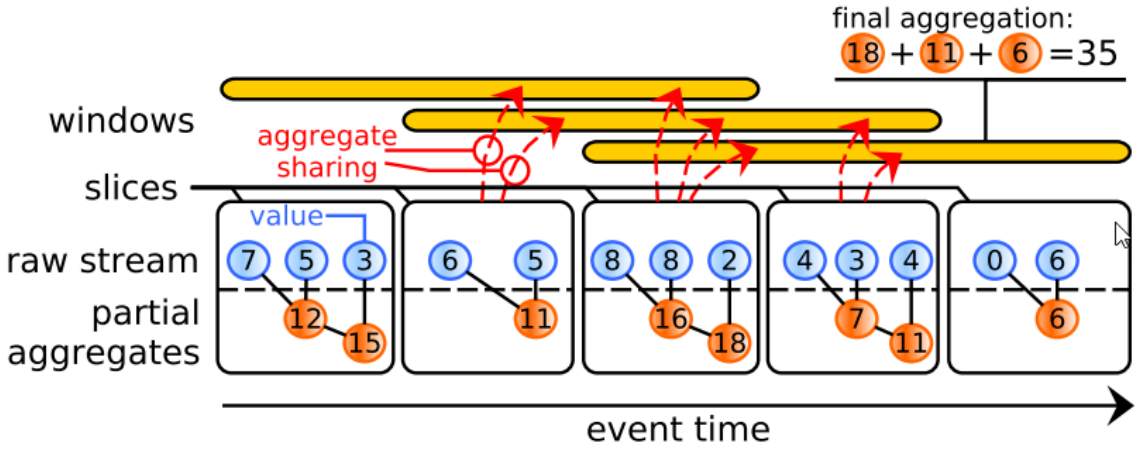

Figure 3.6: Stream Slicing Introduced by [46]

The general window slicing of [10] does not only provide the slicing itself. Specific workload character styles determine the costs of operations and how often they are executed. Depending on the character style, either the individual tuples are cached or the number of slices that are created, saved and recalculated are minimized. The algorithm automatically decides how to handle the incoming stream most efficiently and adapts its internal handling of slices and aggregations accordingly. The general architecture of stream slicing is visualized in Fig. 3.7 and works as follows. Users define their queries in a high-level language such as stream SQL or a functional API. The query translator then analyzes the structure of the query and the characteristics of the input stream. Based on this, the slicing technique automatically adapts to the given characteristics. It is also decided whether individual tuples have to be cached or omitted after aggregations. Finally, when a new query is submitted to the aggregator, a re-evaluation of the applied algorithm is carried out.

While evaluating the solution the authors showed that their window slicing could increase the throughput of window discretization and aggregation by an order of magnitude. A dataset of the Grand Challenge 2013 [52], which contains sensor data generated during a soccer match, serves as the evaluation basis. It contains about 15000 data points per second. For a match duration of 90 minutes this results in a data volume of 80'000'000 records. As aggregation functions, a distinction is made between simple summation and more complex, holistic aggregations such as minimum/maximum and first/last value of a window. Another difference which must be pointed out is the difference of the data and key characteristics. In the work of [10] mainly queries with a small key space and extensive data sets are used. However, for the planned aggregations in the present work (especially for scenarios 1 and 2), a large key space with sparse data is expected. As a

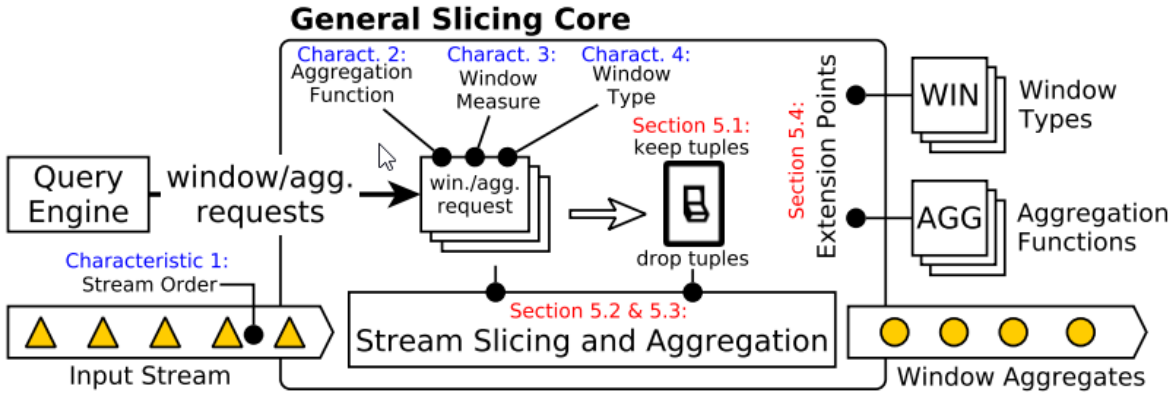

Figure 3.7: General Stream Slicing Architecture Introduced by [10]

result, any performance gains from pre-aggregations are less significant. Additionally, the implemented solution does not work with a state backend but only with an in-memory structure. Furthermore, the Flink version used is Flink 1.3, which is outdated compared to the current release 1.12. Therefore, a direct adoption of the library is not possible. However, slicing itself remains an important component which is also of great relevance for the present work. If all windows are handled individually, this leads to a huge amount of duplicated events in the different windows with 1485 envisioned features. A window slicing and sharing is therefore inevitable. Even though window slicing already offers great advantages, storing all events in the memory is not suitable. Therefore, another state backend had to be used. The performance of the RocksDB state backend was always an important question. Parameter tuning plays a major role for optimization. The next Section is consequently dedicated to this topic.

3.4 RocksDB Parameter Tuning

In order to understand the tuning of the RocksDB state backend, the general principle of operation must first be known. RocksDB [21] is described by [35] as a key-value store implemented as a log-structured merge tree (LMS-tree) [53]. When RocksDB is used as a state backend, the state to be stored is stored as a serialized byte string either in the off-heap memory or the local disk. [35] describes the process of adding a keyed state as follows. The entry to be added is mapped to a column family which can be compared to a table in a traditional database. As described above, the keys and values are stored as serialized byte strings. This means that a de/serialization must take place for every read and write operation. This leads to reduced performance compared to the in-memory state backend of Flink.

However, RocksDB makes up for the performance losses with its other advantages. It is not affected by the garbage collector and is the only option that allows incremental Checkpointing. The size of the state is also only limited by the size of the hard disk and not by the memory. Fig. 3.8 depicts the basic functionality of the read and write

operation in RocksDB. A write operation stores its data in the currently active memory table. When such a memory table is full, it is converted to a read-only table and replaced by a new one. The read only tables are then periodically stored by a background process in read only files sorted by key. These unchangeable tables (SSTables) are then compacted in the background by a multiway merge.

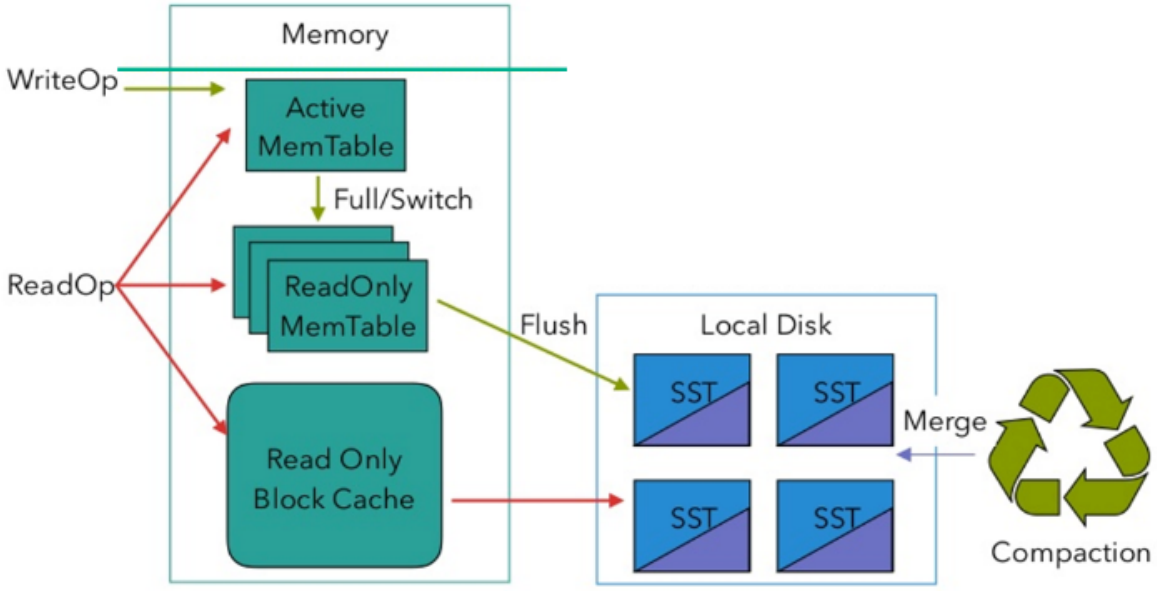

Figure 3.8: Basic Read/Write in RocksDB According to [35]

Read operations in RocksDB access the first active memory table that responds to a request. If the key is found, the read only memory table is searched from the most recent to the oldest value. If the key is not found, the operation accesses the SSTables. They are either fetched from the block cache (*i.e.,* uncompressed table files) or, in the worst case, read from the disk.

It seems clear that it makes a big difference in which structures the searched values are stored and that it is not good from a performance point of view to have to access the disc for read operations. The list of parameter tuning options mentioned in [35] is not exhaustive, but it lists three important points that can be taken as a starting point to make resource utilisation more efficient.

- **state.backend.rocksdb.writebuffer.size** controls the maximum size of the memory tables in RocksDB. This setting will also increase memory usage. According to [4] this size is set to 64 MB by default. Adjusting this parameter can reduce the impact of write operations. However, at the same time pressure is built up on the upper levels as larger amounts of data have to be handled in the subsequent compactions. Therefore, it is recommended to adjust this parameter only in combination with the compaction parameters. These parameters regulate, among others, the interval at which the compactions are carried out.

- **state.backend.rocksdb.writebuffer.count** controls how many memory tables are held until they are flushed to the local disk as SSTables. This is the maximum number of read only tables in the memory.
- **state.backend.rocksdb.writebuffer.number-to-merge** is mentioned by [4] and controls the minimum number of memory tables it needs before performing a flush. By default this value is set to 1. If it is set to two, a flush is only triggered when two such immutable tables are available. This allows more changes to be noted before a flush. Tests carried out by [4] have shown that the best performance is achieved when setting the value to 2 or 3.
- **state.backend.rocksdb.block.blocksize** represents the actual block size. This is set to 4KB by default. According to [4] it is recommended to increase this value to 32KB in productive environments. However, this setting should be increased together with the **block_cache_size**. Otherwise, the number of blocks in the cache is reduced, which in turn has a bad effect on the read times.
- **state.backend.rocksdb.block.cache-size** represents the maximum number of cached and uncompressed blocks in the memory. As this number grows, so will the memory size, but the advantage is that less disk access will be needed. In addition, a specified level of memory can be set. The size is set to 8MB by default. However, [4] recommends setting this size to 128 MB or even 256 MB to lower the needed time for reading operations significantly.
- **state.backend.rocksdb.thread.num** controls the paralellism of background flushing and compaction. The read and write operations are carried out directly in RocksDB. However, the flushing from memory to the local disk and the compacting are carried out in background threads. In a machine with multiple CPUs, [21] recommends increasing this parameter as it is too low by default.

Another parameter that has a direct influence on RocksDB is the Flink managed memory which is allocated as native memory (off-heap). It can be set by configuring the `taskmanager.memory.managed.size`. As stated by [54], RocksDB limits the native memory allocation to the size of the managed memory. Therefore, increasing this parameter also increases the memory RocksDB has at its disposal to store state. In the worst case scenario, if the RocksDB memory control is disabled, `TaskManagers` can be killed in containerized deployments when RocksDB allocates more memory than the requests container size.

With the help of these parameter adjustments, it should be possible to observe an increase in the performance of the RocksDB state backend.

The last Chapter provided an introduction to related work and concepts. Especially the slicing technique and split/join pattern will find direct application in the present thesis. Next, chapter 4 is the dedicated to the design decisions taken and elaborates the optimization approaches chosen.

Chapter 4

Design

The following Chapter presents the design decisions for the optimization strategies to be implemented. The problems of the baseline implementation mentioned in Section 2.5 are addressed and the potential solutions discussed. It is important to note that the optimization strategies differ depending on the key cardinalities and the overall complexity of a given Flink job. A slicing technique does not have the same significance when applied to windows with sparse, respectively extensive data. Also, an I/O optimization is more efficient when there is a large number of parallel tasks generating I/O. The remainder of this Chapter is structured as follows. First, a measurement strategy is designed to identify the effective latencies and congestion in the existing system. Second, a solution for the high number of windows and keys in the state backend has to be found. In the third Section, the I/O between the tasks as well as the shuffle operations between the individual operators are looked at. Lastly, approaches to optimize the used algorithms will be introduced. These include the further reduction of the number of parallel window aggregations, the handling of streams with different output behaviour as well as the continuously sliding window algorithms.

4.1 Metrics

Metrics are of critical importance when assessing the performance of a streaming system. To make the measurement of values in the individual pipelines as convenient as possible, a systematic approach is used. Fig. 4.1 visualizes an example job which is used to calculate three different windows and combine them with the raw event. The goal regarding the metrics is to measure the complete duration of the window aggregation and the split/join operation.

A distinction is made between inter and intra operator measurements, which must meet the following requirements:

- Perform measurements on `ProcessFunction` level. Detailed insights shall be provided regarding the duration of single operations inside the functions.

- It should be possible to measure how much time individual events spend between operations when being transferred or buffered.

For the second requirement, the latency measurement provided by Flink cannot be used directly. As described in Section 2.4.8, it measures latency by sending a special `LatencyMarker` event through the pipeline. Thus, it measures the pure I/O time between operators, but not how much time is used inside of the operators for computations.

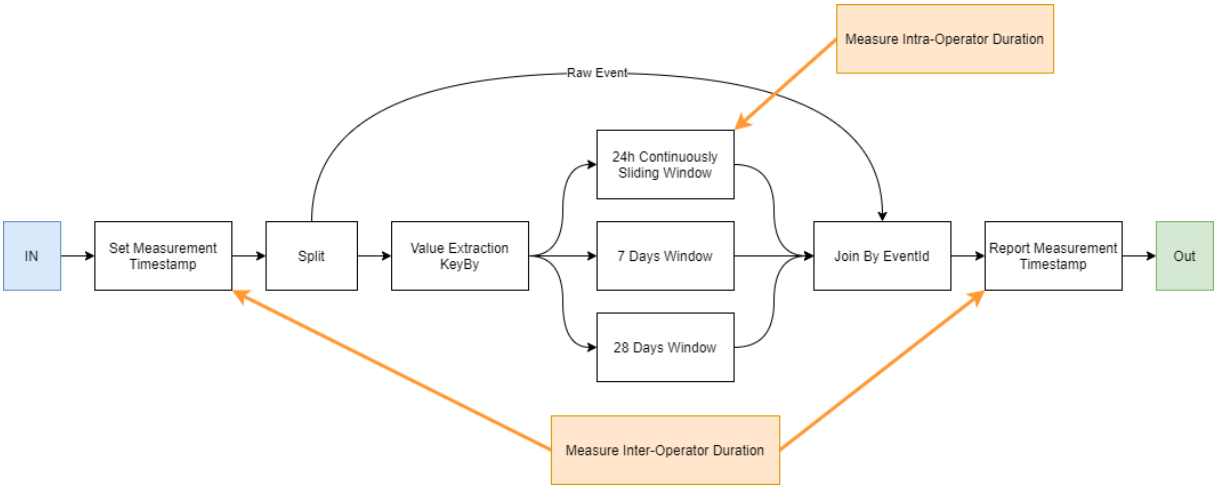

Figure 4.1: High-Level Overview of The Two Additional Measurement Types

In Section 2.4.8 the different metric types were listed. For the additional metrics to be implemented, histograms are used. These provide values per time unit divided into confidence percentiles. Thus, statements can be made about the probability that the values do not exceed a certain limit.

An interface needs to be implemented for the detailed measurements in the `ProcessFunction`. It should offer the possibility to initialize the measurement with histograms for pre-defined categories. The categories can be defined by the developer and passed as a parameter at initialization. A separate histogram is then created for each entry in the list. To start measurements for a category, the function `startCategory(category: String)` will have to be called. For example, `startCategory("StateInteraction")` can be used to start a measurement of a state interaction (*e.g.*, read, write, delete). This time measurement is subsequently interrupted by the next call to `startCategory("somethingElse")`, which in turn starts the new category called *"somethingElse"* If the category *"StateInteraction"* is started a second time, the newly measured value is combined with the existing one. Thus, a holistic picture of the elapsed time of a category in a function is achieved. Finally, the measurements must be terminated with a call to the `report()` function. This call closes the currently active category and updates the histograms with the accumulated values.

The second type of measurement should make it possible to measure the duration between two points of a job. For this purpose, each event is enriched with a timestamp. This timestamp can be initialized and updated with the help of a `map()` function. At a desired point, a `ProcessFunction` implemented specifically for this purpose can be used to read

this timestamp and write it to a histogram. Another version of this measurement is the *HotTrack* measurement. An `ingestionTimestamp` is set on each record during the incorporation of the corresponding event into the pipeline. The value corresponds to the current time in nanoseconds. If this metric is of interest, a `ProcessFunction` can be inserted at the end of the job, which extracts the ingestion timestamp, subtracts the current nano time and stores it into a histogram. The exact details of the implementation are explained in Section 5.1.

In this Section, it was described how the additional measurement methods will be approached. Consequently, this Chapter will proceed with the optimization strategies regarding the operational state.

4.2 Operational State Optimizations

The optimization of the operational state will be divided into two areas. First, possibilities are elaborated how the number of parallel windows as well as active keys in the state backend can be reduced. This is due to the fact that a larger key and data space has an influence on the performance. In the second step, the potential combination of the RocksDB state backend and a manually managed Heap data structure for state storage depending on the cardinalities of certain keys is elaborated.

4.2.1 Reduction of Number of Parallel Windows

The reduction of concurrently active windows is accompanied by a reduction of the key and data space. The fewer windows are kept active, the fewer resources are consumed. This Section strongly parallels Section 4.3.2. By reducing the number of concurrent windows, the number of active tasks is also reduced, which in turn affects the latency caused by the I/O.

In the baseline implementation, windowing was implemented as shown in Fig. 4.2. For the basic features, three different aggregators are needed. These are the statistical momentum (*e.g.,* sum, mean, standard deviation, skewness), the unique count as well as the oldest/recent aggregates. These aggregation functions have to be repeated for each key domain and window length. Fig. 4.2 shows a section of the aggregation by card id and the required operators for a 7-day sliding window. This pattern is repeated multiple times. First, one such block is needed per window length. Thus, there are at least four (*i.e.,* day zero, 7-days, 28-days, 392-days) such blocks per aggregation function (*e.g., statistical momentums*, *unique count*, *oldest/recent*). This number is multiplied by the number of key domains. Finally, combined with all associated operators, an execution plan with approximately 1800 nodes results.

In Fig. 4.2 it can be seen that each window has its own separate area in the job. This means that in each of these blocks all events must be stored and aggregated separately. Due to the business requirement, daily sliding 7, 28 and 392 day windows are used. Consequently, every day new windows are created to which the records of up to 392 days in the past are

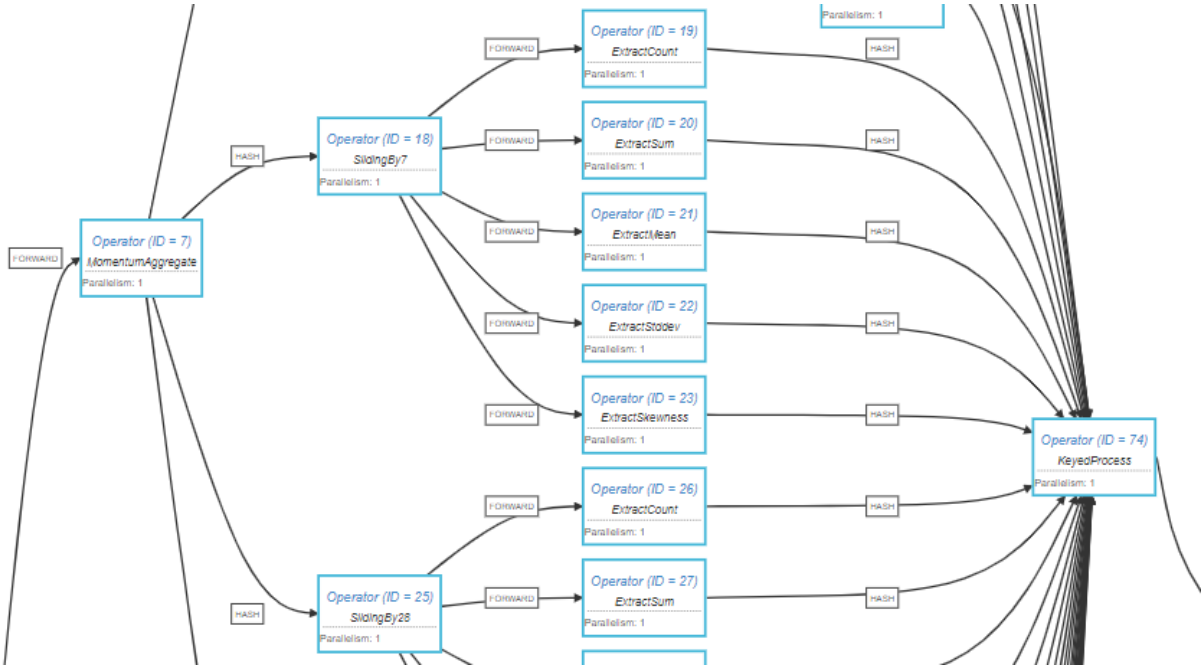

Figure 4.2: Momentum Aggregate Per CardId With a 7 Days Sliding Window

assigned to. For scenario 1 (*i.e.,* key by card id), with four different window lengths and three different aggregation functions, 12 independent, parallel windows are created on a daily basis. For scenario 2 this number becomes even larger due to the multiplication of keys.

The proposed solution for this issue refers to the fact that a two-stage windowing algorithm can be used. Instead of implementing daily sliding windows, daily tumbling windows can be created in a preliminary stage. These daily tumbling windows are then aggregated to larger windows in a second stage. Thus, individual windows (*i.e.,* containing up to 392-days) are no longer created each day, but only one window per day per aggregation function and key domain. This one window is then used by the second aggregation level to form the 7, 28 and 392 daily sliding windows. Fig. 4.3 compares this approach with the baseline implementation.

Another potential optimization strategy related to operational state is the switch between the RocksDB state backend and a manually managed Heap data structure while still being able to use the Checkpointing API provided by Flink. This approach is introduced in more detail in Section 4.2.2.

4.2.2 Manually Managed State on Heap Data Structures

For certain keys it is feasible that state is kept in-memory. This speeds up the access times as it is not necessary to access a state backend for each record. As is should still be possible to benefit from the Checkpointing API provided by Flink, the `CheckpointedFunction` interface has to be implemented. It forces the developer to implement the two functions

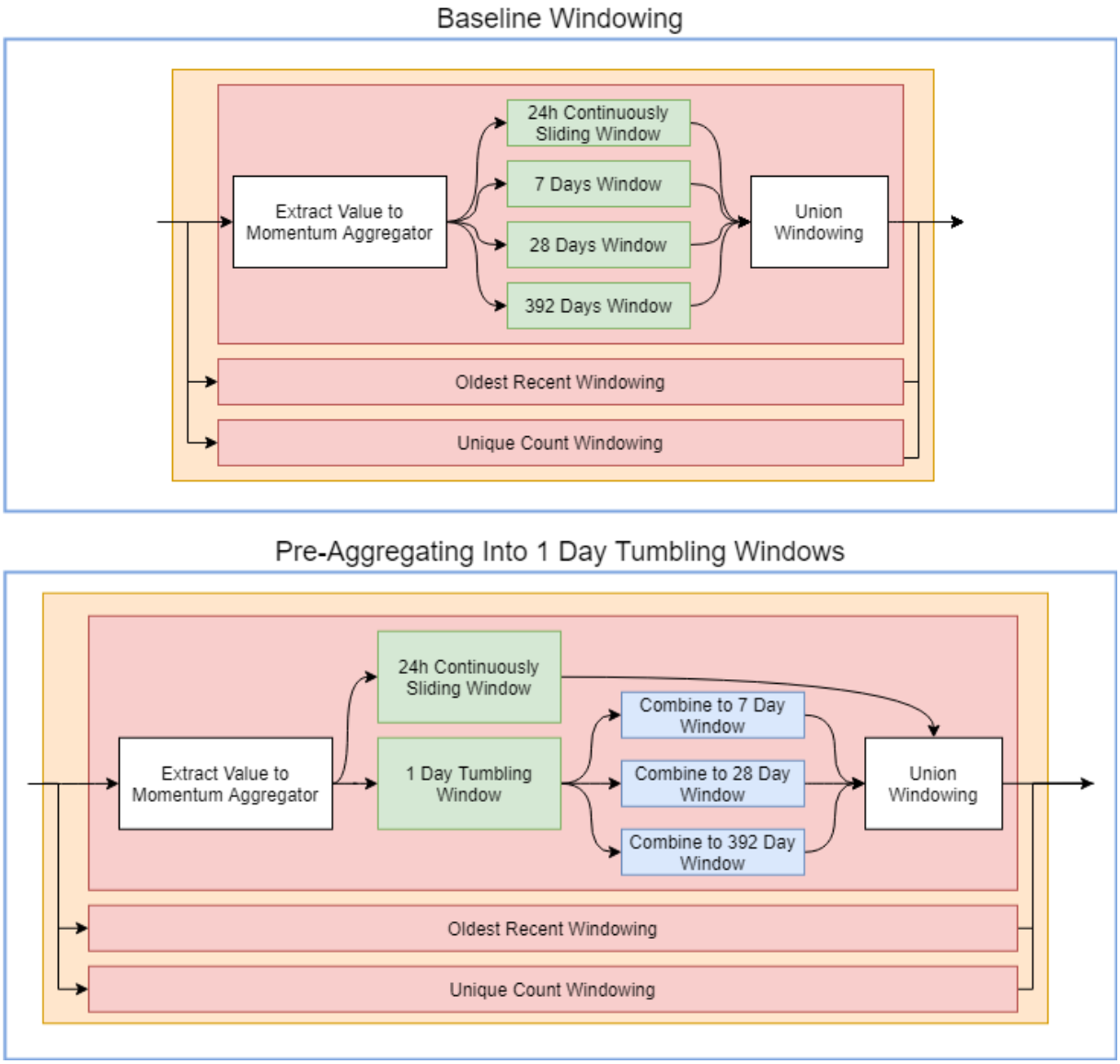

Figure 4.3: Comparison of Baseline And Two Stage Windowing

`snapshotState` and `initializeState`. The `Context` provided in those functions allow access to the state backend. This way is not only possible to store and initialize the operator state, but also the keyed state. A `HashMap` structure can now be created on the Heap. All operations involving intermediate state will work with this Heap data structure. As soon as the `snapshotState` function is called, the data inside of the `HashMap` structure is transferred into a `List` and stored in the state backend. The detailed implementation and the chosen state primitive is further described in Section 5.2. In case of a fresh start, the Heap structure is initialized. In case of a restart, the Checkpointed data stored in the state backend is copied to the in-memory structure.

Important for the implementation is the knowledge about the key and data cardinalities. When storing state in an in-memory structure, special care must be taken to ensure that the data does not exceed the allocated resources. In a first step, this approach can be implemented when the developer is sure that the data will fit inside of the memory. In a

second step, it can be considered to count the number of records and the number of keys with the help of a metric counter. Until a certain threshold is not reached, data is saved in the Heap structure. As soon is it grows too big, an automatic switch to the RocksDB state backend is carried out.

The adjustments mentioned in the last sections are planned with respect to the operational state optimizations. The next Section will focus on potential solutions to optimize the I/O.

4.3 I/O Optimizations

After a detailed analysis of the baseline implementation, it was found that not only the high number of operators and tasks, but also their linkages impact the overall performance. As already explained in Section 2.4.7, there is a distinction between the hash, forward and rebalance partitioners, which represent the links between the individual operators. They are used to determine how data is forwarded from the instances of an operator to the next one. With a forward partitioner, the data is passed from one instance of an operator to the exact same instance of the following operator. If a hash partitoner is present, they are distributed among the instances of the subsequent operator based on their hash value. In case of a rebalance, any (potentially skewed) data is distributed with a round-robin scheme (*cf.* [31]. The use of a hash or rebalance partitioner between the operators causes the data to be de/serialized. However, this procedure is not necessary in all cases, which is used for the following optimization technique.

Another topic is the de/serialization of data between individual tasks. The conditions described in Listing 2.8 indicate when an existing operator chain is interrupted and a new task begins. For the baseline implementation, the condition `downStreamVertex.getInEdges().size() == 1` is most important. It states that an operator may only be chained if it has one input at a time. However, this is not the case when using the split join pattern in the baseline implementation.

In the following Section, the approach to remove unnecessary hash partitioners is described. Afterwards, the design decisions made to introduce the approach to extend the task chaining is shown.

4.3.1 Transforming Redundant Shuffle Operations

In Flink, the return type after applying a transformation on a `KeyedStream` is a `DataStream`. If two successive operations (on the same key attributes) are to be performed on a `KeyedStream` (*e.g.,* two-stage windowing algorithm), the stream must be keyed again. This is done even though the stream would actually already be correctly partitioned. However, if the key domain is not changed, no re-partitioning is necessary. This behaviour can be remedied by the Flink function `reinterpretAsKeyedStream`. It allows to reinterpret a `DataStream` into a `KeyedStream` without performing the effective `keyBy` operation. However, the documentation [8] states the following point. It is important to note that for

each partition of the base stream all keys of the records must be partitioned exactly the same as if they were created by a `keyBy`. This will especially be important for handling the delayed streams introduced in Section 4.4.2.

Fig. 4.4 shows an execution plan with a large number of redundant shuffle operations (*i.e.,* hash partitioners). The shuffles that originate from operator 7 are particularly noticeable. This behaviour is shown as a `keyBy` is executed right after operator 7. A potential solution is to add a `map` operator before operator 7 that does not apply any logic to the stream but just forwards the records. Also, the `keyBy` has to be moved just after this new `map` operator. Thus, the `keyBy` moves one position to the left. Then operator 7 is executed which transforms the stream from a `KeyedStream` into a `DataStream`. This is due to the fact that a rehash operation is a virtual operation in Flink, which is in effect applied to each input of successive operators. The `map` reduces the number of successive operations to 1 before the fan-out to the individual successive operations. Afterwards, a call to `reinterpretAsKeyedStream` can be used to transform the `DataStream` into a `KeyedStream` again. Finally, all the redundant hash partitioners are omitted.

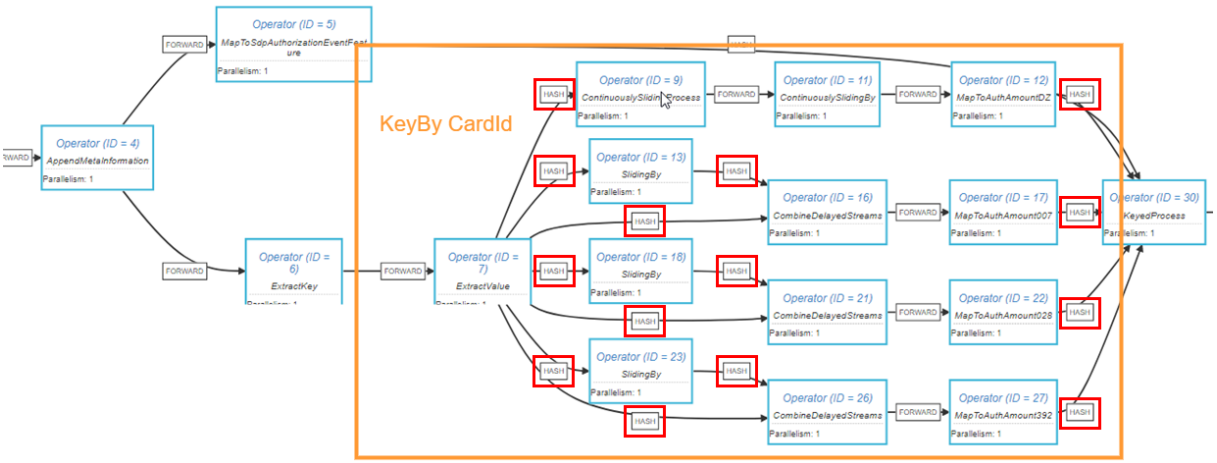

Figure 4.4: Shuffles That Can be Transformed Into Forwards

It can be stated that a `keyBy` and thus a shuffle operation only makes sense if the key domain changes. Otherwise, `reinterpretAsKeyedStream` can be used if a `KeyedStream` is required for further operations. Although the shuffle operations are eliminated, it still happens that different operators cannot be chained. In some cases, this behaviour can have a bad influence on the performance. As a result, the next Section is dedicated to the second problem domain in the I/O area, the task chaining.

4.3.2 Task Chaining

The overall idea of the extension of the task chaining as described in Section 2.4.7 is to reduce the de/serialization and I/O effort by reducing the total number of tasks in a Flink job. Due to the conditions described in Listing 2.8, it is not possible to chain operators

with two inputs. Not even if the chaining policy is set to `Always` in the configuration and a forward partitioner is present.

The goal of this experiment is to prevent tasks from being subdivided as shown in the `JobGraph` depicted in Fig. 4.5. It is a representation of the execution plan drawn in Fig. 4.4. For small jobs, this optimization may seem trivial. However, if the execution plan of the baseline implementation with 1800 nodes is taken as a reference, the number of tasks increases rapidly and each new task means de/serialization and I/O effort.

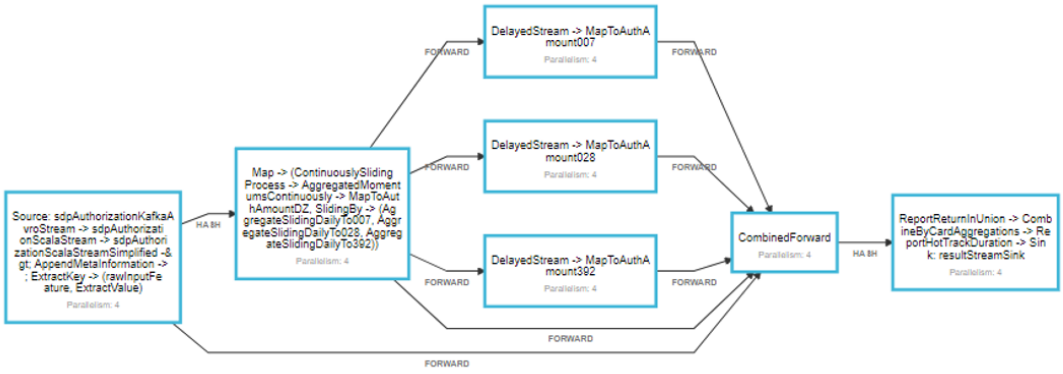

Figure 4.5: Splitting of a Job Into Separate Tasks For Operators With Two Inputs

For the implementation of the task chaining extension, some adjustments have to be made in the framework itself. The goal is to have the option of specifying an individual task chaining strategy using the strategy pattern when creating the `JobGraph`. As stated in Chapter 2.4.7, the JobGraph creation is on the second level of the transformation of a program to an execution plan. Due to the closed architecture of Flink in this region, it is essential to develop a separate implementation of the first and second level of the transformation. The new implementation of the environment should then provide a function called `setChainingStrategy`, which allows to change the chaining via configuration parameters. This should also enable efficient testing and comparison of the different strategies. Fig. 4.6 visualizes the desired outcome of the different chaining strategy implementations.

The last Section presented the optimization strategies designed to improve the I/O. Consequently, the next Section is dedicated to the algorithmic optimizations.

4.4 Algorithmic Optimizations

The last area to be addressed is the optimization of the used algorithms. First, the concept of the reduction of the number of parallel windows presented in Section 4.2.1 will be deepened. Afterwards, the handling of delayed streams is discussed. Finally, the concept of the continuously sliding window algorithm, which is not provided out of the box by Flink, is elaborated.

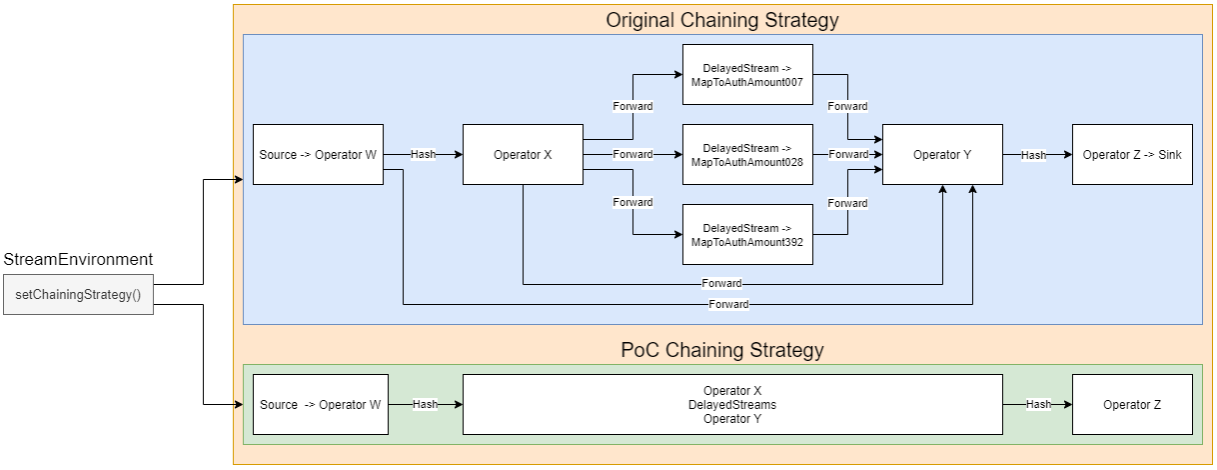

Figure 4.6: Differences Between `JobGraph` With Original and PoC Chaining Strategy

4.4.1 Reducing Redundant Operators

This approach aims at reducing the high number of parallel operators needed for windowing. The beneficiary effects of this optimization are manyfold. On the one hand, parallel operations on the state backend can be reduced. On the other hand, the I/O can be reduced due to the overall smaller number of operators and tasks. Also, redundancies in the code can be prevented by generalizing interfaces.

The operations that are performed on the individual key domains are mostly the same. The only difference is the data type of the extracted key. Therefore, the approach should allow a transformation of a key into a generalized form. With the generalization, all needed key types and combinations (*i.e.,* scenario 1 and 2) can be specified. The algorithm can be summarized as follows:

1. Extract all keys (per key domain)
2. Convert the keys to common data type
3. Attach the key to a separate copy of the original event
4. Forward (key, event)

By using this process, all key types can be handled in the same pipeline.

However, parallel processing of the windows (*cf.* green boxes in Fig. 4.7) still takes place. In the current pipeline, there are three operators used for windowing (*i.e.,* 7, 28, 392 day windows), which generates de/serialization overhead and state interaction per window algorithm.

In a second step, the merging of the window aggregations is to be considered as depicted in Fig. 4.8. The continuously sliding windows are excluded from this optimization because they are not based on the daily tumbling windows.

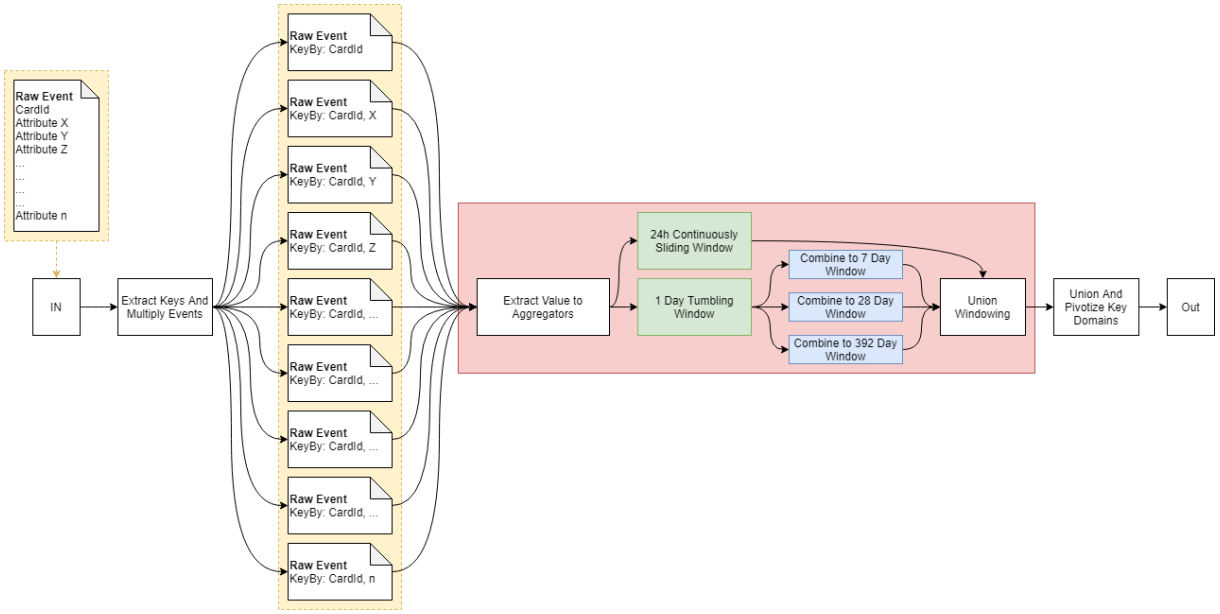

Figure 4.7: Splitting of an Event Into Multiple Events With a Generalized Key

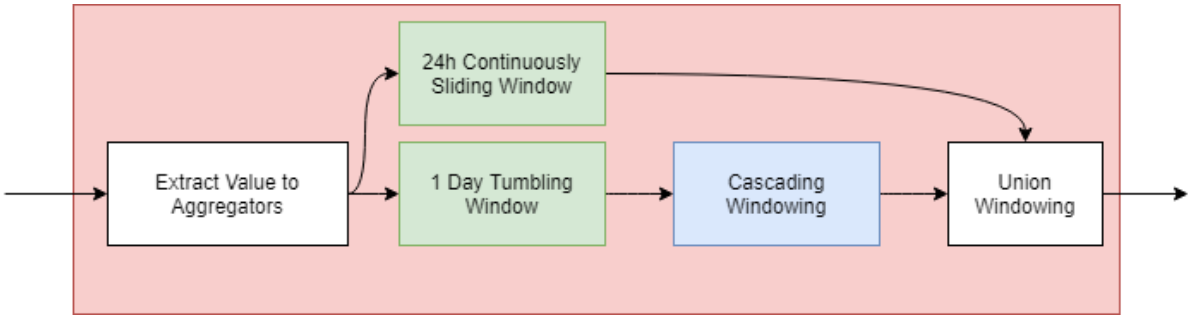

Figure 4.8: Combining The Different Operators Used for Windowing

It should be possible to pass any number of days to the algorithm, reflecting the windows that need to be aggregated. The operator should then automatically calculate the individual time intervals. This process is to be carried out serially. First, the records stored in the state are divided into distinct time intervals. Afterwards, groups are formed from the smallest to the largest interval and incrementally rolled up.

For the example visualized in Fig. 4.9, 7, 28 and 392 day windows are to be created. The events delivered so far are shown on the timeline. The records are divided into distinct groups according to their time interval. This is done as larger intervals always contain the smaller ones and thus, a roll-up of the these values can be carried out. After an interval has been calculated, the corresponding window result is emitted. The windows of the cascading window aggregation should be configurable. Intervals other than 7, 28, 392 days should be definable without having to change the code.

The great advantage of this method is that the windows are calculated in one go. Thus, the corresponding `MapState` containing all events of a key is iterated only once instead of once per window aggregation. Additionally, the aggregation of the entries is more efficient.

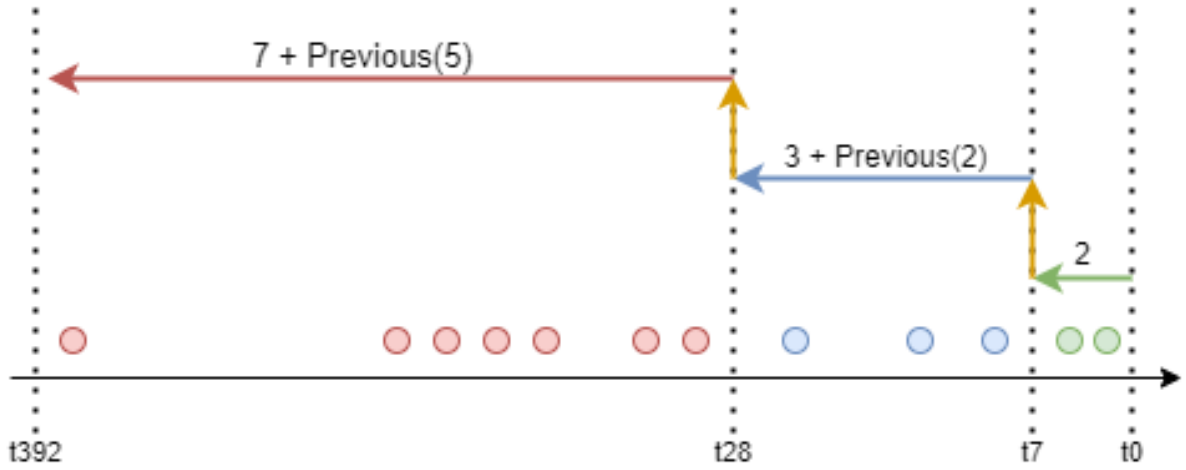

Figure 4.9: Cascading Window Aggregation Example

This is due to the fact that if the first record of the Flink RocksDB `MapState` has been found, the pointer for the following entries is already at the right position. Consequently, the complexity drops to $O(1)$.

The next approach shows a potential optimization strategy that can be used when multiple streams with different output behaviour are to be joined together. For example, this will be necessary when combining the day zero (*i.e.,* one input / one output) and daily sliding windows (one input / 1 output per day).

4.4.2 Handling Delayed Streams

An important point which is not obvious at first is the handling of time-delayed streams. Operators can have different output behaviors. An input / output consideration must be made. The continuously sliding window algorithm generate one output per input. However, this is not the case with daily (or longer) sliding or tumbling windows. Here, many events are bundled and normally output as a single record when the end of the time interval is reached.

However, if events from a non-time-delayed route (*i.e.,* day zero windows) have to be combined, this does not work without further ado. On the one hand, if the split join pattern is used, the records can only be sent further down the stream when all events have arrived in the join operator. However, this does not correspond to the desired result. It also causes the events in the baseline implementation to jam at the operator that was supposed to do the joining of the split data. This happens because the data on the ”fast” route was written almost immediately to the state of the joining operator where it had to wait for the results of the daily sliding windows. On the other hand, if a `union` operation is used, the different records must still be combined and pivoted into an output record. As a potential solution, a `CoProcessFunction` can be implemented. As described in Section 2.4.5, it contains two inputs and thus, two `processEvent` functions. `ProcessEvent1()` represents the ”fast” route and is called when an event is delivered on

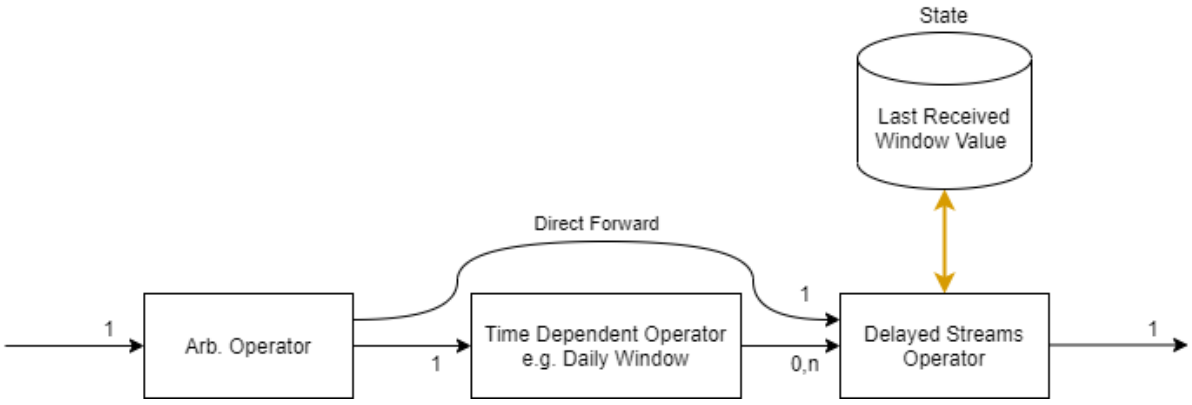

Figure 4.10: Delayed Streams Pattern

input 1. `ProcessEvent2()` is consequently called when a delayed event arrives on the second input. The architecture of the pattern is visualized in Fig. 4.10. The stream is split before the operation is performed that causes the time delay (*i.e.,* before the split into different output behaviours). From then a route flows via the delaying operator into the subsequent `CoProcessFunction` as input 2. The other route passes without detours into the `CoProcessFunction` as input 1. The function contains, depending upon the use case, a `Value-` or `MapState` which holds the last delivered value(s) of the time-delayed route. If such a record is output, it is written into the state structure of the `CoProcessFunction` as the current value. If a "fast" event is delivered on input 1, the state structure is checked whether a current value is present or not. If so, it is passed on as the current aggregation result. Otherwise, a `null` event is sent on.

With the help of this pattern, a pile-up of events can be prevented, since an output is always generated for each input. Consequently, it is not necessary to wait 392 days until a result is available at the end of the pipeline. As a second algorithmic optimization strategy, the continuously sliding window algorithm is examined in more detail.

4.4.3 Continuously Sliding Windows

The continuously sliding window algorithm should make it possible to keep the last **x** time units (*e.g.,* seconds, minutes, hours, days) in a window at any time. The default sliding or tumbling windows emit a result only at the end of a defined time interval. However, with the continuously sliding windows, there should only be one window, which slides with millisecond precision. This concept is illustrated in Fig. 4.11. The requirements can be specified as follows.

- It should be possible to define a dynamic window length, which contains the current result of the window length with millisecond precision.
- When a new event is added to the window, it should be automatically added to the state of the window.

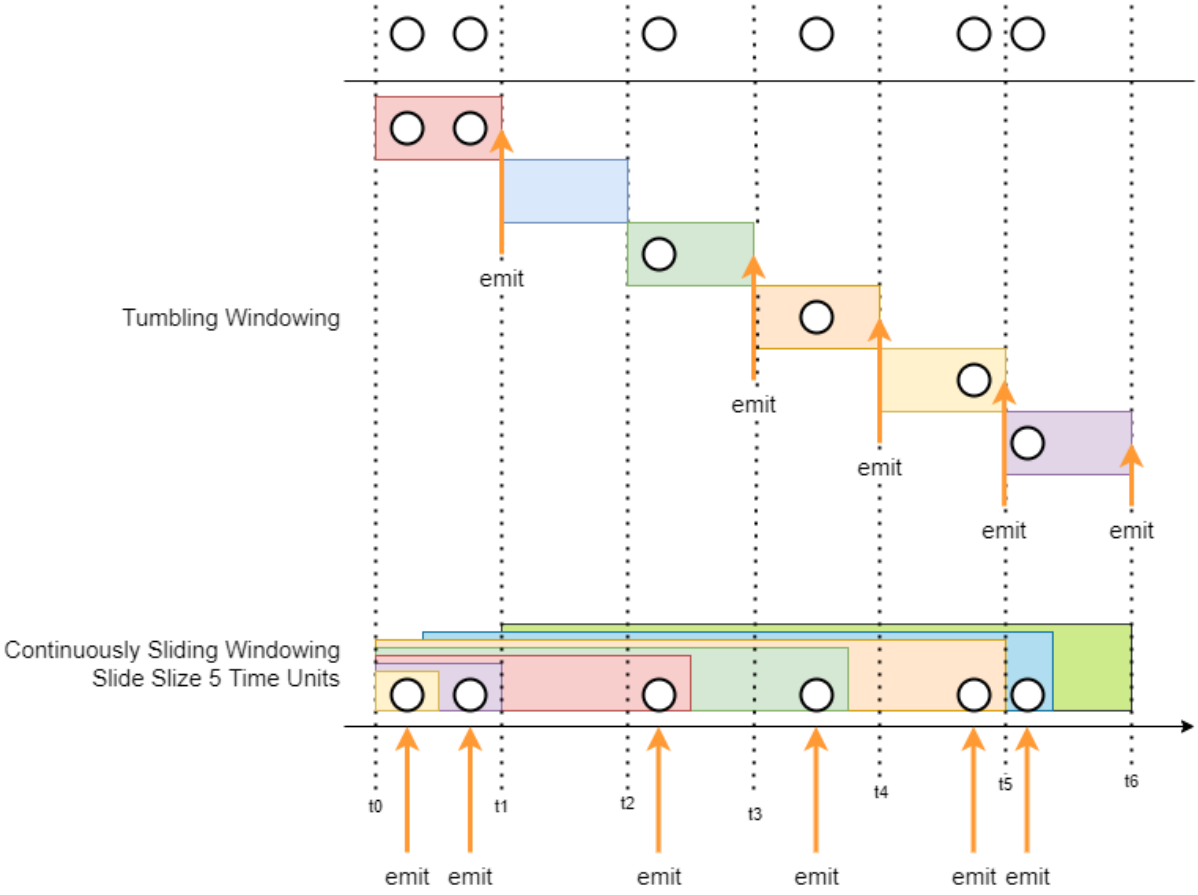

Figure 4.11: Comparison Tumbling And Continuously Sliding Windows

- If an event is no longer in the current time to be examined, it shall be automatically removed from the window.

The above requirements imply some challenges which have to be addressed. Adding events to such a window is trivial. However, extracting them is not as it implies that all events belonging to the window must be cached. Moreover, not all incremental algorithms support an inverse operation, which in turn makes it necessary to recompute the individual values.

One Stage Algorithm

A first idea which could be implemented with standard Flink means relies on a `ProcessFunction`. The approach is visualized on the left side of Fig. 4.12. When a new event arrives, it is ingested in the `ProcessFunction`, added to a `MapState`, aggregated with the pre-existing values and emitted as a result. Finally, a timer is set based on the duration for which the event is relevant plus the event time. When this timer is due, the event will be removed from the state. Even though the complexity of the aggregation per event is $O(n^2)$, this approach can be used for keys that contain little data per time interval.

Two Stage Algorithm

To reduce the impact of the incremental re-aggregation, it was decided to introduce a two-stage window algorithm (*cf.* right side of Fig. 4.12). In general, the process should work as follows. The events are first divided into smaller tumbling windows (*e.g.,* 30 minutes) (*cf.* box *Windowing*). These smaller windows in turn are collected in a second stage by a `ProcessFunction` (*cf.* box *Rollup*) and stored in a `MapState` structure. If one of these windows is now either added or renewed, only the pre-aggregated windows have to be re-aggregated, but not all events of the complete time interval.

If a new event is delivered, an event time timer is registered in the `Trigger`, which defines when the event must be removed from the window again. Afterwards, the event is stored in the `MapState` structure of the `ProcessFunction` (*cf.* box *Windowing*). These values are incrementally aggregated and passed on as intermediate window result.

Subsequently, the intermediate window results are stored in a `MapState` structure of a second `ProcessFunction` (*cf.* box *Rollup*). The new intermediate window result is incrementally rolled up with the existing window results in the `MapState` structure and returned as the result of the continuously sliding window at this exact point in time. As the intermediate window results get outdated at some point, for every window a removal timer is registered as soon as it appears for the first time. This way no state bloat can occur. If an event is delivered, for which a window was already emitted, the existing intermediate window result is overwritten.

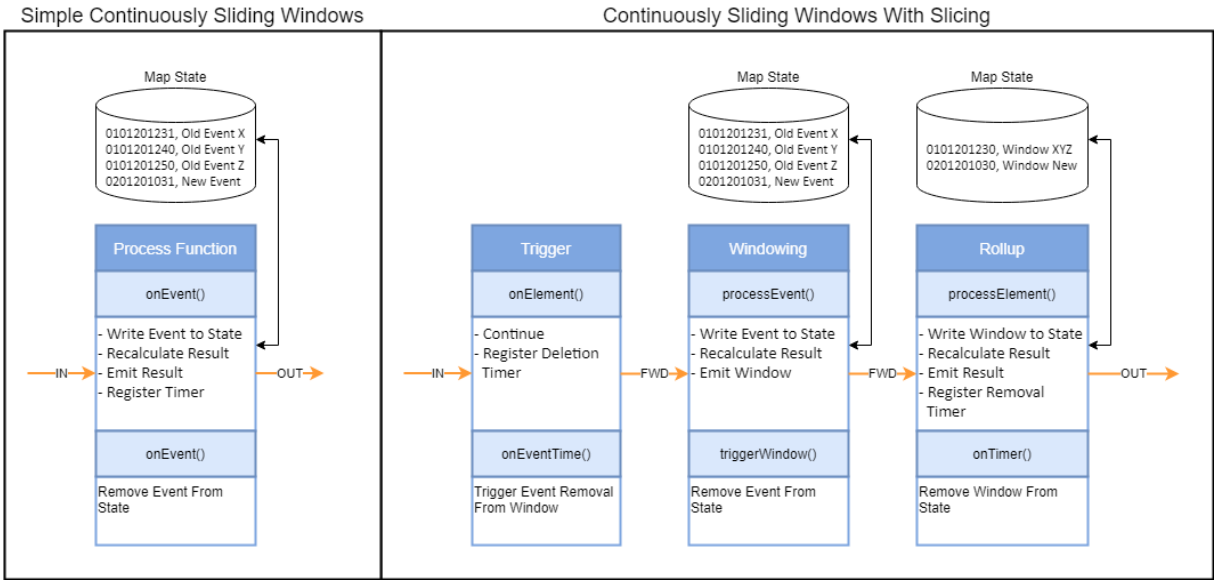

Figure 4.12: Simple And Slicing Continuously Sliding Windows

However, both variants are not optimal in terms of complexity of the aggregation. For each new event, either all events of the time interval, or all events of the window, respectively all intermediate window results must be aggregated. The reason why a re-aggregation of the window content is necessary is that no numerically stable subtraction operation is given for the required holistic aggregation algorithms. If values are repeatedly calculated to and

from a result, a small error can accumulate and finally, lie far away from the actual value. This error must be isolated and eliminated. A possible solution is the removing the old intermediate results and re-aggregating all remaining values. The following algorithm for continuously sliding windows has the potential to lower the impact of the re-aggregation each time a value has to be removed from a window.

Reverse Aggregation

The two-stage continuously sliding window algorithm mentioned in Section 4.4.3 assumes that a re-aggregation of the cached events of an intermediate window result must be performed with each added and removed record. This is due to the fact that no numerical stable inverse operation is available for the required aggregations.

Assumed that at a certain point in time no more events are added to a window, but only removed, the algorithm visualized in Fig. 4.13 can be implemented.

In a first phase, all events are handled the same way as described in Section 4.4.3. When point in time *tx* is reached, all values are written to another `MapState` structure in reverse order. Furthermore, the values are incrementally aggregated and stored together with the timestamp at which the corresponding event would have to be removed. Afterwards, as soon as an event is removed from the slice, the already aggregated value can be output. Therefore, a re-aggregation for each removed event is no longer necessary.

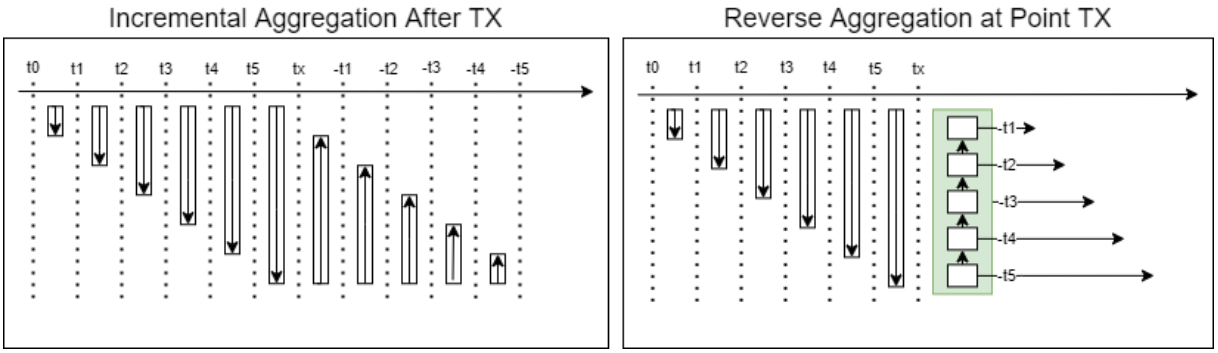

Figure 4.13: Comparison Between Incremental and Reversed Removal of Events in Slices

The design decisions made in this Chapter should allow the baseline implementation to be executed without overloading or congesting the system. The exact implementation follows in Chapter 5.

Chapter 5

Implementation

This Chapter discusses the implementation of the design decisions made in Chapter 4. The remainder of the Chapter is structured as follows. First, the implementation of the custom metrics is discussed. This is followed by explanations of the operational state optimizations. Namely, the reduction of the number of parallel windows and the switch between the RocksDB state backend and a manually managed Heap data structure for state storage. The third topic focuses on the I/O optimizations. The transformation of redundant shuffle operations and the extension of the task chaining are discussed. Finally, the algorithmic optimizations are described. These include the handling of streams with different output behaviour and the continuously sliding window algorithms.

5.1 Metrics

In Section 4.1, the additionally required metrics were divided into intra and inter operator measurements. The former is intended to provide a detailed insight into the individual operators, while the latter is used to measure the duration between two points in the job. The following Section will go into more detail on how these two requirements were implemented.

5.1.1 Intra-Operator Measurements

The structure of the intra operator measurements relies on the `getRuntimeContext` function of the `RichFunction` interface. It allows to interact with metrics using the function `context.getMetricGroup()`.

To enable a user to perform categorically subdivided measurements in a `ProcessFunction`, the in Listing 5.1 depicted trait was implemented. The `LatencyHistogram` class is used as an example of the actual implementation of the trait. The `open` function seen in Listing 5.2 is called when the `MetricsReporter` is initialized. This function handles the creation of the individual metrics. The operator name and the `MetricGroup` of the

corresponding `RuntimeContext` must be passed as parameters for the identification of the newly registered metrics. Furthermore, the list of categories and list of ratios of the categories can be provided. For the ratios, a list of tuples is required, where the first category of the tuple is divided by the second one. Subsequently, the function `startCategory` (*cf.* Listing 5.3) can be used to start one of the previously defined categories. The other two functions `startCustomDuration` and `endCustomDuration` can be used to make overall measurements independent of the categories. Finally, if the `report` function (*cf.* Listing 5.4) is called, the histograms are updated with the measured values.

```
1  trait MetricsReport {
2    def reset()
3
4    def open(
5        operatorName: String
6        , metricGroup: MetricGroup
7        , categories: List[String]
8        , ratios: List[(String,String)]
9    )
10
11   def startCustomDuration()
12
13   def endCustomDuration()
14
15   def startCategory(category: String)
16
17   def report()
18 }
```

Listing 5.1: MetricsReporter Trait Used For Intra-Operator Measurements

For each element in the list of categories to be created, a histogram is initialized which contains the operator name and the category itself as identification. Furthermore, the slide size of the histogram `WindowReservoir` can be defined. It determines the value capacity of the histogram. The same initialization process is carried out for the ratios. The only difference is that the two parts of the ratio tuple are taken as the name of the metric.

```
1  override def open(operatorName: String, metricGroup: MetricGroup,
       categoriesToAdd: List[String], ratios: List[(String,String)]) = {
2
3      categoriesToAdd.foreach {
4        x => {
5          categories = categories+(x ->
               metricGroupWithSubGroup.histogram(s"${operatorName}_${x}", new
               DropwizardHistogramWrapper(new Histogram(new
               SlidingWindowReservoir(100))).asInstanceOf[Histogram]))
6        }
7      }
8  }
```

Listing 5.2: Implementation of `open()` Function

If a new category is to be started, the system first checks whether another category is already active. If this is the case, the active category is terminated and the duration of the category measured. It is checked if a measurement for this category already exists. If so, the two values are combined. Otherwise, only the new measurement is stored. Afterwards, the currently active element is set to the new category. This process is depicted in Fig. 5.3.

```
override def startCategory(category: String): Unit = {

    endAndResetActiveElement()

    currentlyActive = Some(CurrentlyActive(category, System.nanoTime()))

}
```

Listing 5.3: Implementation of `startCategory(category: String)` Function

When the `report` function (*cf.* 5.4) is called, the currently active category is terminated. For each category, the collected values are retrieved and the histogram is updated. The same procedure is carried out for the ratios, except that the two values belonging to the ratio are divided by each other.

```
override def report(): Unit = {

    endAndResetActiveElement()

    categories.foreach {
        x => {
            val currentHistogram = x
            val currentValue = valuesByCategory.get(x._1)
            currentValue.map(value => currentHistogram._2.update(value))
        }
    }
    reset
}
```

Listing 5.4: Implementation of `report()` Function

With the help of these functions, it is possible to instantiate an implementation of the `MetricsReporter` in a `RichFunction` to perform categorical measurements. The next step is to look at the inter operator measurements.

5.1.2 Inter-Operator Measurements

Two options were introduced for the inter operator measurements. On the one hand, the duration from the ingestion of an event to the end of the pipeline can be measured. For this purpose, each event is enriched with a `MetaInformation` object at the beginning of the job. In this object, not only the event id for a possible split join scenario is stored but also the ingestion time (*i.e.,* `ingestionNanos`) and a timestamp which can be updated (*i.e.,* `lastProcessingTs`).

For measuring the **total duration**, the current nano time of the system is set by default when creating the `MetaInformation`. At the end of the job, a last monitoring `ProcessFunction` (*e.g.,* `ReportTotalDurationProcessFunction`) can be inserted. This operator contains the instantiation of a `MetricsReporter` which subtracts the ingestion time from the current nano time in the `ProcessFunction` and saves it in a histogram. This way, the latency of the entire pipeline can be measured.

If the **duration between one or more operators** is to be measured, the `lastProcessingTs` timestamp can be used. First, it must be updated by using a `map()` operator. Subsequently, the `ReportProcessingTimeProcessFunction` can be inserted at the desired position in the job. In its `open()` function, a histogram is initialized. The `processEvent` function is then used to subtract the `lastProcessingTs` value from the current nano time. The whole process is shown as code in Listing 5.5. An overview of the different inter-operator measurements is depicted in Fig. 5.1.

```
1
2  val withMetaInfo = inputStream.appendMetaInformation()
3
4  val updatedTs = withMetaInfo.map(event =>
5          event.update(event.metaInformation.copy(lastProcessingTs =
               System.nanoTime())))
6      )
7
8  val applyOps = updatedTs.process(new SomeOperationsProcessFunction())
9
10 val measureTime = applyOps.process(new
       ReportProcessingTimeProcessFunction())
11 }
```

Listing 5.5: Updating `lastProcessingTs`, Applying Operations And Measuring Duration

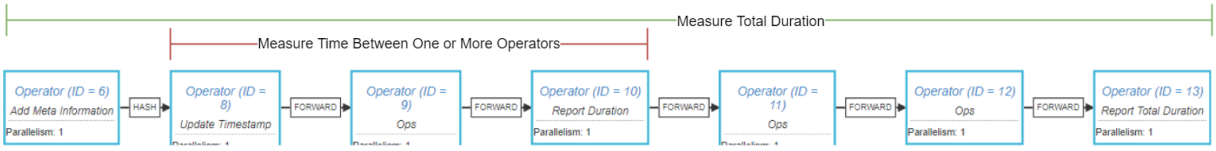

Figure 5.1: Comparison of Different InterOperator Measurement Types

With the help of the intra and inter operator measurements, more detailed insights into a running job can be provided. The next step is to take a closer look at the operational state optimization strategies and their implementation.

5.2 Operational State Optimizations

The state optimizations are mainly concerned with reducing the number active parallel windows. This is important as the use of a large sliding window size with small slide parameter is associated with potential issues. For example, for each slide interval, all

records of the given time period are stored. This means a multiplication of the data that has to be stored, which has a negative effect on performance. The parallel execution of the window aggregations also generates de/serialization effort, as the individual windows need to be combined afterwards. The second optimization regarding operational state is the situational switch between the RocksDB state backend and a manually managed Heap data structure if the data and key cardinalities allow it. It is not possible to store all data in memory. However, for keys that do not expect an extensive amount of data, this approach can be considered.

5.2.1 Reduction of Number of Parallel Windows

The initial problem that made the reduction of the number of parallel windows necessary was the duplication of data and state accesses as mentioned in Section 4.2.1. When looking at the number of features that need to be calculated, multiple parallel running windows generate a considerable amount of overhead.

Therefore, a two-stage windowing algorithm was introduced that helps with the reduction of the number of parallel windows. As a first step, the lowest common denominator of the window size has to be found. Due to the business requirements in the present case, daily tumbling windows are chosen.

These daily tumbling windows are gathered in a second step by the subsequent `ProcessFunction` and stored in a `MapState` structure. The key of the `MapState` is the end time of the daily tumbling window added to the maximum window length in milliseconds. This is due to the automatic deletion of the daily tumbling windows from the `MapState` structure by a timer as soon as the record is not relevant anymore.

```
1  //Daily Tumbling Windows With an Allowed Lateness of 28 Days
2  val tumblingByOneDay = aggregatedTumblingWindowFeatures(inputStream,
       Time.days(1), Time.days(28))
3
4  //Aggregation of Daily Tumbling Windows
5  val d007 = tumblingByOneDay
6      .reKey(x => x.key)
7      .aggregateWindowsBy("AggregateTumblingDailyTo007",
           Time.days(7).toMilliseconds)
8
9  val d028 = tumblingByOneDay
10     .reKey(x => x.key)
11     .aggregateWindowsBy("AggregateTumblingDailyTo028",
           Time.days(28).toMilliseconds)
12
13 val d392 = tumblingByOneDay
14     .reKey(x => x.key)
15     .aggregateWindowsBy("AggregateTumblingDailyTo392",
           Time.days(392).toMilliseconds)
```

Listing 5.6: Aggregation of Daily Tumbling Windows

If the window value already exists in the `MapState` structure, it is overwritten with the new one (*i.e.,* late event handling). The output is calculated by aggregating all elements in the `MapState` structure. Finally, the timer for the deletion of the daily tumbling window from the state is registered.

Listing 5.6 presents the usage of the two-step algorithm. First, the incoming events are divided into the daily tumbling windows. They are accumulated with the existing window values of the previous days by using the function `aggregateWindowsBy`. The currently presented algorithm emits a result once a day for each window (*i.e.,* one input / zero-to-one output). However, if a window is added that has a different output behaviour (*i.e.,* one input / one output), the merging of the different output behaviours becomes more complex. This problem is handled in more detail in Section 5.4.2. The reduction of the number of parallel windows already takes some pressure off the system. Further improvements to this issue are presented in Section 5.4.1. The next topic that is discussed in terms of the state is the switch between the RocksDB state backend and a manually managed Heap data structure. It allows the user to operate on the Heap while still being able to benefit from the Flink Checkpointing API.

5.2.2 Manually Managed State on Heap Data Structures

The switch between the RocksDB state backend and a manually managed Heap data structure allows a user to decide which state store to use based on the expected data volume of a key. For this, the `CheckpointendFunction` interface is used. It grants control over the process that handles the initialization and snapshotting of state in a stateful function. The `CheckpointedFunction` interface forces the user to implement the `initializeState` and `snapshotState` functions. `InitializeState` is used to initialize the state on startup or when in recovery from a Checkpoint. `SnapshotState` is called when a Checkpoint is triggered and a snapshot of the state needs to be generated.

The overall goal of the approach is to enable a user to store values into an in-memory structure which is still able to benefit from the Checkpointing API provided by Flink. Thus, the application is still recoverable in case of a failure although values are not directly stored in a state backend.

As aggregations per key were needed until now, the `KeyedStateBackend` would be used to store incoming events or window values. For the present approach, the operator state has to be used which was introduced in Section 2.4.6. It differs from keyed state in terms that all records handled by the same operator access the same state. Whereas for keyed state, each key is separated and has its own state. When using operator state, different state primitives are available. For the present approach, the `ListState` and `UnionListState` are considered. While the `ListState` safes a list of the entries per operator instance, the `UnionListState` stores a list of all values across all parallel operator instances. The advantage of the latter is, that in case of a change in parallelism, the values can be distributed to the correct operator by checking the key group assignment and subtask index. The `ListState` will be evenly split and then sent to the different parallel instances of the operator. Due to the aforementioned advantages in case of a restore, the `UnionListState` was chosen as the state primitive.

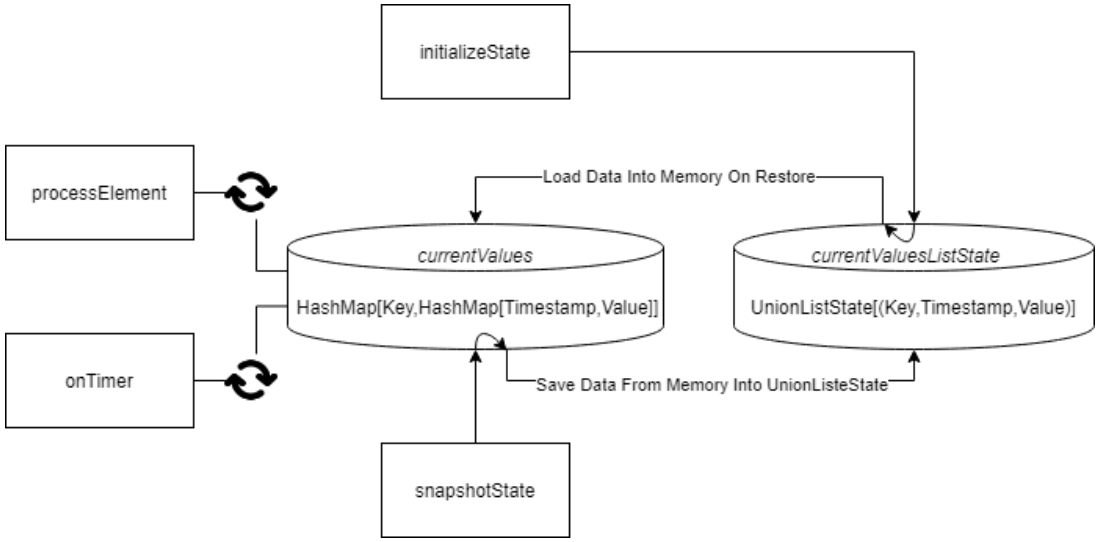

Figure 5.2: Algorithm Used For The Manually Managed Heap Data Structure

Fig. 5.2 visualizes the algorithm presented in Listing 5.7. The functions `processElement` and `onTimer` are not discussed in detail as they are not central for the algorithm. `ProcessElement` is called when a new event is received. It stores the event in the in-memory `HashMap` and aggregates it with the already existing values. Finally, the new window value is output and a timer is set to remove the value from the in-memory state again in the `onTimer` function. It is important to note that both functions work with the in-memory `HashMap`. No direct operations are performed on the `UnionListState`. Another important point is the nesting of the `HashMap`. As operator state is used, values are not grouped by key when accessing the state. This task is left to the user. Therefore, the first level of the `HashMap` is the grouping by individual keys, the second level is the grouping by timestamp. This way, when removing a value from the in-memory structure, a search for the key can be performed. Afterwards, the timestamp to be deleted can be removed from the second `HashMap`.

The core of the algorithm is found in the `snapshotState` and `initializeState` function. As stated before, the `UnionListState` is used. Therefore, when a Checkpoint is triggered, the in-memory `HashMap` structure needs to be converted into a `List`. As to safe all relevant information, a `Tuple3` consisting of the key, timestamp and value itself was used as the `List` entry. Afterwards, the newly formed Tuple gets added to the `UnionStateList`. The `initializeState` function is called when initializing the job. This can either be due to a fresh start or a restore. In case of a fresh start, only the `UnionListState` needs to be initialized so it can be used for the Checkpointing. In case of a restore, the algorithm is more complex. As the `UnionListState` contains the data of all instances of an operator, a splitting has to be carried out. Not every operator instance needs to have all values. To achieve this, the `KeyGroupRangeAssignment` class was used, which handles the assignment a given key to a parallel operator index in Flink. For every value in the list, the `assignKeyToParallelOperator` function is called. Afterwards, it is checked if the key belongs to the index of the operator instance. If true, the value is written into the in-memory `HashMap`. If not, no action is taken.

```
1  class D0ProcessFunction extends KeyedProcessFunction[KEY, KV, KVW] with
       CheckpointedFunction{
2
3      @transient lazy val currentValues = new mutable.HashMap[KEY,
           mutable.HashMap[Long,KV]].empty
4
5      @transient lazy var currentValuesListState: ListState[(KEY, Long, KV)]
           = null
6
7      def processElement()
8
9      def onTimer()
10
11     override def snapshotState(functionSnapshotContext:
           FunctionSnapshotContext): Unit = {
12         currentValuesState.clear()
13         for {
14             entry <- currentValues
15             k = entry._1
16             v <- entry._2
17         }yield(currentValuesState.add((k,v._1,v._2)))
18     }
19
20     override def initializeState(functionInitializationContext:
           FunctionInitializationContext): Unit = {
21         val descriptor = new ListStateDescriptor[(KEY, Long, KV)](...)
22         val operatorStateStore =
               functionInitializationContext.getOperatorStateStore
23
24         currentValuesState =
               operatorStateStore.getUnionListState(descriptor)
25
26         if (functionInitializationContext.isRestored) {
27             val iterator = currentValuesState.get().asScala
28
29             iterator.foreach(entry=>{
30                 val operatorIndexForKey =
                       KeyGroupRangeAssignment.assignKeyToParallelOperator(...)
31                 val indexOfSubTask =
                       getRuntimeContext().getIndexOfThisSubtask()
32                 if(operatorIndexForKey.equals(indexOfSubTask)){
33                     addToHashMap(entry._1,entry._2,entry._3)
34                 }
35             })
36
37         }
38     }
39 }
```

Listing 5.7: KeyedProcessFunction With CheckpointedFuntion Interface

With this algorithm it is possible to store data in a manually managed Heap data structure while still being able to benefit from the Checkpointing mechanism provided by Flink. For future work, a dynamic switch between the in-memory storage and the RocksDB state

backend based on the cardinalities of a key could be implemented. This concludes the optimizations that were applied to the state. In the next Section, the I/O optimizations follow.

5.3 I/O Optimizations

The I/O optimizations deal with the reduction of latency by reducing I/O interactions. For this purpose, two subtopics were highlighted in Chapter 4. Their implementations are discussed in the following Section. First, the transformation of redundant shuffle operations is described. Afterwards, the extension of the task chaining algorithm is explained in more detail.

5.3.1 Transforming Redundant Shuffle Operations

First of all, the individual key domains must be precisely known for this optimization strategy. The developer must ensure that the data in the section to be optimized is always partitioned the same way. Then, a `DataStream` can be reinterpreted into a `KeyedStream` by using the function `reinterpretAsKeyedStream`. To elevate the usability, the function has been put into a syntax object (*cf.* Listing 5.8). This syntax allows the extension of classes with functions

```
1  implicit class DataStreamSyntax[I](i: DataStream[I])
2  (implicit IInfo: TypeInformation[I]) {
3
4      def reKey[K](ik: I => K)
5      (implicit KInfo: TypeInformation[K]): KeyedStream[I, K] =
6      new KeyedStream[I,K](
7          DataStreamUtils.reinterpretAsKeyedStream(i.javaStream
8          , new KeySelector[I, K] {
9              override def getKey(in: I): K = ik(in)
10         }, KInfo)
11     )
12 }
```

Listing 5.8: `DataStream` Syntax to Conveniently Use `reinterpretAsKeyedStream`

The `reKey` function is given a projection which extracts a key from the record type of the stream. The `reinterpretAsKeyedStream` function takes the source stream, a `KeySelector` which determines the key using the given projection function as well as a `TypeInformation` of the key type as a parameter. The final result after applying the function is a `KeyedStream`, transformed from a `DataStream`.

As a last step, the shuffle operations before and after the split/join pattern have to be bundled. This can be achieved by inserting a `map` operator. An example pipeline is depicted in Fig. 5.9. First, the key and the value to be aggregated is extracted from the stream. Then a `keyBy` is performed. Afterwards, the map operator is inserted which

should bundle the shuffle operations. Next, a `reKey` is used which converts the `DataStream` after the `map` operation into a `KeyedStream` again. This is followed by splitting the stream and performing the individual operations. Finally, the splitted streams are combined again with a `union` operator, bundled with a `map` operator and then keyed by the evend id. The last keyBy event id is needed for the subsequent joining of the individual streams. The principle of the split join pattern was explained in Section 3.2.

```
1  val bundledShuffle = inputStream
2      .extractKey(x => x.cardId)
3      .extractValue(x => MomentumAggregate(x.amount))
4      .keyBy(x => x.key)
5      .map(x => x).name("Bundle Shuffle")
6      .reKey(x => x.key)
7
8  val D0 = bundledShuffle
9      .process(new ApplyOperationsProcessFunction())
10
11 val D7 = bundledShuffle
12     .process(new ApplyOperationsProcessFunction())
13
14 val D28 = bundledShuffle
15     .process(new ApplyOperationsProcessFunction())
16
17 val D392 = bundledShuffle
18     .process(new ApplyOperationsProcessFunction())
19
20 val bundledOut = D0
21     .union(D7,D28,D392)
22     .map(x => x)
23     .name(s"Bundled Forward")
24     .keyBy(f => f.metaInformation.eventId)
```

Listing 5.9: Bundling of Shuffle Operations

Fig. 5.3 depicts the revised pipeline of Fig. 4.4 after removing the redundant shuffle operations. Thus, the total number of shuffle operations for this key domain could be reduced from 14 to 2.

The next optimization strategy focuses on the extension of the task chaining. It should help to lower the number of tasks, and thus, lowering the latency caused by the de/serialization and buffering between the tasks.

5.3.2 Task Chaining

The extension of the task chaining was mainly considered an issue due to the large number of nodes that were created in the baseline implementation (*cf.* Section 2.5). For each created task of the pipeline, a de/serialization and buffering of the data is carried out. This leads to a potential delay which is not neglectable in an execution plan with around 1800 nodes. Section 2.4.7 describes how an execution plan is built by Flink and how task chaining is embedded in it. Especially of interest is the second transformation when

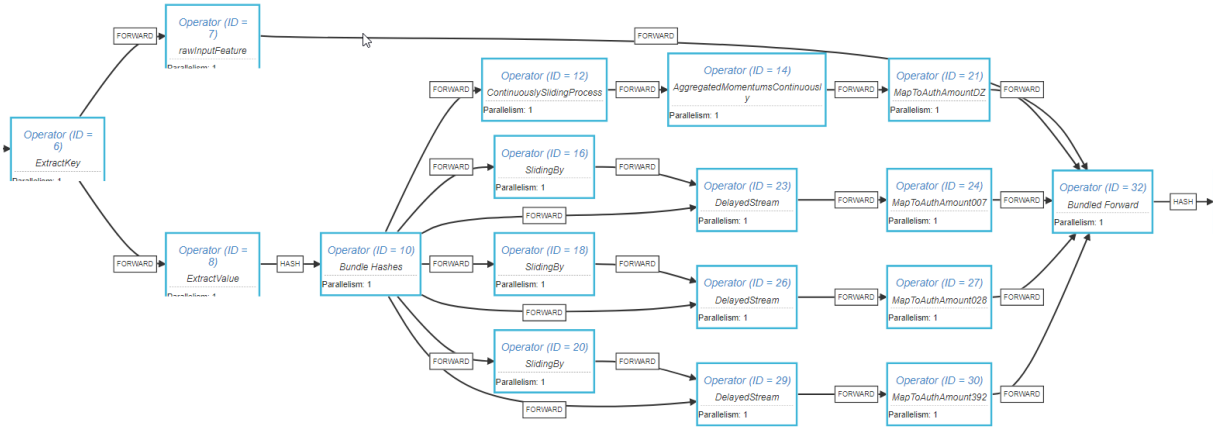

Figure 5.3: Pipeline Shown In Fig. 4.4 After Removing Redundant Shuffle Operations

converting a `StreamGraph` into a `JobGraph`. In this stage, the task chaining takes place. Flink currently relies on the algorithm shown in Section 2.4.7 to determine whether an operator can be chained or not. For the use case listed in this paper, the issue was found in the `downStreamVertex.getInEdges().size() == 1` statement of the `isChainable()` function. It made it impossible to chain operators with more than 1 input.

To mitigate this issue, a way had to be found to implement the possibility to add a custom chaining strategy to the `JobGraph` generator. Due to the closed architecture of Flink, a large number of changes had to be applied to the classes involved in the transformation of a Flink program into an execution plan. For simplicity, not all the changes will be described in this Section.

As a first step, the `StreamExecutionEnvironment` needs to be extended. As a result, the newly created `DiStreamExecutionEnvironment` is able to accept a representation of the `ChainingStrategy` interface presented in Listing 5.10

```
1 public interface IChainingStrategy {
2     boolean isChainable(StreamEdge edge, StreamGraph streamGraph);
3 }
```

Listing 5.10: Chaining Strategy Interface

An implementation of the `ChainingStrategy` interface is then handed down the transformation chain from the `DiLocalStreamEnvironment` to the `StreamGraph`. As it is also built in a closed way, a custom implementation has to be provided, called the `DiStreamGraph`. Besides other changes, it passes the `ChainingStrategy` to the `StreamingJobGraphGenerator`. Again, a closed architecture is present and thus it has to be re-implemented as the `DiStreamingJobGraphGenerator`. In there, the passed `ChainingStrategy` is used by a strategy pattern to switch between the different chaining implementations, which can be configured from the outside.

The whole process was enormously more complex due to the closed architecture of the `StreamingJobGraphGenerator` as well as the `StreamGraph` and the associated classes

that determine the chaining of tasks. Due to the reduction of the number of parallel operators described in Section 5.4.1 this optimization will not have a big impact on the overall performance of the present pipeline. Nevertheless, an interesting concept is shown which can be useful under certain circumstances.

The last Section described ways to lower the latency generated by the I/O. It presented the transformation of redundant shuffle operations as well as an extension of the task chaining. The next section discusses the implementation of the algorithmic optimizations.

5.4 Algorithmic Optimizations

The first part of the algorithmic optimizations is dedicated to the refinement of the reduction of parallel window operations shown in Section 5.2.1. The second part focuses on the handling of streams with different output behaviour, which led to an almost uncontrollable growth of state in the baseline implementation. Finally, the implementation of the continuously sliding window algorithm is discussed and the functionality is explained in detail.

5.4.1 Reducing Redundant Operators

The concept of the reduction of redundant operators as mentioned in Section 4.4.1 is based on processing all key domains (*i.e.,* when the same operations must be performed) in the same pipeline. Furthermore, a cascading of the window aggregations shall be carried out. Thus, a fanning out due to different window lengths is no longer necessary.

To handle keys of several key domains (*i.e.,* different key types) in the same pipeline, they must be converted into a common type. A first solution to achieve this goal was to convert the keys by means of the Flink serializer into a byte array and use this array as a key. Due to a restriction in Flink, this approach had to be discarded. In the documentation [8] the following statement is made.

A type cannot be a key if:

- It is a POJO type but does not override the hashCode() method and relies on the Object.hashCode() implementation.
- It is an array of any type.

Next, obtaining a common type by using a hashing algorithm for the different keys was discussed. The key is converted into a byte array. Subsequently, this array is converted by an interchangeable hash algorithm (*e.g.,* Sha-256, Sha-512) into a string representation. Thus, all keys have the same data type and can be used in the same pipeline. The disadvantage of this approach is that a hash is a ”one-way” function. A possible later reversal of the hash is not possible.

Therefore, this design was also discarded and the solution described in Listing 5.11 was implemented. First, the keys are serialized into a byte array. The byte array is then converted into a char array. Since a string is a representation of a char array, the char array can be converted to a string, which is a common data type that is not an array. A conversion of the string to the original representation is still possible, which makes the function revertible.

```
1  def projectToHash[T, K](project: T => K)
2  (implicit aInfo: TypeInformation[K], jobEnv: JobEnv)
3  : T => String = {
4      val ts = aInfo.createSerializer(jobEnv.getConfig)
5      def convert(input: T): String = {
6          val byteArrayStream = new ByteArrayOutputStream()
7          val out: DataOutputView = new
               DataOutputViewStreamWrapper(byteArrayStream)
8          //Function Used to Project a Record Into a Key
9          val key = project(input)
10         ts.serialize(key, out)
11         byteArrayStream
12             .toByteArray()
13             .map(_.toChar)
14             .mkString
15     }
16     convert
17 }
```

Listing 5.11: Key Generalization

The second optimization which should reduce the number of parallel operators is the cascading of the different window aggregations. A two-stage window algorithm is proposed for this. In a first step, windows of the smallest common unit are created. In the present case, sliding windows of different sizes (*e.g.,* 7, 28, 392 days) are required by the business. The windows should slide by one day. Thus, tumbling windows are created on a daily basis. In the remainder of this section, the term D1 will be used for the daily tumbling windows. In the second stage, the core piece of the algorithm follows. In Fig. 5.4, the operations that take place in the `KeyedProcessFunction` utilized for the calculation of the cascading windows are visualized. The list below provides an overview of the necessary steps for the algorithm.

1. Incoming D1 windows are stored in the `MapState`
2. An iterator is used to loop through the `MapState`
3. Remove first value if it's timestamp is out of scope
4. Dynamically divide the D1s into buckets according to the defined window sizes
5. While dividing the values into buckets, use the aggregate function to accumulate the values
6. Add calculated buckets to a list, always append new buckets to the head of the list to flip the ordering

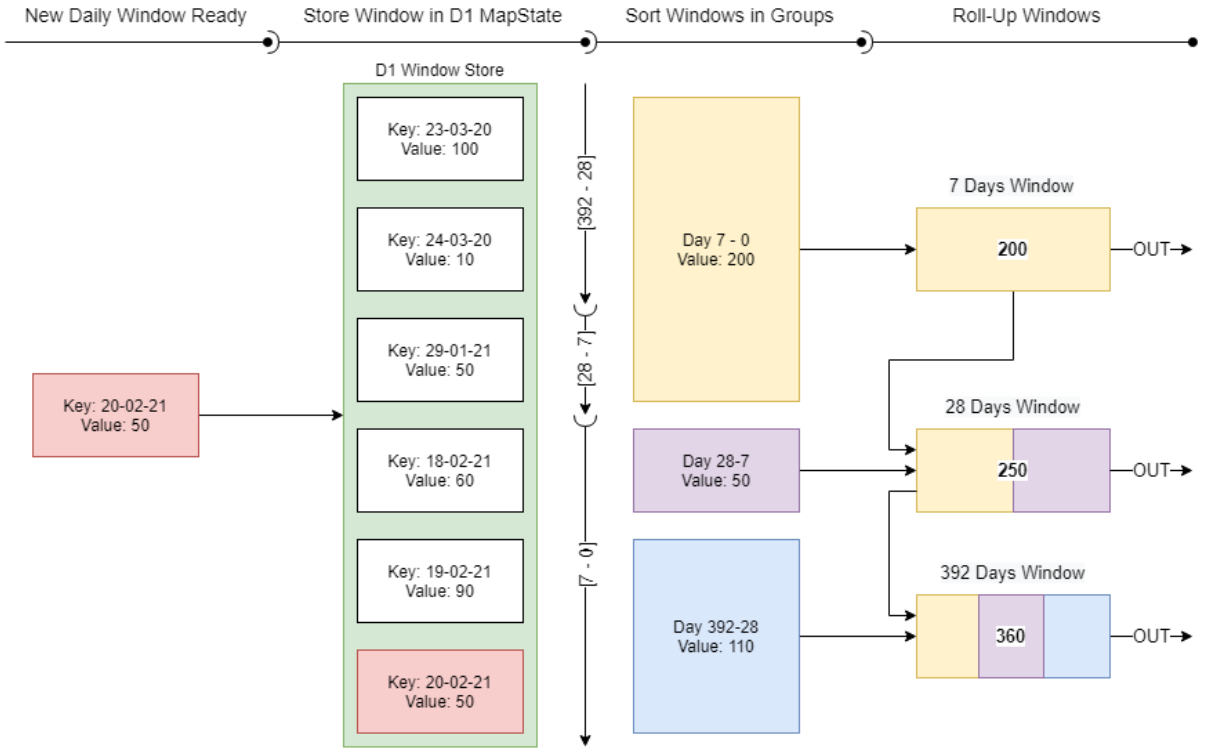

Figure 5.4: Dnn Algorithm Used For Cascading Window Calculation

7. Register a timer for the next day to check if a value needs to be removed from the `MapState`

8. Iterate through the list, incrementally add all buckets to the corresponding size and output window results

It is important to note that this approach works due to an inherent characteristic of the RocksDB state backend. The values stored in RocksDB are canonically ordered due to the usage of a column family storage system. The algorithm exploits this characteristic when using an iterator on the `MapState`. Due the column store, no more search operations have to be performed when the first entry for the query has been found. The iterator is able to iterate over the returned data structure in constant time and aggregate all the windows in one go. This property is not given for the other state backends. The advantage of this procedure is that all windows for a key can be calculated serially. No parallel state access for the different windows is necessary. Although an algorithm is used that is worse in theory, the given characteristics of the RocksDB state backend mitigate some disadvantages of the approach. In addition, no de/serialization effort is generated when combining the individual window sections, which also has a positive effect on the performance.

With this algorithm it is possible to calculate the different daily sliding windows in a performant way. Additionally, multiple key domains can be handled by the same pipeline. No operators need to be duplicated solely because of a different data type. Still, there is an open issue with the combination of the daily sliding windows and the continuously sliding

windows, which have a different output behaviour. Thus, the next Section is dedicated to the handling of the so-called delayed streams.

5.4.2 Handling Delayed Streams

Combining streams with different output behavior is a problem that is not obvious at first glance. The streams can be merged (*i.e.,* if the streams have the same output data type) with a `union` operator. However, if a pivoting of the events has to be done, there is almost nothing else to do but to wait for the end of the window for each event and then output a common record. This process is described in the split/join pattern shown in Section 3.2. Only when all streams have delivered a result, the process can continue. Therefore, an accumulation of events in the joining operator and thus, a bloating of the state occurred in the baseline implementation. A possible remedy is the pattern visualized in Fig. 4.10.

A stream is split with one outgoing path performing the actual windowing algorithm before being combined with the other path in the subsequent `KeyedCoProcessFunction`. This merging is presented in Listing 5.12. It should be noted which input is passed into the `processEvent1` and `processEvent2` function. The input which is listed as a parameter the `connect` operator is always linked to the `processEvent2` function.

```
1 val delayedStream = inputStream
2     .connect(windowedStream)
3     .process(new CombineDelayedStreamsFunction())
```

Listing 5.12: Combination of Delayed Streams With a KeyedCoProcessFunction

The `KeyedCoProcessFunction` contains either a `MapState` or `ValueState` structure depending on the scenario. If a window is now supplied on the second input, the `processEvent2` function updates the state of the `lastWindow` value as presented in Listing 5.13.

```
1 override def processElement2(
2     value: IN2
3     , ctx: CoProcessFunction[IN1,IN2, OUT]#Context
4     , out: Collector[OUT]
5 ): Unit = {
6
7     lastWindow.update(value)
8
9 }
```

Listing 5.13: ProcessEvent2 Function of DelayedStreams Operator

Since the incoming event is directly forwarded by the route that is skipping the windowing, a one-to-one relation between the input and output is achieved. The directly forwarded events are handled by the `processEvent1` function. It checks whether a window is already stored in the state. If no window is present, an `Option[None]` is emitted. Else, the last received window is output. In the `out.collect` statement, the `MetaInformation` of the

buffered window result is overridden. This is due to the fact that the stored window result contains an outdated event id, which would cause problems when a merge had to be carried out later. Therefore, it needs to be adapted to the current `MetaInformation`. This process is presented in Listing 5.14.

```
override def processElement1(
    value: IN1
    , ctx: CoProcessFunction[IN1,IN2, OUT]#Context
    , out: Collector[OUT]
): Unit = {
    val last = lastWindow.value()

    if (last == null) {
        out.collect(None)
    } else {
        out.collect(Some(last.copy(metaInformation =
            value.metaInformation)))
    }
}
```

Listing 5.14: ProcessEvent1 Function of DelayedStreams Operator

The last Section described how for each incoming event an outgoing one is emitted. A piling up of the events does not occur any more, which increases the performance noticeably. The last optimization is dedicated to the continuously sliding window algorithm and how slicing can leverage the performance of the window calculation.

5.4.3 Continuously Sliding Windows

A custom implementation of the continuous sliding window algorithm was necessary as Flink does not offer an efficient operator for this requirement. The problem with continuously sliding windows is that with large amounts of data, no efficient slicing can be introduced that acts with millisecond precision. Normal tumbling windows are only calculated when the end time of the window is reached. However, if a result is to be output for each event, it must be handled via triggers. A trigger can be defined that calls the `processEvent` function of the `ProcessWindowFunction` for each incoming event. After the desired slide period has expired, the event must be removed again. Accordingly, another trigger must be defined for each event in order to remove it. Alternatively, an evictor can be used. However, both alternatives have the disadvantage that they keep all events of the window in a `List` object, which does not provide any ordering. Consequently, if an element is to be removed, all elements of the `List` must be checked for their timestamp whether it is expired or not. This leads to a complexity of $O(n^2)$. Therefore, this is not a valid approach. Especially with large amounts of data, no satisfactory performance can be guaranteed. In Section 4.4.3, two approaches were introduced that could help to mitigate this problem. The remainder of this Section is used to describe the implemented solutions.

One-Stage Algorithm With Process Function

The first algorithm that was considered mainly works for keys that cover a smaller amount of data. In this variant, no slicing takes place. Thus, no previous trigger must be registered and only a `ProcessFunction` is required. If a new value is added, all events of the slide period are aggregated repeatedly. As shown in Listing 5.15, the incoming events are stored by the `processEvent` function in the `MapState` (*i.e.,* currentValuesState) structure. Subsequently, the map is iterated through for each value and the records are aggregated incrementally. At the end the window result is output and a timer is registered which is needed to remove the previously stored event from the `MapState`. The whole process is lightweight but should only be used if a small amount of data is expected for certain keys.

```
override def processElement(
    value: KV
    , ctx: KeyedProcessFunction[KEY, KV, KVW]#Context
    , out: Collector[KVW]
): Unit = {

    val deletionTime = ctx.timestamp() + keepEventFor

    currentValuesState.put(deletionTime, value)

    val it = currentValuesState.iterator
    var aggregatedValue: KV = af.createAccumulator()
    while (it.hasNext) {
        aggregatedValue = af.add(it.next(), aggregatedValue)
    }
    out.collect(aggregatedValue.addWindow(None))
    ctx.timerService().registerEventTimeTimer(deletionTime)
}
```

Listing 5.15: `ProcessEvent` Function of The One-Stage Algorithm

Two-Stage Algorithm With AllStateWindow

The implementation of the continuously sliding windows with a two-stage algorithm is achieved by using an extension of the Flink `WindowOperator` developed in the cooperating company. The basic idea behind this is that a `ProcessWindowFunction` can distinguish based on a trigger whether a new event is added or an action on an existing window (*e.g.,* remove operation) is to be performed. A representation of the adjustments is visualized in Fig. 5.5. The upper diagram shows the original `WindowOperator` and how it interacts with the `WindowFunction` based on the `TriggerResult`. The lower diagram shows the adapted `AllStateWindowOperator`, which is derived from the existing `WindowOperator`. In addition, the `ProcessAllStateWindowFunction` is derived from the `ProcessWindow-Function` and extended by another function called `triggerWindow`. This differs from the default implementation as follows. The standard `WindowOperator` relies on the two functions `processElement` and `onEventTime`. The former is called as soon as a window is ready and executes the `process` function in the `ProcessWindowFunction` attached to

the operator. The function `processElement` is only called when a `TriggerResult == FIRE` is present. For every other `TriggerResult`, the algorithm stops. The `onEventTime` function distinguishes between the `TriggerResult PURGE` and `FIRE`. The former clears the values in the window, the latter calls the `ProcessWindowFunction` along with the entire window content.

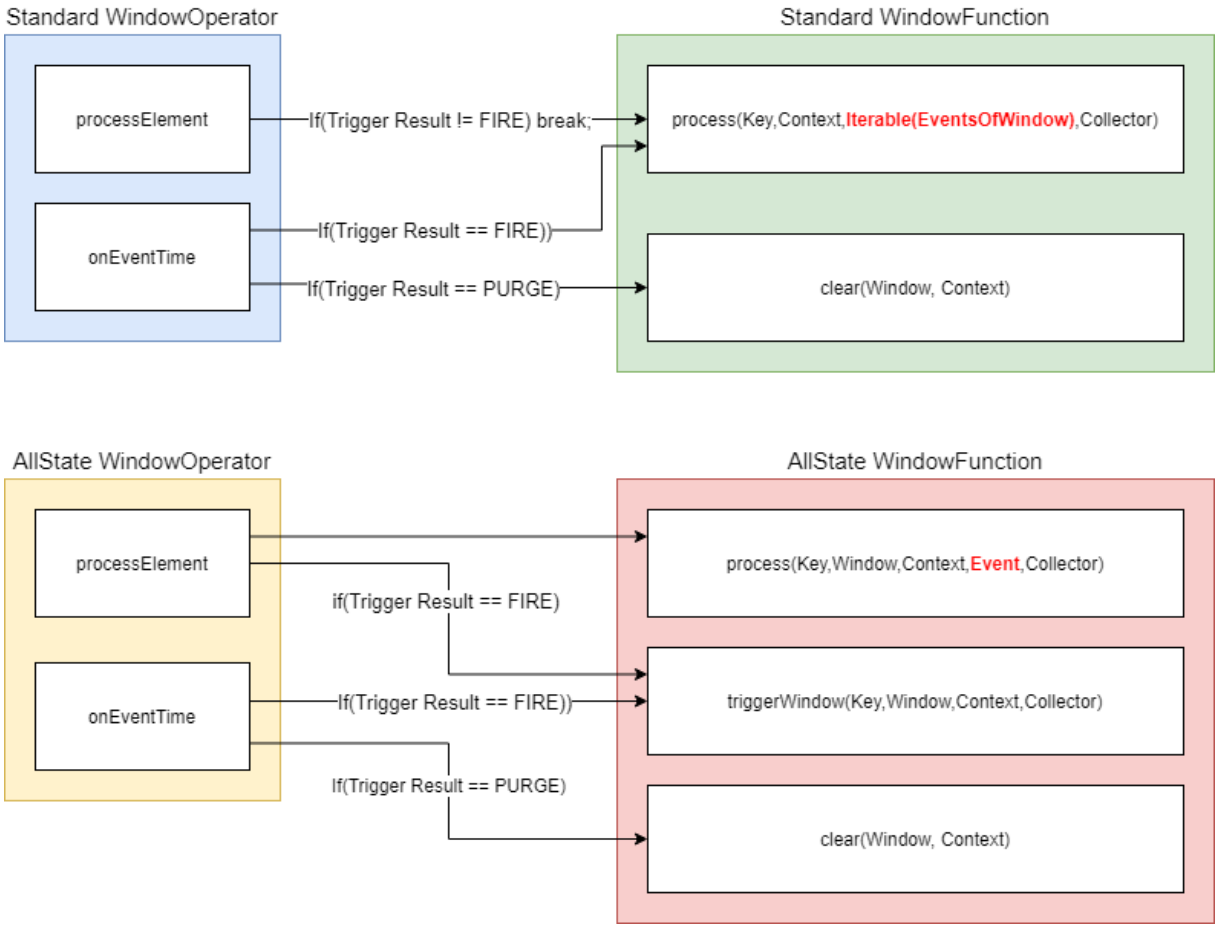

Figure 5.5: Difference Between `WindowOperator` And `AllStateWindowOperator`

In the customized `AllStateWindowOperator`, the `processElement` function calls the `ProcessWindowFunction` for each event when a `TriggerResult==CONTINUE` is present. As a first difference compared to implementation in Flink, no `TriggerResult==FIRE` is needed to call the `process` function. Nevertheless, if a `FIRE` is present, the `triggerWindow` function is called additionally. Another difference is found in the function interface. The standard implementation uses a `List` of the whole window content as an argument. The new implementation only hands-over the current event to be processed. If a `onEventTime` is triggered, the `process` function of the `ProcessWindowFunction` is not called as usual, but the `triggerWindow` function instead. This function is supplied the current timestamp of the triggering event in addition to the existing attributes of the `Context`. This makes it possible to find and remove events stored in the `MapState` structure with constant complexity. The need to search a `MapState` or `ListState` to find the value that needs to be removed can be omitted. In a second stage, these pre-aggregated windows are combined into the actual continuously sliding window with its full slide size. For the present case,

this means that the pre-aggregated windows of 30 minutes length are combined into 24 hour windows. This is achieved by implementing a `ProcessFunction` that contains a `MapState` structure with the deletion timestamp of the window as a key and the window result as value. When a new window is delivered or an existing one is updated, the `MapState` structure is adjusted accordingly and a re-aggregation of the windows is carried out. Finally, the accumulated value is emitted as the current result of the continuously sliding window.

Reverse Aggregation

The in Section 5.4.3 presented version of the continuously sliding window algorithm relied on re-aggregating all values in a certain window when a record had to be removed. Furthermore, it is not distinguished between an add or remove operation. An alternative approach was described in Section 4.4.3. At a certain point in time, values will only be deducted from a window with a high probability. This point in time is normally reached when the first event gets removed from the window. Speaking in a continuously sliding context, this point is reached when the first call to `triggerWindow` is made. As for the cascading window calculation shown in Section 5.4.1, the approach is based on the canonical ordering that is given by the RocksDB state backend.

The implemented process is depicted in Fig. 5.6. The algorithm is divided into two different phases. Phase one is dedicated to the addition of values to the `MapState` containing the **Raw Input** and accumulating a **Running Aggregation** which is implemented as a `ValueState`. The key for the **Raw Input** `MapState` is the time to remove the entry and the value represents the unaggregated value of the event. When the aggregation of the newly delivered event is done, the result of the `ValueState` is output.

As soon as the first call to the `triggerWindow` function is made, phase 2 starts. Initially, the **Raw Input** is taken and the values are put into a `List` in reverse order. Subsequently, an iterator is used to incrementally aggregate the values for each key. By the time the trigger for the `triggerWindow` function fires, the current value is already outdated. Therefore, the incrementally aggregated values are shifted by one iteration. When all values are calculated and stored in the second `MapState` structure (*i.e.,* **Reversed State Storage**), no recalculation of the window has to be performed anymore as the results are already present for the respective key.

In the last paragraph it was stated that at a certain point, no new values will be added to the window **with a high probability**. This indicates that late events are still possible. Thus, the algorithm also needs to be able to handle this case. Fig. 5.7 visualizes the measures taken to counter this issue. When a late event arrives, it is picked up but not written into the **Raw Input**. Instead, an iterator is used to go through the **Reversed State Store**. For each element, the current value is retained in a variable. Then the element is checked if the key that has to be added is smaller than the key of the current element. If it is smaller, the new value is added to the current value and the next element is checked. If it is not smaller, the new key is inserted into the `MapState` together with the new value plus the retained value of the first key that was bigger. As soon as the value is found, the algorithm stops as no further steps need to be taken.

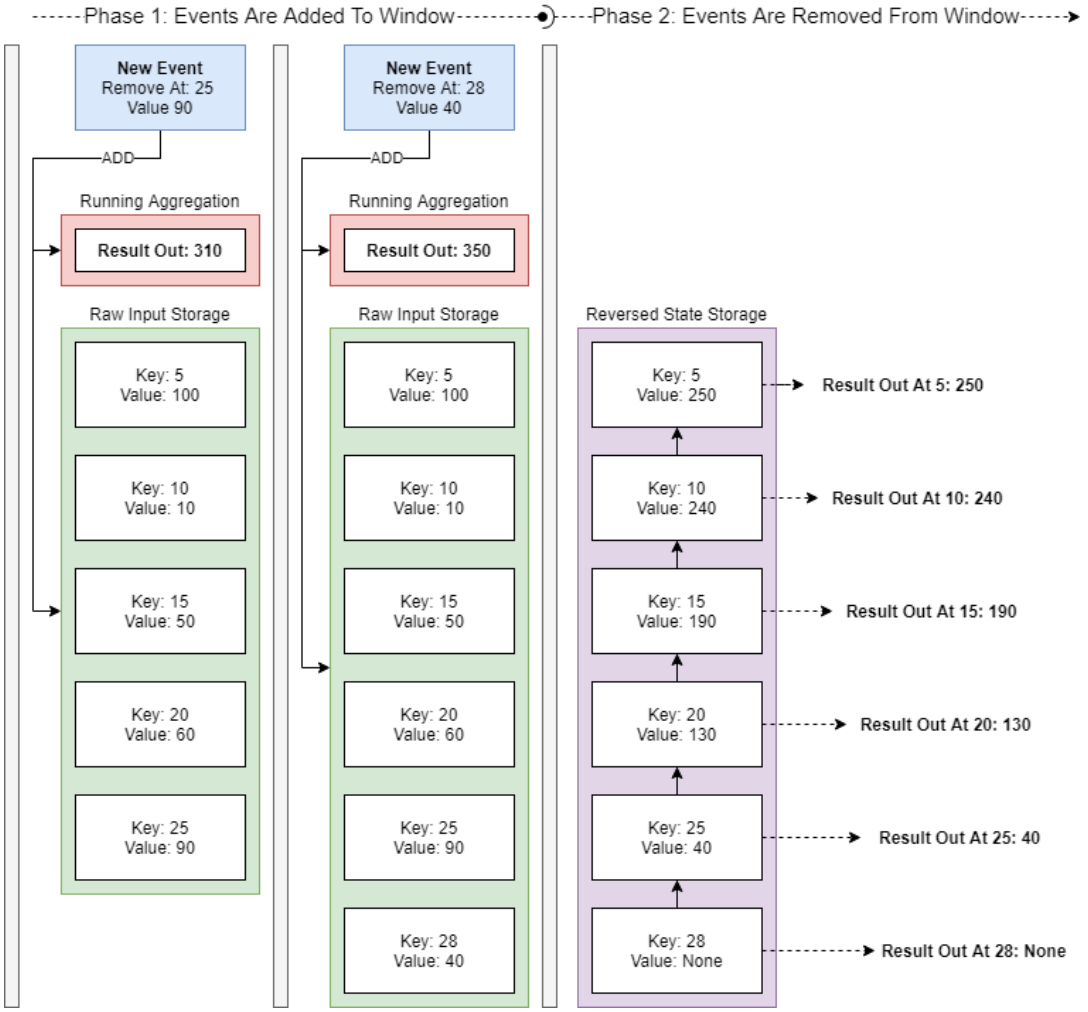

Figure 5.6: Phases of The Continuously Sliding Window Reverse Aggregation Algorithm

This section concludes Chapter 5, which described the implementation of the design decision made in Chapter 4. First, the reduction of the number of parallel windows and the switch between the RocksDB state backend and a manually managed Heap data structure for state storage were shown. Afterwards, the I/O optimizations, namely the transformation of redundant shuffle operations and the extension of the task chaining were described. Finally, the reduction of redundant operators, the handling of streams with different output behaviour as well as the different continuous sliding window algorithms were explained. Next, Chapter 6 will evaluate the implemented solutions regarding their effect on the performance of the present streaming job.

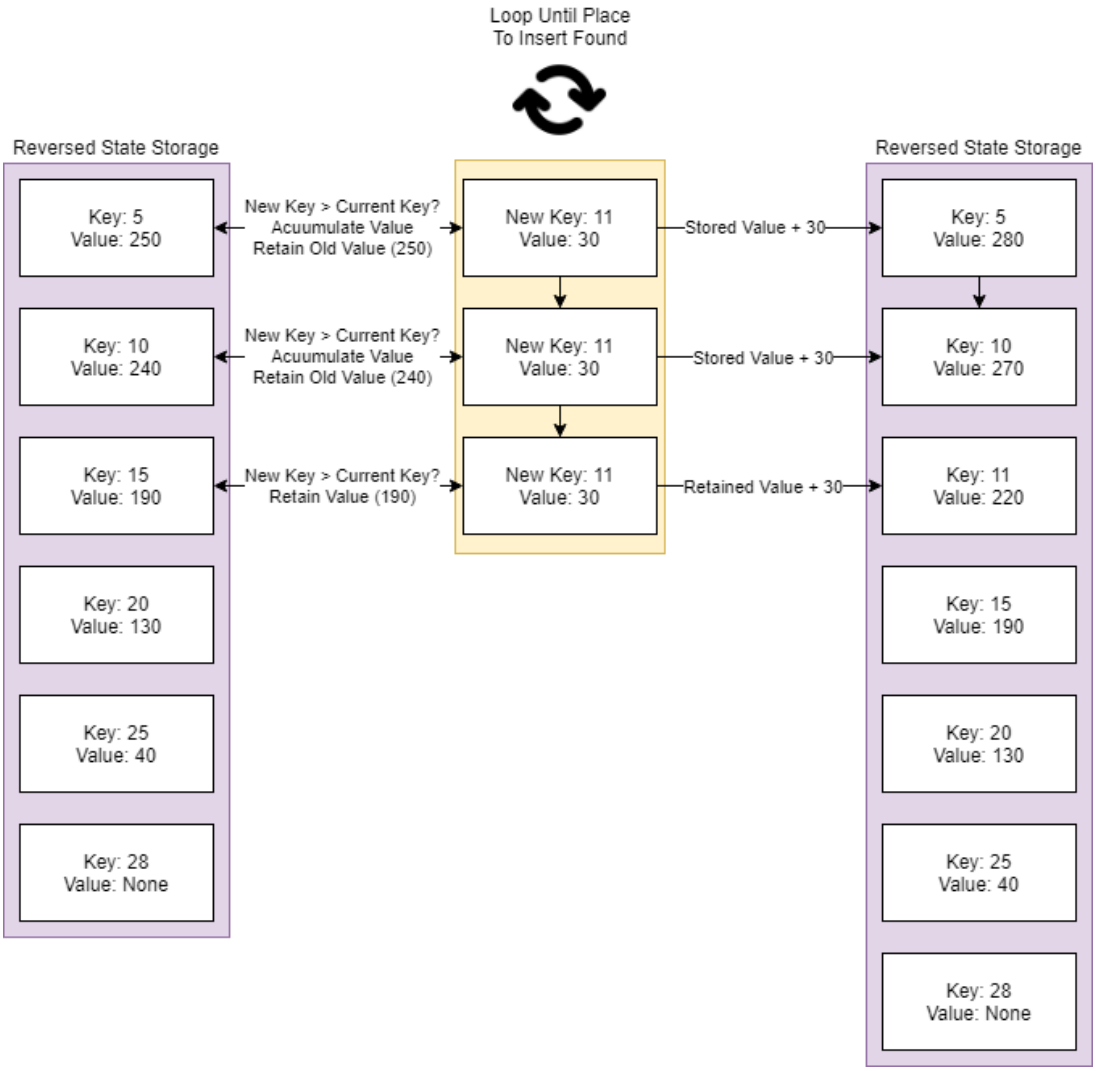

Figure 5.7: Late Event Handling of The Reverse Aggregation Algorithm

Chapter 6

Evaluation

While testing the implemented baseline described in Section 2.5, some logical issues in scenario 3 were found. Furthermore, as the results of the baseline implementation with an even higher number of parallel operators would not be more expressive, it was decided to focus on scenario 1 and 2 for the evaluation. Scenario 3 could not be fully implemented due to a time shortage, thus it was delayed for future work. Nevertheless, a special characteristic and interesting test case is the aggregation of extensive data for a small key space in scenario 3. To simulate this, the same aggregations as in scenario 1 were considered, but the key was exchanged with one key of scenario 3. Table 6 summarizes the characteristics for the different scenarios.

The different key characteristics are taken into account, as some of the effects only get visible in certain use cases. For instance, I/O optimizations will be most effective, when there is a considerable number of tasks and operators that generate I/O delays. Optimizations regarding aggregations in windows will be more relevant if there is either a bigger key space, or higher data volume.

	Scenario 1	**Scenario 2**	**Scenario 3**
Key Domains	1	8	24
Key Space	Small (2 Million)	Large (1.6 Billions)	Medium (60 Million)
Data Characteristics	Sparse Data	Sparse Data	Extensive Data
Number of Events / Day	400k Events	400k Events * 8 Keys	400k Events * 4 Keys * 3 Classifications
Aggregations	Momentum Aggregate		
Window Types	DZ, 7, 28, 392 Daily Sliding Windows		

Table 6.1: Scenario Definitions

6.1 Setup

The evaluation of the different use cases was performed on a Virtual Desktop Infrastructure (VDI) client with 16 GB of memory and an Intel(R) Xeon(R) Gold 6142 CPU with

2.60 GHz. The test data is generated by the data generator implemented in the previous master's project [6]. The data is stored in the test environment of the Kafka server of the company [7].

The largest time horizon to be looked at is 392 days. As per year 200 million authorizations are performed in reality, around 215 million synthetic records are being generated for the test cases.

During the implementation of the master thesis, a switch from Flink version 1.8 to Flink 1.12 was necessary due to a bug that caused the incremental Checkpointing to fail. The task chaining was still implemented with version 1.8 and could not yet be adapted to 1.12. Also, Flink 1.12 offers some improvements in that exact topic. Due to a time shortage, further investigations in that direction had to be postponed. Nevertheless, it proved that optimizations in that field are currently also discussed by the developers of Flink.

One further restriction that arose from the existing environment is related to the authentication to the Kafka server with Kerberos [55]. When testing from the VDI, a Kerberos ticket has to be acquired. Due to security restrictions, the validity period for this ticket is set to a few hours without a possibility to extend. If the time-frame has passed, a new ticket has to be requested. Due to the local execution, the job is not able to renew the ticket while running and thus fails to retrieve data from Kafka after a certain time. Therefore, the maximum run duration of a test is restricted. As a result, the job that is used to evaluate scenario 1 and 2 in combination will not be able to finish all of the 392 days needed to get into a steady state. As a compromise, only a maximum of 200 days will be tested. The resulting performance will be analyzed to make statements about the expected performance if the whole range of days would be used.

Furthermore, no production like cluster is available to test the job. Thus, another way had to be found to simulate the parallel execution of the job and the splitting of data. To mitigate this issue, the sharding of the data as presented in Listing 6.1 was introduced. The key of the incoming event is hashed. If the result of the modulo operation between this hash and the maximum shard count equals the shard index that should be retrieved, the element is forwarded. Else it is filtered out. This way the data is split as if it would be spread out to different instances. For the upcoming evaluation, a shard count of 16 is chosen.

```
1  if (shards > 0) {
2      authStream = input
3          .filter(
4              new FilterFunction[EVENT] {
5                  val shardCount = shards
6                  val shardInd = shard
7                  override def filter(value: EVENT): Boolean = {
8                      key.hashCode % shardCount == shardInd
9                  }
10             }
11         ).name(s"shard${shard}of${shards}")
12 }
```

Listing 6.1: Sharding Used To Simulate Parallel Execution

The last Section described the general setup for the evaluation of the different optimization strategies. The next Sections will provide an individual look at the optimization strategies and compare them to the baseline implementation. Some strategies have a bigger effect on the overall performance than others. Thus, certain optimizations (*e.g.,* Section 6.2.3 and 6.3.2) that have a more subtle impact are not compared to the baseline implementation, but to an already improved version of the job.

6.2 Operational State Optimizations

For the state optimization strategies, a look is first taken at the impact of RocksDB parameter tuning. Afterwards, the reduction of the number of parallel windows and the switch between the RocksDB state backend and the manually managed Heap data structure is evaluated.

6.2.1 RocksDB Parameter Tuning by [4]

If the available memory of a Flink job is too small, the evaluation of the algorithms will be affected by environmental factors. This leads to distorted results. Parameter tuning has a potentially large impact on the efficiency of a job. To increase the significance of the strategies, the parameter tuning will be applied to all the following tests. For this reason, this type of optimization is evaluated first in order to identify possible, indispensable effects on the performance. In Fig. 6.1 and 6.2 the difference between a job with and without additional managed memory (*cf.* Fig. 3.4) is depicted. It can be seen that this parameter is crucial for jobs that need to store a considerable amount of state. When no additional memory is assigned (*i.e.,* default of 256MB is used), the calculation of one single day takes approximately around 10 minutes. Furthermore, it can be seen in the bottom of Fig. 6.1 that during the calculation of the windows, all other parts of the job are blocked. The biggest increase in throughput could be seen when assigning the job around of 1GB of memory. Of course, this value depends on the requirements of the job itself and the data it needs to process. Nevertheless, once enough memory is assigned to Flink, a further increase of the memory does not provide a better performance. When using RocksDB, the increase of the background thread number also showed to be essential. For the other parameters listed in Section 3.4, no clear indication could be found that they have a significant impact on the present job. Nevertheless, further investigations in this directions should be carried out to find the optimal parameters, as soon as the other optimization strategies are implemented and do not yield any more performance gains.

6.2.2 Reduction of Number of Parallel Windows

This optimization strategy aims at reducing the number parallel windows by pre-aggregating the sliding windows into daily tumbling windows. The approach was compared to the baseline, aggregated with keys from scenario 1 and 2. The difference between the two

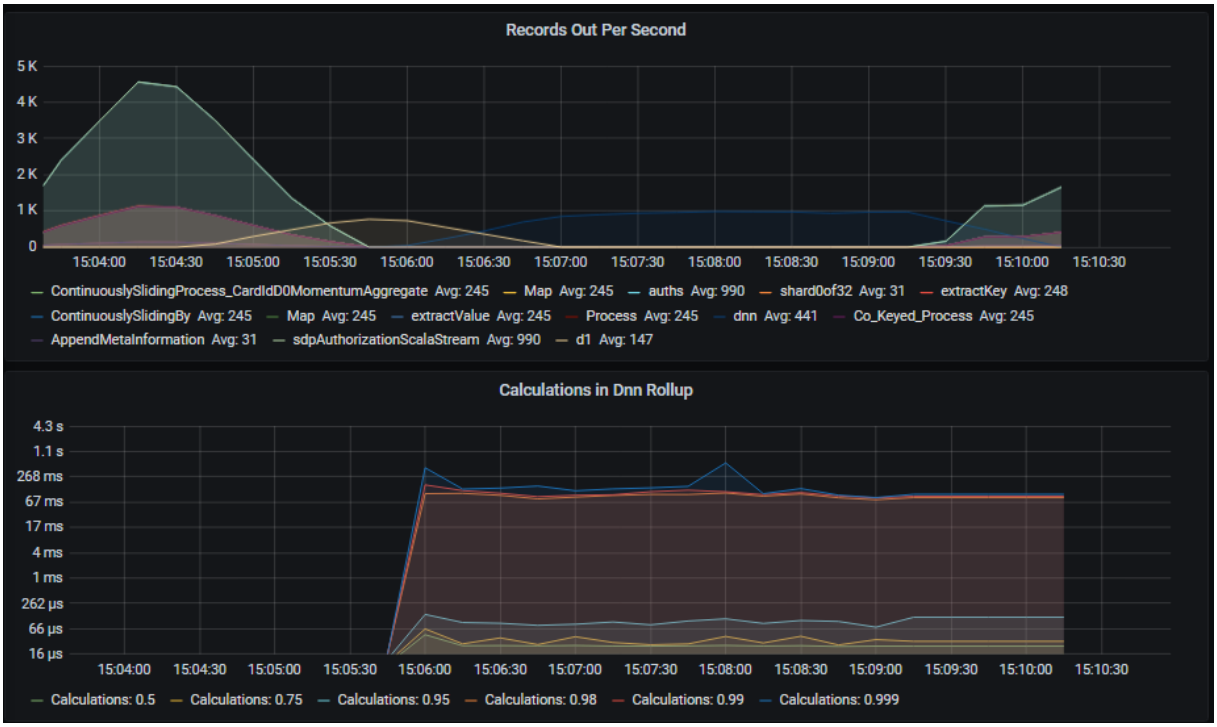

Figure 6.1: Flink Job Without Adaption of the Default Managed Memory

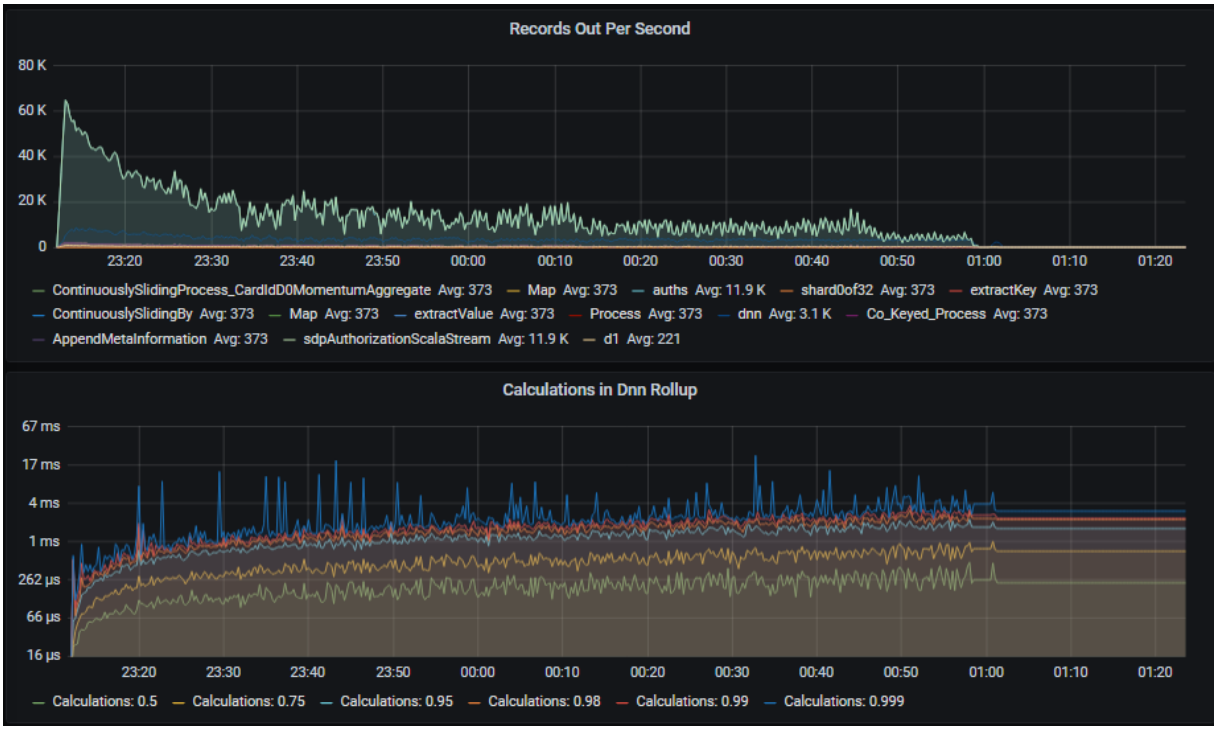

Figure 6.2: Flink Job With 4GB of Managed Memory

algorithms is shown in Fig 6.3 and 6.4 in terms of number of records out as well as the overall output pattern the pipeline generates.

What can be seen first is a significant improvement when it comes to the number of records out per second. Almost immediately, the performance of the baseline implementation severely degrades as seen in Fig. 6.3. The number of records out per second gets smaller and the space between the single spikes is increasing. This is mainly due to the aggregation of the sliding windows. First, for every day a sliding window of the defined length needs to be kept in state and aggregated. As a consequence, the events will take increasingly longer to process after some time. The job also blocks itself after a short period of time. While evaluating, most of the time this happened shortly after one hour passed. When looking at the pre-aggregation approach in Fig. 6.4, spikes can be seen. When comparing the time of the spikes with the watermark progress it shows that the spikes always happen when the daily tumbling windows are aggregated. This effect is described by [56] as the **the trembling herd effect**. It states that when a lot of concurrent windows need to be calculated at the same point in time (*e.g.,* daily sliding windows). The sudden appearance of a *herd* of calculations blocks the system for some time. To mitigate this problem, [56] suggest to stagger the window creation and therefore the time when it is evaluated. For the system at hand, this is unfortunately no option as the windows really need to cover one whole business day from 00:00 to 23:59. It is not possible to start a window at 10:30 and another one at 15:09. This leads to the conclusion that optimizations have to be applied to the algorithm for the aggregation of the daily windows. The optimization implemented for this issue is the cascading window aggregation described in Section 5.4.1.

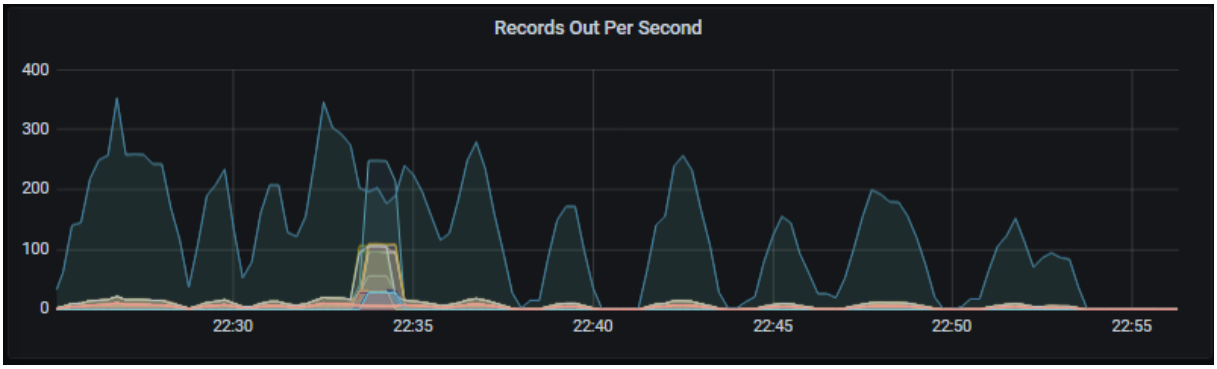

Figure 6.3: Number of Records Out Behaviour of The Baseline

When comparing the number of active keys in the state backend, the difference between the baseline implementation (*cf.* Fig. 6.5) and the presented approach (*cf.* Fig. 6.6) is obvious. By using the aggregation of daily tumbling windows, a significant reduction of the active keys could be achieved. For the observed interval, the number of active keys at a time went down from around 20 million to less than 1 million.

Although reducing the number of parallel windows yielded good results (*i.e.,* increasing throughput from a few 100 to 10'000 events per second), three individual `ProcessFunctions` are still used for window aggregation. Hence, three different `MapStates` per key area and aggregation function. This fact diminishes the impact of pre-aggregation into tumbling windows. A potential solution for this is the cascading window calculation (*cf.* Section 5.4.1) where only a single `MapState` primitive per key domain and aggregation function is required. Still, Fig. 6.6 shows a promising property. After the time of a window

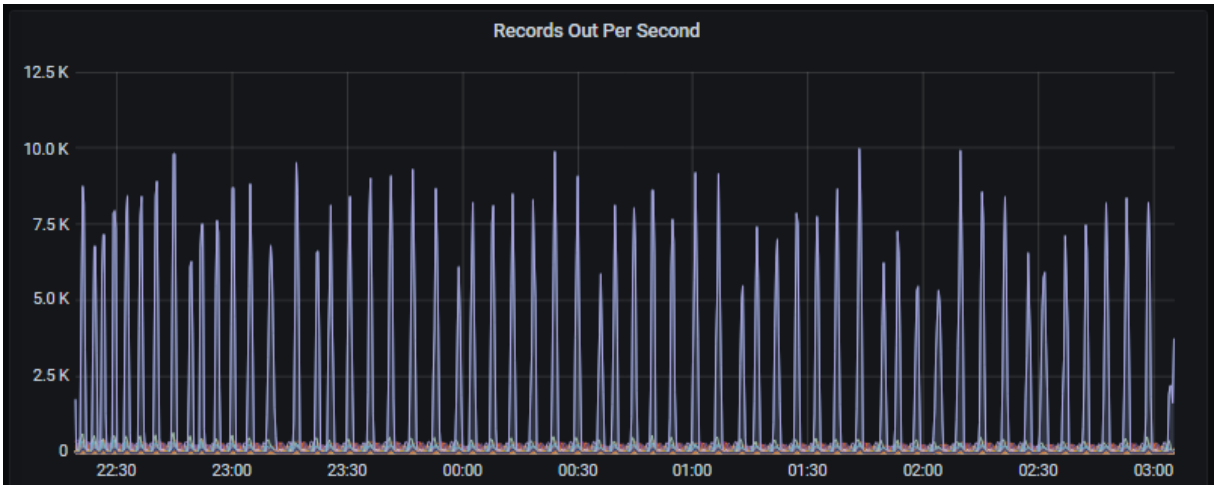

Figure 6.4: Number of Records Out Behaviour For Parallel Window Reduction

horizon has passed, the amount of keys for this state primitive stays nearly constant. This shows that the values get added to and from the aggregations and no state bloat occurs.

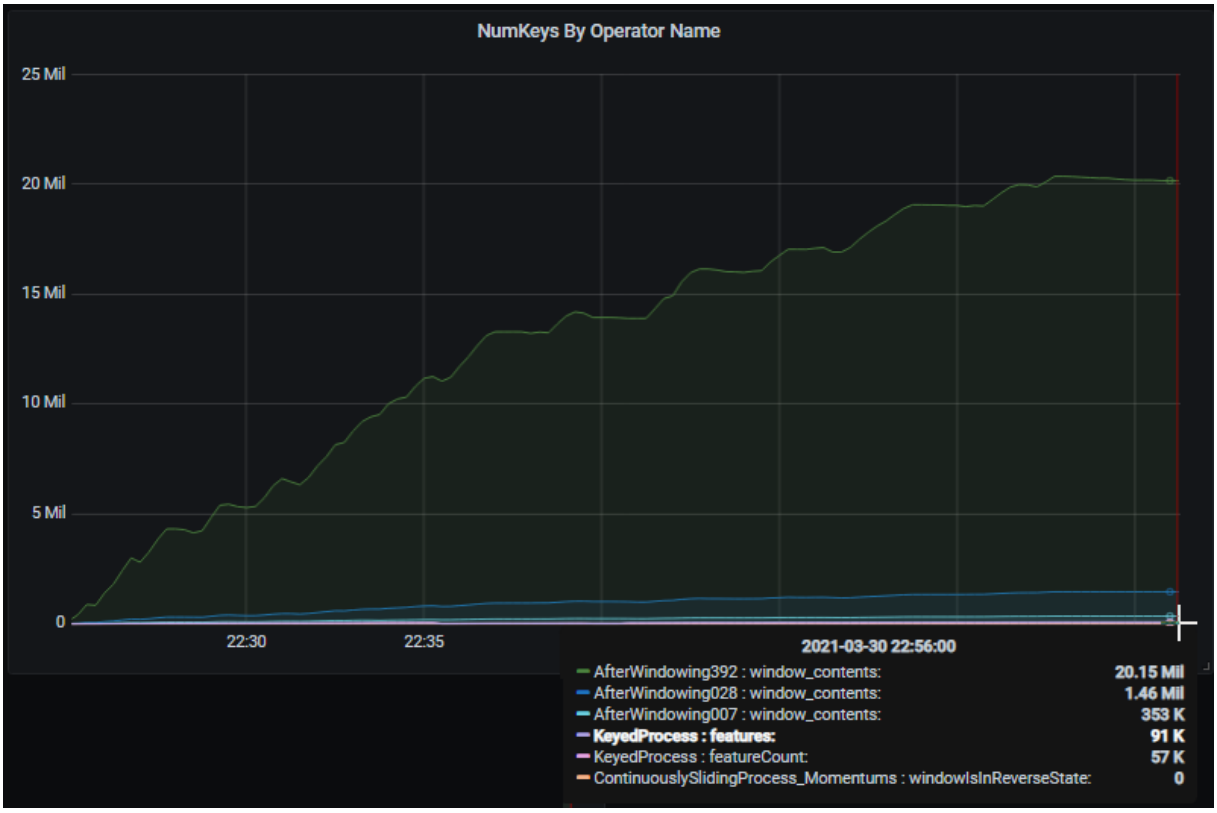

Figure 6.5: Number of Active Keys in State Backend Without Pre-Aggregation

The reduction of the number of parallel windows is a promising approach. Still, the calculation of the daily windows causes the job to slow down. The refined technique of cascading window aggregations will be evaluated in Section 6.4.1. The next optimization strategy regarding state is the switch from RocksDB to a manually managed state in Heap

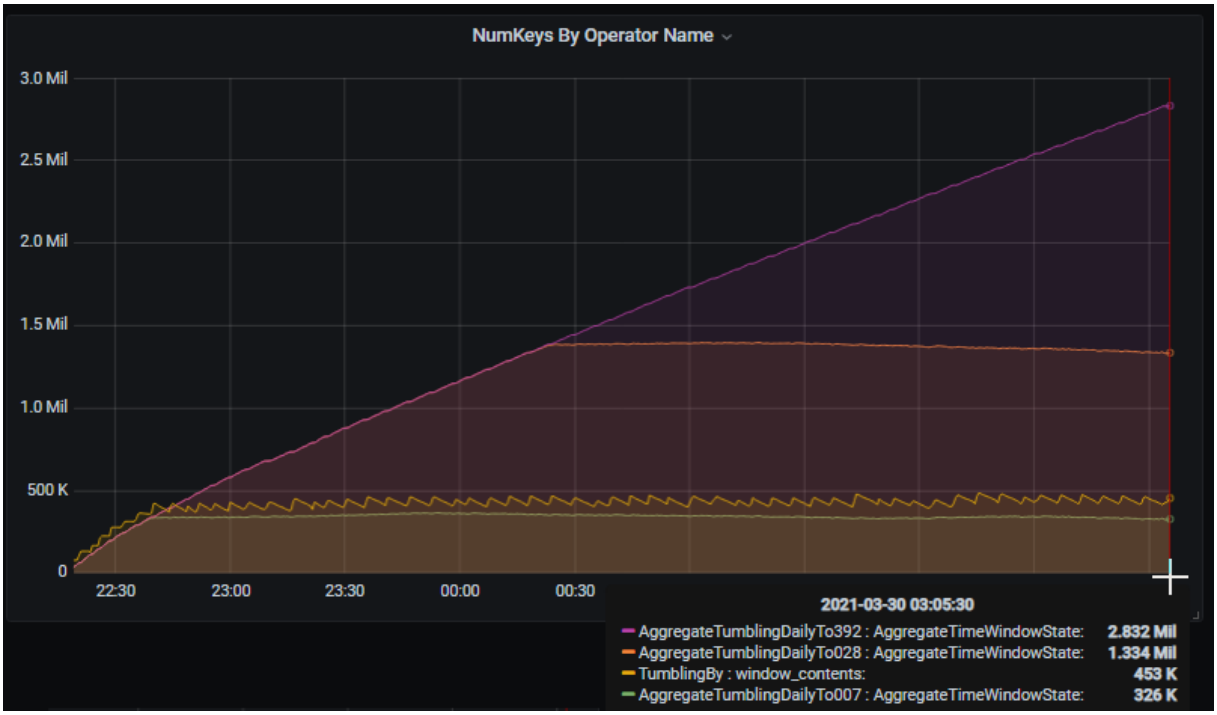

Figure 6.6: Number of Active Keys in State Backend With Daily Tumbling Windows

that uses the Flink Checkpointing.

6.2.3 Manually Managed State on Heap Data Structures

To manually manage state on heap data structures while still being able to use state persistence by means of Flink per operator Checkpointing API has a limited, yet important use case. If the key cardinalities allow it to store all operational state data inside of the memory, a switch between the usage of the state backend can be considered. In Fig. 6.7, the calculation time (right) and the state access time (left) is visualized when using the RocksDB state backend to store the operational state data. Fig. 6.8 is used to compare the numbers (*cf.* Fig. 6.7) to the algorithm that stores the operational state data in a Heap structure. What can be seen is that only half the time is needed for calculations when storing the values in the memory. Also, the overall time of the operation reduces as no state access is required. The state interaction that can be seen in Fig. 6.8 originates from saving the in-memory structure into a `UnionListState` managed by Flink when a Checkpoint occurs. As the Checkpointing is performed asynchronous, it does not count towards the latency. Nevertheless, as the `UnionListState` will grow continuously with the amount of data, the approach has to be used with caution.

Concluding Section 6.2, the reduction of the number of parallel window aggregations already has a considerable effect on the overall performance. Still, the processing of the daily tumbling windows slows down the system to some extend due to the trembling herd effect. Also, although the overall state usage could be minimized by pre-aggregating values, the overhead for calculating three different windows in parallel stays. A potential

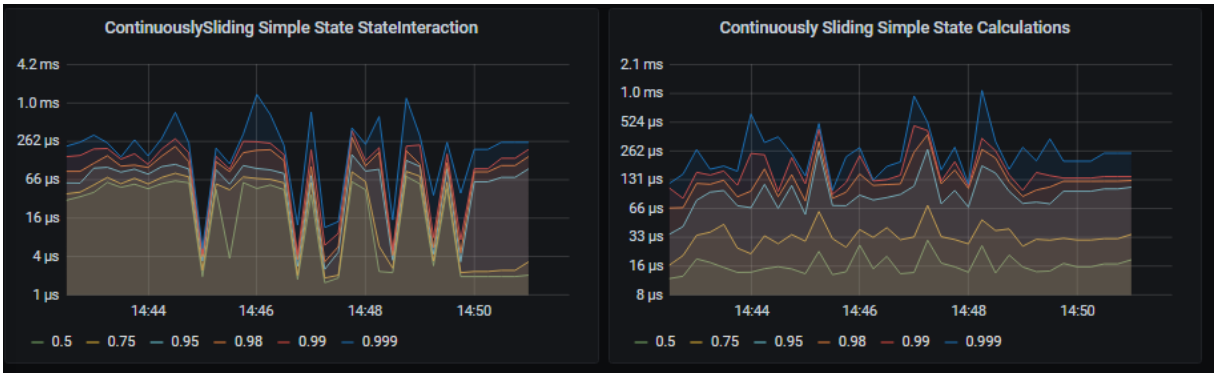

Figure 6.7: Continuously Sliding Window Using RocksDB State Backend as Storage

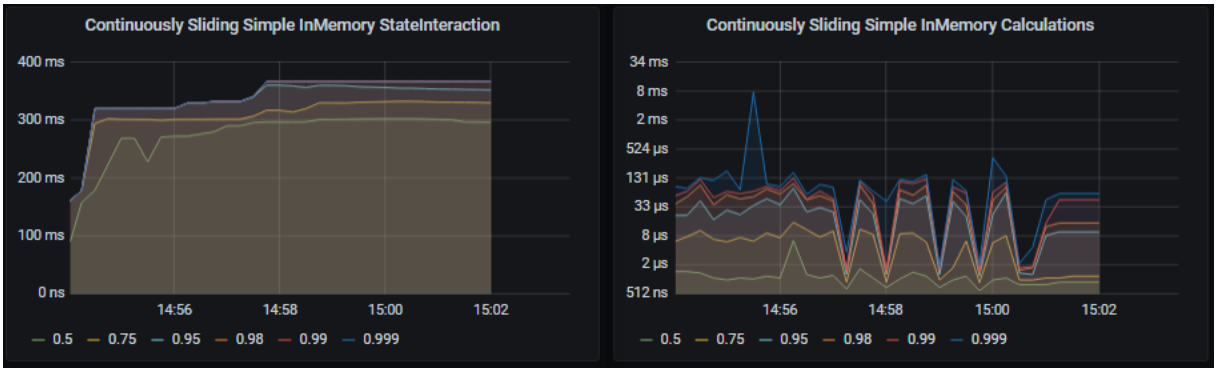

Figure 6.8: Continuously Sliding Window Using Heap as Storage

solution for this issue is the cascading window algorithm shown in Section 5.4.1, which will be evaluated in Section 6.4.1. The switch between the heap and RocksDB state backend has to be refined some more. Nevertheless, the approach yields a promising potential for future work. The RocksDB parameter tuning proved to be essential in terms of performance. As shown in the last section, the default amount of managed memory (256MB) used by the RocksDB state backend per Flink Taskmanager is not sufficient for the present job. As caching of entries in RocksDB is limited, access times to the hard disk slows down the process.

6.3 I/O Optimizations

The I/O optimization strategies aim at reducing the overall delay between operators and tasks by lowering the de/serialization effort and buffering needed. A first approach is dedicated to detecting and transforming redundant shuffle operations. The second optimization tries to apply an adapted chaining strategy to the task chaining algorithm of Flink.

6.3.1 Transforming Redundant Shuffle Operations

The detection and transformation of redundant shuffle operations has the potential to take a lot of de/serialization effort, as well as network I/O from the system. The results shown in the evaluation are astonishing. It was expected that removing redundant shuffle operations would lower the latency. Actually, the exact opposite was measured. On the one hand, Fig. 6.9 visualizes a Flink job that uses a `keyBy` statement if a `KeyedStream` is needed for the operations. On the other hand, Fig. 6.10 depicts the same job when removing the redundant shuffle operations. The average latency for the job without the optimization was only about half the time it needed when using the `reKey` functionality. This could be the result of the given setup. As only a parallelism of 1 is configured, no shuffle I/O could be measured. To mitigate this flaw, the same job was measured with a parallelism of 4, 8 and 16. The results showed the same pattern as depicted in Fig. 6.11 and 6.12. The job with the usage of `reKey` performed worse on average. Yet, with a growing parallelism, the overall difference seems to get smaller. Currently, the job is executed locally and no data has to be sent over the network. One possible explanation could be that as soon as shuffle operations over multiple network nodes are carried out, the performance of the `keyBy` operation will be worse as the data will not stay on the same machine. As the results do not conform to what was expected, further investigations need to be carried out in future work.

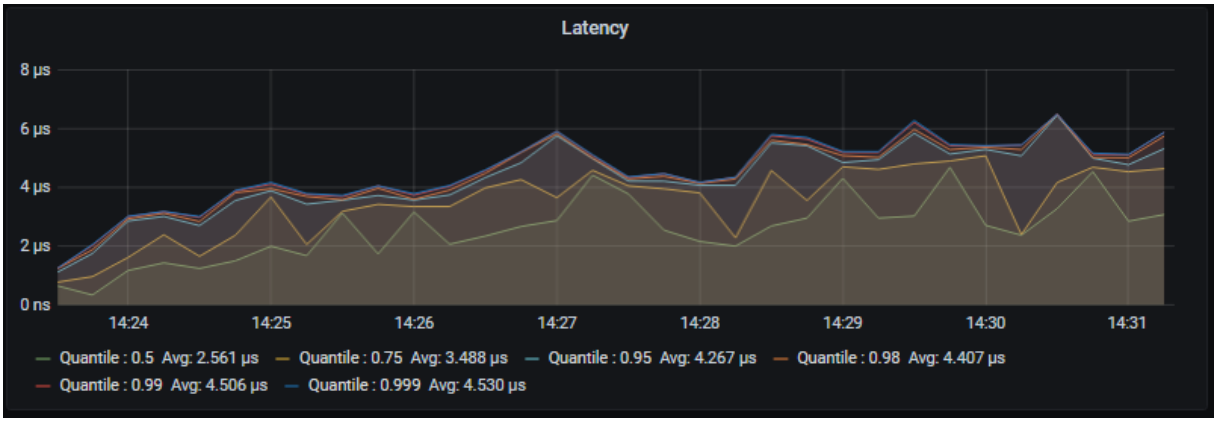

Figure 6.9: Latency Without Removing Redundant Shuffle Operations, Parallelism of 1

As a next topic for the I/O optimizations, the custom task chaining will be looked at. By reducing the amount of individual tasks, the de/serialization effort should be lowered and thus the latency.

6.3.2 Task Chaining

The extension of the task chaining was implemented in Flink 1.8, before the switch to Flink 1.12 had to be performed. Still, this should have no impact for the evaluation of the task chaining. The overall goal was to reduce the I/O by reducing the de/serialization effort, as well as the buffered I/O between different tasks. This should be achieved by decreasing the number of tasks created by the chaining strategy. The evaluation showed

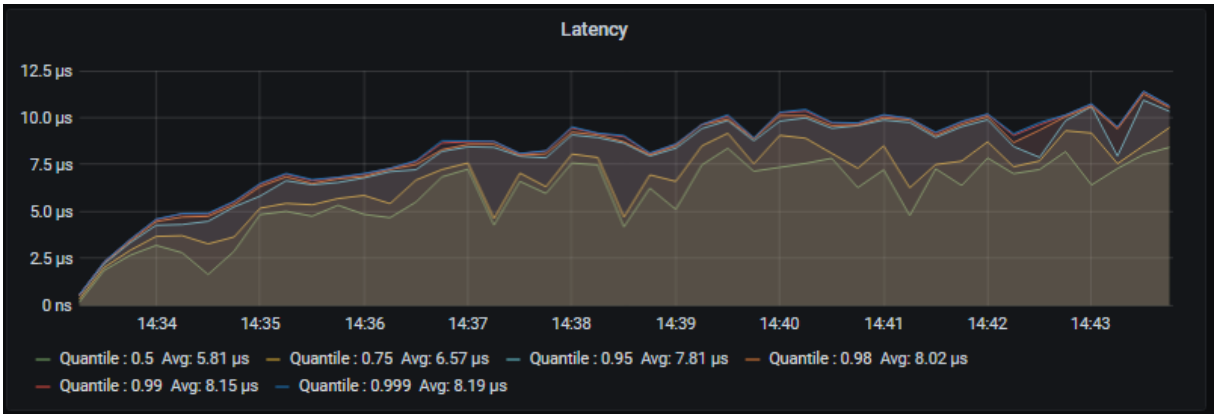

Figure 6.10: Latency With Removed Redundant Shuffle Operations, Parallelism of 1

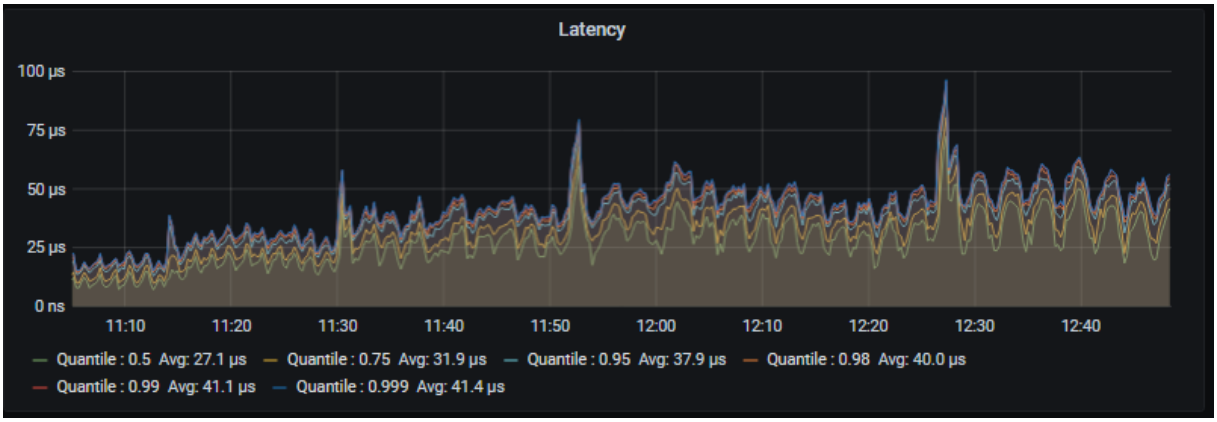

Figure 6.11: Latency Without Removing Redundant Shuffle Operations, Parallelism of 8

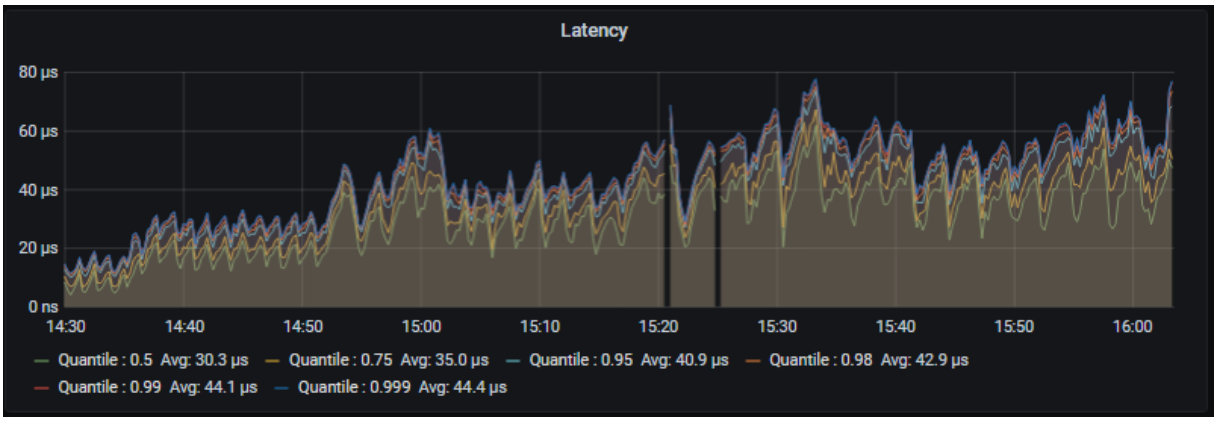

Figure 6.12: Latency With Removed Redundant Shuffle Operations, Parallelism of 8

a significant difference between the default task chaining by Flink in Fig. 6.13 and the custom implemented chaining strategy in Fig. 6.14. The custom implementation proved to be capable of cutting the pure latency measured by Flink in half. It dropped from an average of 9 μs to around 4-5 μs.

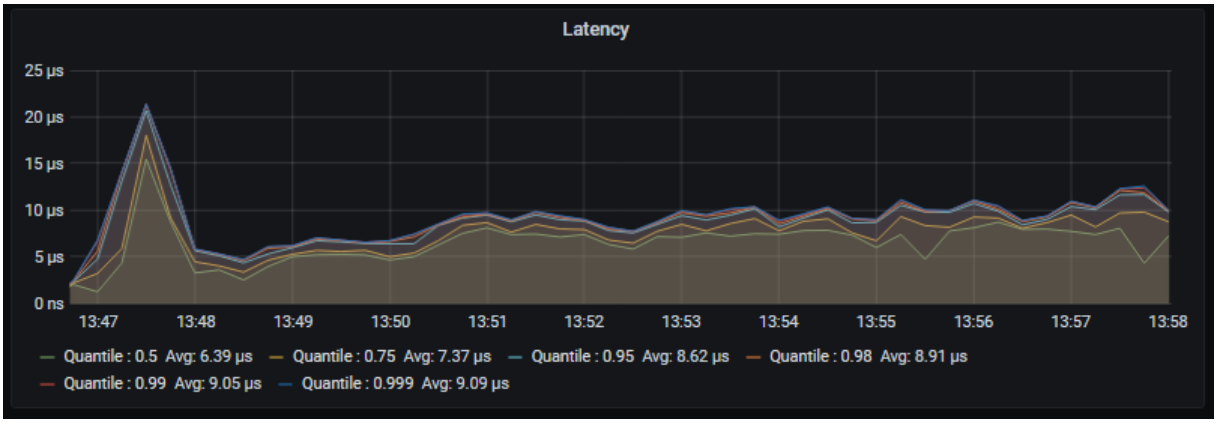

Figure 6.13: Flink Job Latency With Default Task Chaining Strategy

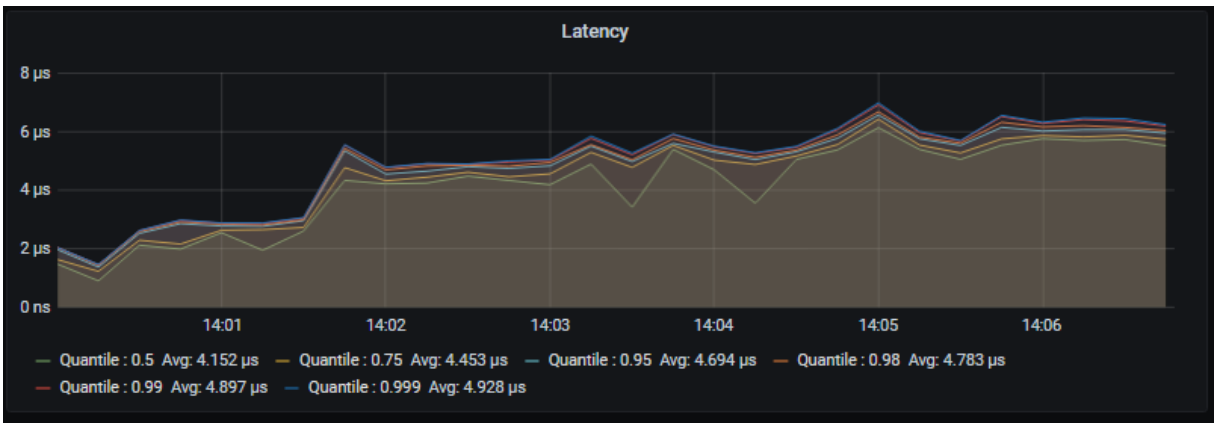

Figure 6.14: Flink Job Latency With Adapted Task Chaining Strategy

The implementation of the custom task chaining strategy lowered the latency as expected. Nevertheless, Flink 1.12 applied some changes in that field and the implemented solution would need to be adapated to those changes. Due to time restrictions and optimization strategies with more severe impact, this task is material for future work.

The I/O optimizations represent an optimization on a very fine granular level and are overshadowed by more siginificant problems (*e.g.,* unoptimized window aggregation). For future detail optimizations, fitting the task chaining strategy is a valid solution. However, the implementation has to be adapted to Flink 1.12, which requires some effort. Changing the `keyBy` to `reKey` statements and thus transforming unneeded shuffle operations in the job had the opposite effect of what was expected. This will have to be investigated further in future work.

6.4 Algorithmic Optimizations

The algorithmic optimizations offer great potential to improve the performance of the streaming job. The optimization strategy of serially processing the window computations

by reducing parallel operations is an interesting approach which should improve the time needed for aggregations by lowering the number of state backend queries. An important adaptation, which is also essential for the logic, is the handling of time-delayed streams. It is not only relevant for performance reasons, but also for the output behavior. Finally, the different implementations of the continuously sliding window algorithm are evaluated.

6.4.1 Reducing Redundant Operators

The first optimization strategy this Section evaluates is dedicated to the further development of the approach discussed in Section 6.2.2. It aims at cascading the window calculations for the individual lengths of the sliding windows. Fig. 6.16 visualizes the number of records out for the optimized cascading window calculation approach. The red line shows the output of the aggregation, green the events ingested in the pipeline. It is compared to the first implementation of the reduction of the number of parallel windows depicted in Fig. 6.15. The blue line shows the newly ingested events, whereas the outgoing events are represented by the small bumps between the spikes. The first thing that stands out is the much lower rate of outgoing events at around 200-500 events per second. In the cascading window approach, this changed. The output of the pipeline is much higher with up to nearly 40'000 records per second. Also, the spikes in the *Records Out Per Second* metric are flattened and the overall output is smoother. The cascading window calculations increase the general performance significantly. Still, it tends to get slower after some time due to the number of records to process. Nevertheless, the job does not get blocked and continues to work.

There are still some spikes occurring which need explaining. Most of the operators in the job are executed serially when using this algorithm. Therefore, the job has to spend an increasing amount of time in the cascading window aggregation since the state and the number of keys are growing. During this process, no new events will be ingested as the calculations need to take place. With higher parallelism, the duration between the spikes can be reduced.

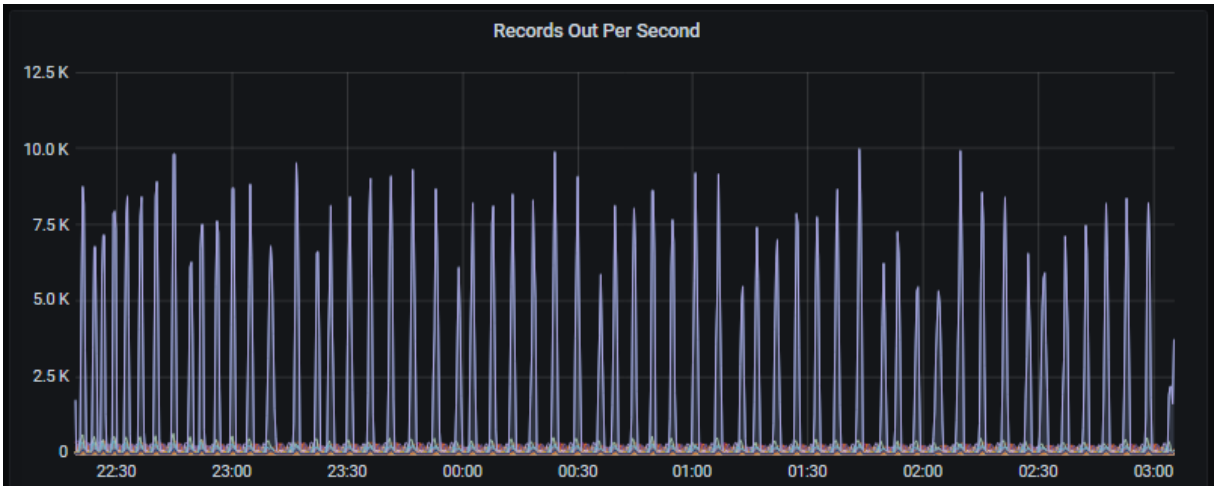

Figure 6.15: Parallel Window Reduction Approach With 9 Key Domains

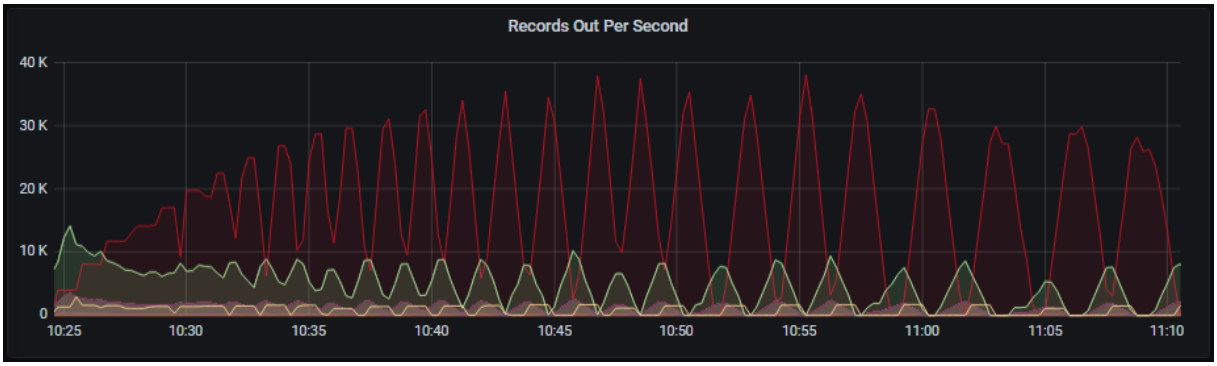

Figure 6.16: Cascading Window Aggregation With 32 Shards

The approach showed to be highly effective for the present use case. By implementing the slicing and cascading window aggregation technique the throughput of the pipeline could be increased by a factor of 100. Now that the windows can be calculated in a performant way, the approach to efficiently handle streams with different output behaviours will be evaluated.

6.4.2 Handling Delayed Streams

The handling of streams with different output behaviour was necessary due to the required combination of streams that delivered one result per day or one result per event. In Fig. 6.17 the resulting number of active keys in the state backend can be seen when the different output behaviours are not taken care of. In the join operator, a pile up of events occur as the operator has to wait until all events of the corresponding event id have been delivered. In the job that was used for the evaluation, the tumbling windows were output after one day. Thus, after a day had passed and the windows got emitted, the size of the state in the join operator dropped to some extend until later rising again (green line). Only at the time the number of buffered records in the join operator drop (*i.e.,* they are sent downstream), the subsequent operators can start with their operations and acquire state (yellow line). This piling up has negative effects on the performance of the job. It also starts to congest the whole pipeline after some time. As a solution, the `Delayed-StreamProcessFunction` was implemented. For each incoming event, the `MapState` or `ValueState` structure in the operator is checked and the last received aggregate is output. If no value for a key has been received, a null value is given out. This way, a result can be emitted for every incoming event. Hence, the pile up does not occur anymore as depicted in Fig. 6.18. The red line shows the storage of the last received values increasing per day, yellow shows the aggregated windows and the green line represents the nearly not existing buffer for the records to be joined.

This optimization approach proved to be efficient for the present semantics of the job. This strategy can be used when it is needed to join multiple streams with different output behaviour. Nevertheless, it had to be kept in mind that potentially outdated results are shown until a new window is emitted. Thus, the approach has to be used carefully and

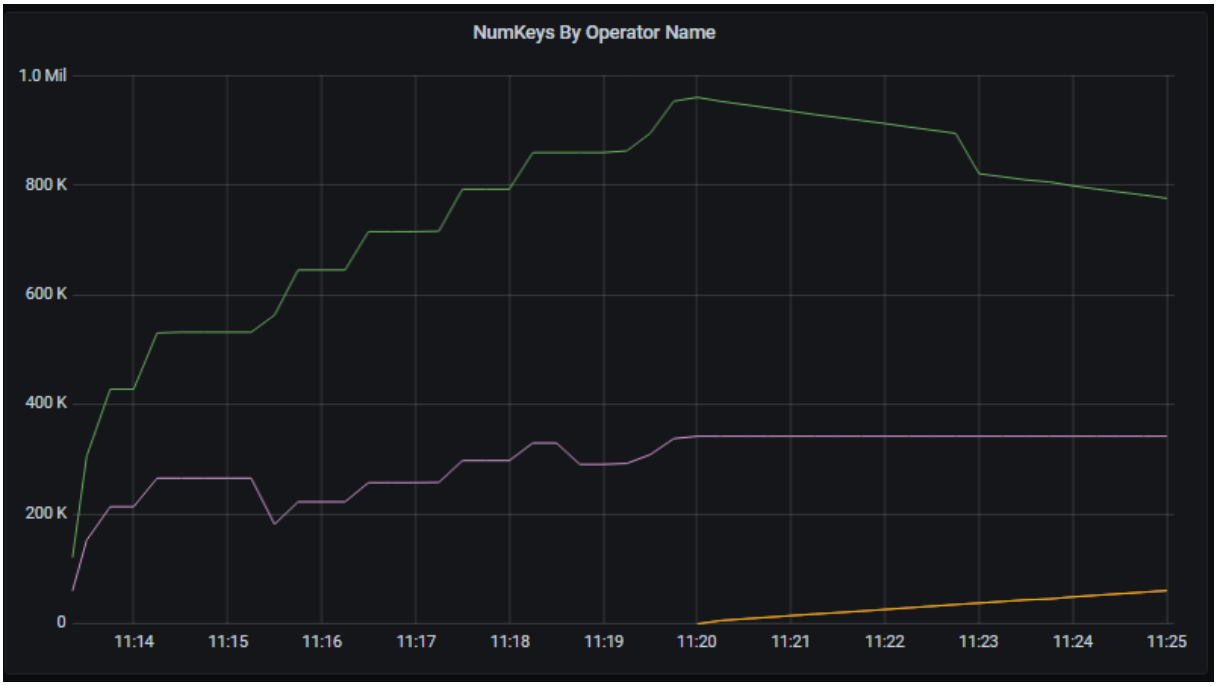

Figure 6.17: Number of Active Keys in State Behaviour of The Baseline

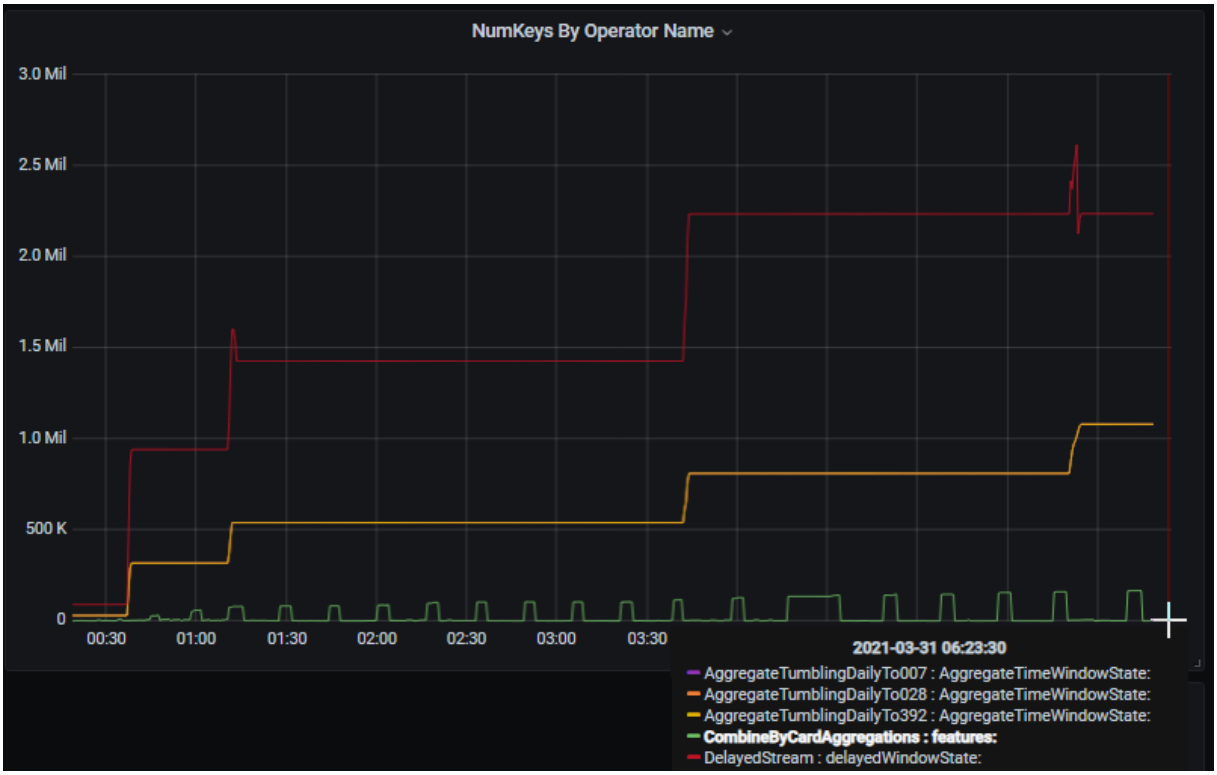

Figure 6.18: Number of Active Keys in State Behaviour With Activated Delay Handling

only when the semantics of the job allow it. As a last algorithmic optimization, the continuously sliding windows will be evaluated in the next Section.

6.4.3 Continuously Sliding Windows

The continuously sliding windows were an important topic of the present work. They represent a requirement that is not inherently thought of in Flink in this way. Different algorithms had to be elaborated until a promising approach was found. The first approach tried to handle the continuously sliding windows in a naive way. It relied on a `ProcessFunction` that added an event to the `MapState` when it was ingested and removed it with a timer as soon as it got irrelevant for the time horizon. The second approach tried to adapt the `WindowOperator` and allow to distinguish between an execution of the `ProcessFunction` due to a new incoming event, or the trigger of an event timer. The last option was an optimization of the aggregation algorithm used for the second approach and aimed at reducing the necessary recalculation in the incremental accumulation of the stored events. To see the impact of the different algorithms better, the aggregations were made with a key that holds extensive data and consequently many aggregated events per base key window. In the current context, such a key could be the country attribute of an event.

One-Stage Algorithm With Process Function

The one-stage algorithm showed an acceptable performance when using a single key that holds sparse data. With a growing number of values per window to aggregate, the performance degrades quickly as visualized in Fig. 6.19. In a worst case scenario, the calculation of the aggregates took on average 212 milliseconds. On the positive side, the state interactions stayed more or less constant with on average at most 2.29 milliseconds.

Two-Stage Algorithm With AllStateWindow

The initial two-stage algorithm that extended the `WindowOperator` used a slicing technique to eliminate some of the incremental recalculations by pre-aggregating into smaller windows. This caused a great improvement of the performance in terms of the calculations. The maximum needed time for aggregations dropped from 212ms (*cf.* Fig. 6.19) to 17.5ms. Again, the time used for state interactions such as read and write stayed nearly constant with an average maximum of 1.70ms.

Reverse Aggregation

Finally, another big improvement could be made by applying the reverse aggregation algorithm to the two-stage windowing approach as described in Section 5.4.3. It caused not only the calculation time to drop from 17.5ms (*cf.* Fig. 6.20) to at most 4.1 ms on average, but also the state interactions could be further lowered to 973 μs

The reverse aggregation algorithm for the continuously sliding windows is currently the way to go. It offers the by far best performance for keys with sparse and extensive data. The one-stage approach could be used for cases where the overhead implemented in the

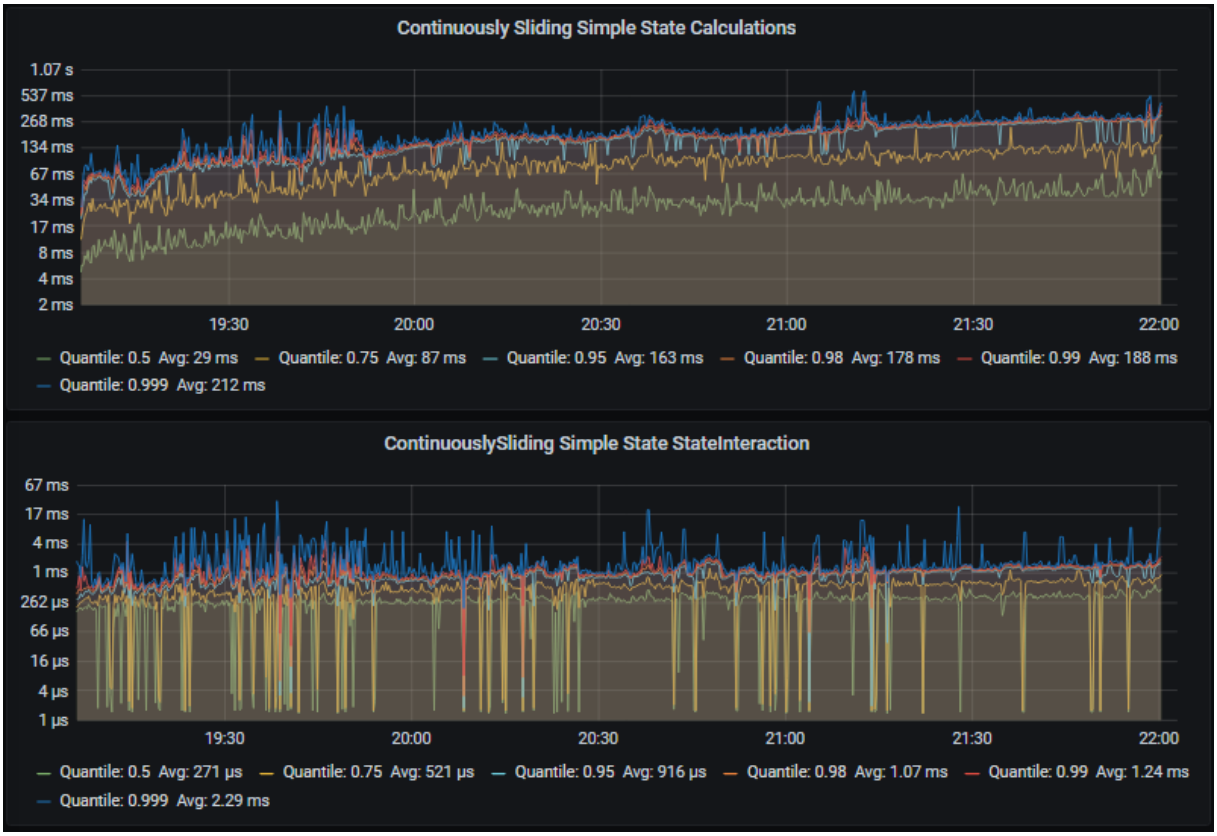

Figure 6.19: Calculation and State Interaction Duration of The One Stage Algorithm

two-stage algorithm is not needed. Yet, the weaknesses of the implementation have to be kept in mind.

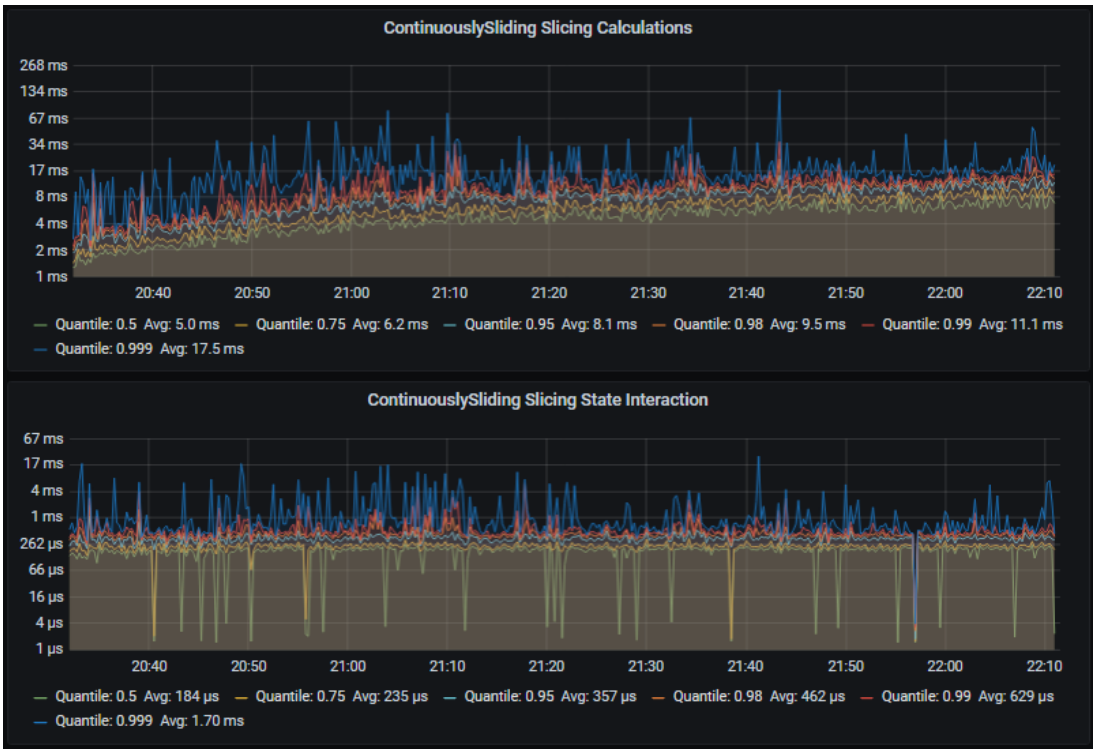

Figure 6.20: Calculation and State Interaction Duration of The `AllStateWindowOperator`

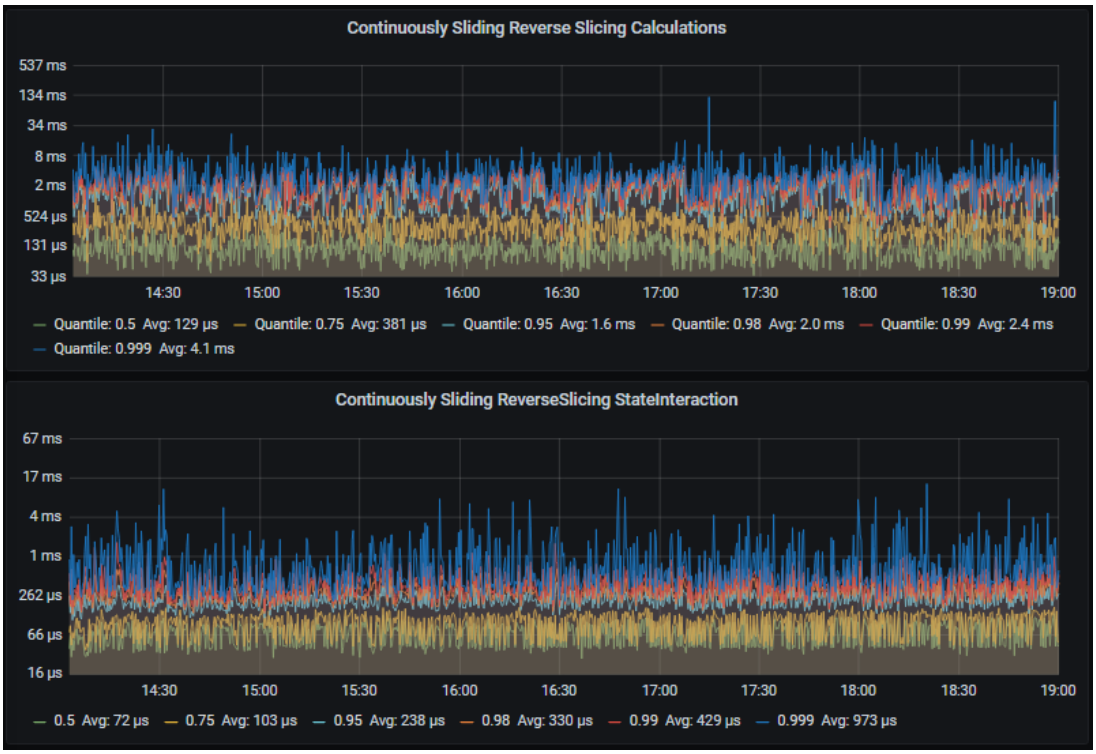

Figure 6.21: Calculation and State Interaction Duration of Reverse Aggregation Approach

Chapter 7

Summary and Conclusions

As described in Chapter 1, the overall goal of this thesis was to study and optimize a data preparation system for fraud prevention in terms of latency, jitter and throughput. The development of the existing system started in the previous master project and was refined before the start of the master thesis.

In a first part, a literature research had to be performed in order to list credit card fraud detection and prevention approaches. Furthermore, algorithms and strategies to optimize a streaming system while adhering to certain constraints have to be covered. In a second part, the existing system had to be analyzed for potential weaknesses. A system to measure the latency, jitter and throughput had to be elaborated and installed. The gained insights would later be needed in the evaluation of the optimization strategies to compare the approaches to a baseline. The third part was dedicated to the design and implementation of the envisioned optimization strategies. Finally, the thesis provides an evaluation of the optimizations and lists their advantages and limitations. The elaborated strategies all have their own field of application. Not every optimization is suited for every type of Flink job or data characteristic. In the following section, the approaches are summarized and statements are made in which case an optimization strategy could be helpful.

7.1 Optimization Strategies

The following Section provides a short summary on the different optimization strategies implemented and provides a statement about the applicability of the approaches.

7.1.1 RocksDB Parameter Tuning by [4]

If the resource requirements of a Flink job regarding the RocksDB cache size are not fulfilled, the performance will degrade severely. To mitigate this issue, parameter tuning can be taken into consideration as a first step. For the present case, the increase of the

background thread number asssigned to RocksDB as well as the overall managed memory size of Flink yielded the biggest effect. Without the parameter tuning, the implemented job was not usable.

Applicability: **Flink jobs in general.**

7.1.2 Reduction of Number of Parallel Windows

The reduction of the number of parallel windows by introducing slicing into tumbling windows and then aggregating the slices is a simple and easy to implement technique. It offers a considerable increase to performance and reduction of overall state size. The **throughput could be increased from a few 100 to a few 1000 events on average**. Furthermore, the **operational state size was reduced from 20 million keys to 3 million**. As the calculation of the parallel window aggregations still causes an omittable overhead, the cascading window aggregation approach shown in 7.1.6 is to be preferred.

Applicability: **Can be used to increase the performance of a streaming job with a high number of concurrently active windows. Nevertheless, it is more beneficiary to use the cascading window aggregation algorithm (*cf.* Section 7.1.6).**

7.1.3 Manually Managed State on Heap Data Structures

The switch between the RocksDB state backend and the manually managed Heap data structures has a limited use case by now. It provides the user with the possibility to decide whether the data should be stored in-memory or in the RocksDB state backend. This way, a trade-off can be made between state size and access time. To use the approach, the cardinalities of the expected data has to be known as too much data in memory will cause the garbage collector to slow down the system. Also, the developer has to know the exact maximum size the state can grow to, otherwise the job is destined to fail due to a memory shortage. Nevertheless, in the evaluation it was shown that the approach is able to **halve the time needed for calculations** (*i.e.,* from approximately 262μs to 131μs). Also, when using the Heap data structure, **nearly no time for state access is needed**. For future work, a dynamic switch between the two variants or a hybrid approach with hot (in-memory) and cold (RocksDB state backend) windows could be considered.

Applicability: **Operators with clear operational state size boundaries, that need to rely on fast calculations.**

7.1.4 Transforming Redundant Shuffle Operations

This approach aimed at reducing the latency by reducing the number of shuffle operations by reinterpreting a stream as an already partitioned one. What appears to be logical had the opposite effect. When applying the `reKey` function, the latency increased. In the current evaluation setup, the network I/O could not be measured as no real cluster

environment was ready to use. This could potentially have a great impact. Therefore, these negative results must be viewed with caution and need to be further analysed in future work.

Applicability: **Currently not usable as it causes the latency to increase.**

7.1.5 Task Chaining

The extended task chaining aims at reducing the I/O by creating a coarser task granularity. Everytime a de/serialization can be omitted, the performance should be increased. The results of the task chaining delivered positive results. The evaluation showed that the extended task chaining strategy **halved the latency** of the pipeline used from an **average of 9μs to 4-5μs**. Nevertheless, the approach has two weaknesses. First, the implementation was done in Flink 1.8. However, in Flink 1.12 changes were made in this exact region. On the one hand, this shows that a hot topic has been touched. On the other hand, it would have to be analyzed to what extent the custom task chaining is now redundant with the official release. Also, the custom task chaining implementation would need to be adapted to Flink 1.12 as some classes have been changed.

Applicability: **Results are promising and the approach lowers the latency. However, currently not usable with Flink 1.12.**

7.1.6 Reducing Redundant Operators

The further development of the reduction of the number of parallel windows in Section 7.1.2 lead to the cascading window aggregation algorithm. It exploits the characteristics of the RocksDB `MapState` structure and the inherent canonical ordering of the stored data. By using an iterator, all values for a certain timeframe can be aggregated incrementally. For all windows that have to be calculated in a cascading way, the records only need to be retrieved once. Additionally, as soon as the first record has been found, the following entries can be returned with constant complexity as the iterator is already at the correct, in-order position. By assigning the stored values into corresponding window time-frame groups, the smaller windows can be merged to form larger ones. This leads to fewer and more efficient state backend accesses and thus, a better overall performance. By using the cascading window aggregation, **the throughput could be increased from a few 100 to about 40'000 events per second**, which corresponds to an increase by a **factor of 100**. Also, by removing redundant pipelines and operators, the overall number of nodes in the execution plan could be lowered by a factor of 60 (*i.e.,* baseline implementation had 1800 nodes, cascading windowing approach approximately 30). This in turn has a positive effect on the general latency of the pipeline as less potential network I/O and de/serialization effort is generated.

Applicability: **Efficiently aggregate multiple windows serially with possibility to use the same pipeline for different key domains.**

7.1.7 Handling Delayed Streams

The handling and combination of streams with different output behaviour was of great use for the present work. Due to the piling up of the individual records in the join operator of the split/join pattern, the job got congested after some time. With increasing size of the time difference between each output, the piling up gets more severe. For instance, if one operator outputs one record per incoming event and another operator one record per 10 days, the join operator needs to buffer all the records for 10 days for joining the former to the latter ones. By using a `CoProcessFunction` the last received record can be buffered. For every new event, this buffered record is then propagated. If no previous record is available, a null record gets emitted. This way, the output behaviour can be controlled. It has to be kept in mind that this procedure only works if the semantics of the job allow for it.

Applicability: **Solves the problem of piling-up records in join operators.**

7.1.8 Continuously Sliding Windows

The continuously sliding window algorithm allows a user to define windows that always contain data for a definable time horizon from now. The challenging aspect is the continuous addition and removal of records to and from the window. What can be easily solved with a `ProcessFunction`, gets increasingly unsuitable with a larger amount of data. As a solution, a two stage algorithm was introduced. It catches up on the idea of stream slicing introduced by [10]. A first approach of this algorithm already delivered promising results. The further implementation focused on an optimization for the continuously sliding window aggregation algorithm.

At a certain point in time, it is highly probable that records will only be removed but none added to the window. In the first approach, for each removed record all the values of the window would be aggregated again to get the current result. In the adapted approach, the values in the `MapState` are reversed and a one-time aggregation with a bottom-up approach happens. This way, when a record is removed, only the corresponding pre-aggregated value needs to be emitted. Again, the characteristics of the RocksDB state backend and the inherent canonical ordering gets used to benefit from a serial processing of the window values. Additionally, by using a slicing technique, the results will stay numerically stable as potential errors get removed from the result continuously. In the evaluation it was shown that the usage of the reverse aggregation for the continuously sliding window algorithm could **lower the latency from 212ms to 4.1ms**, which is a reduction by a **factor of 50**.

Applicability: **Efficient handling of continuously sliding windows with sparse and extensive data.**

7.2 Conclusions And Guidelines

This Section provides the main key points regarding optimization of a streaming job, that resulted from this work.

- **Slicing** techniques help to reduce the number of concurrently active windows.
- **Generalize** interfaces to reduce redundant pipelines for different data types.
- **Cascading** window aggregation reduces the number of concurrent queries to the state backend.
- **Cardinalities** of the expected keys and data have to be known. A job with too few allocated resources is bound to fail.

Except for the transformation of redundant shuffle operations, all other listed optimization strategies proved to be of great efficiency. A first optimization step that always should be considered when state is involved is the parameter tuning. The setting of the managed memory size is of central importance, also when no RocksDB state backend is used. Additionally, if RocksDB is used, the increase of the background thread number allows for further performance gains.

To the best of the authors knowledge, the cascading window calculation and the reverse aggregation of continuously sliding windows represent a novel approach in the Flink environment. By using the canonical ordering provided by RocksDB, an efficient incremental roll-up of the values stored in a window can be achieved. The usage of slicing techniques further increases this effect. By implementing the various presented strategies, the throughput of the initial baseline implementation could be increased from a few hundred events out per second, to up to 40'000. This is equivalent to an **increase in throughput by a factor of more than 100**. By using the **reverse aggregation for continuously sliding windows**, the **latency** of this operation could be **reduced from 212ms to 4.1ms** on average. This is a reduction by a factor of 50.

Nevertheless, measuring the complete latency of a pipeline only makes sense when the whole history of the maximum defined time span (*e.g.,* 392 days) has been processed and the system is in a *live* mode. Still, it can be stated that by reducing the number of parallel window aggregations, and thus the number of operators, also the latency will greatly benefit from this optimization. By implementing a way to run multiple key domains in one pipeline, the latency further benefits from a leaner architecture and less de/serialization effort. In total, **the number of operators could be lowered from 1800 nodes to approximately 30**, which is a reduction by factor 60. The **extended task chaining** remains an open, yet interesting topic. It proved to work and **lowered the pure latency of the pipeline by 50%**. Nevertheless, it needs an adaption to the newest Flink version. As this optimization technique operates in the microsecond range, it represents a fine granular optimization. The benefit of this optimization can be lost if other, more performance-relevant problems are present. Therefore, it only makes sense to perform an I/O optimization if everything else is running optimally.

7.3 Future Work

There exist many topics for future work in this area. First, the implementation of the feature generation job for scenario 3 poses some additional challenges. On the one hand, the keys that need to be added further enlarge the key space and the overall data usage of the state backend. Also, the unique count aggregation for keys with extensive data sets is a difficult topic for incremental aggregations. It has to be found out if algorithms such as HyperLogLog [57] are useful in this field or just further complicate the issue. Also, scenario 3 includes the issue with reclassifications of values for a time-frame of at least 28 days. As it was shown, although many optimizations have been applied to the job and it does not get blocked anymore, the performance degrades over time. Nevertheless, low latency will only be needed when authorizations are aggregated *live* and not while initially rolling up the history. A potential approach to solve this challenge is to initially calculate the state as a batch job. As soon as the current timestamp is reached, the job would switch to the streaming-mode. The state that was calculated in the batch-mode could then be used as a starting point for test scenarios. Other topics are further investigations regarding the latency differences between the `keyBy` and `reKey` functions, the adaption of the custom task chaining to Flink 1.12 as well as the implementation of a dynamic switch between the RocksDB state backend and the manually managed heap data structure.

Bibliography

[1] HSN Consultants, Inc., "The Nilson Report," 2021. https://nilsonreport.com/, Last visit April 10, 2021.

[2] Capgemini, "Credit Card Transaction Fraud and Mitigation Trends," 2017. https://www.capgemini.com/wp-content/uploads/2017/07/Credit_Card_Transaction_Fraud_and_Mitigation_Trends.pdf, Last visit April 10, 2021.

[3] B. Gehring, S. Graf, and T. Trutsch, "Swiss Payment Monitor 2020," 2021. https://en.swisspaymentmonitor.ch/archiv, Last visit April 10, 2021.

[4] Littlemagic's blog, "Tuning practice of Flink rocksdb state back end parameters," 2020. https://developpaper.com/tuning-practice-of-flink-rocksdb-state-back-end-parameters/, Last visit April 10, 2021.

[5] Atlas VPN, "Financial Fraud Reports in the US," 2021. https://atlasvpn.com/blog/financial-fraud-reports-in-the-us-jumped-by-104-in-2020-q1, Last visit April 10, 2021.

[6] D. Bünzli and N. Berni, "Design and implementation of a data preparation system for fraud scoring," May 2020. https://files.ifi.uzh.ch/CSG/staff/scheid/extern/theses/MAP-D-Bunzli-N-Berni.pdf, Last visit April 10, 2021.

[7] Viseca Card Services SA, "Viseca," 2021. https://www.viseca.ch/en/, Last visit April 10, 2021.

[8] The Apache Software Foundation, "Apache Flink," 2020. https://flink.apache.org/flink-architecture.html, Last visit December 21, 2020.

[9] Facebook Open Source, "RocksDB," 2021. https://rocksdb.org/, Last visit April 10, 2021.

[10] J. Traub, P. Grulich, A. R. Cuellar, S. Bress, A. Katsifodimos, T. Rabl, and V. Markl, "Efficient window aggregation with general stream slicing," in *22th International Conference on Extending Database Technology (EDBT). International Conference on Extending Database Technology (EDBT-2019), 22th, March 26-29, Lisbon, Portugal*, OpenProceedings, 2019.

[11] E. de Nooij, "Streaming machine learning models: How ing adds fraud detection models at runtime with apache flink," 2017. https://www.ververica.com/blog/real-time-fraud-detection-ing-bank-apache-flink, Last visit April 10, 2021.

[12] S. Maniraj, A. Saini, S. Ahmed, and S. Sarkar, "Credit card fraud detection using machine learning and data science," *International Journal of Engineering Research and*, vol. 08, 09 2019.

[13] A. K. Rai and R. K. Dwivedi, "Fraud detection in credit card data using unsupervised machine learning based scheme," in *2020 International Conference on Electronics and Sustainable Communication Systems (ICESC)*, pp. 421–426, 2020.

[14] Rajeshwari U and B. S. Babu, "Real-time credit card fraud detection using streaming analytics," in *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, pp. 439–444, 2016.

[15] F. Carcillo, A. D. Pozzolo, Y. L. Borgne, O. Caelen, Y. Mazzer, and G. Bontempi, "SCARFF: a scalable framework for streaming credit card fraud detection with spark," *CoRR*, vol. abs/1709.08920, 2017.

[16] Amazon Web Services, "What is streaming data?," 2020. https://aws.amazon.com/streaming-data/, Last visit December 21, 2020.

[17] Confluent, "Data Streams Explained," 2020. https://www.confluent.io/learn/data-streaming/, Last visit December 21, 2020.

[18] The Apache Software Foundation, "Apache Hadoop," 2021. https://hadoop.apache.org/, Last visit April 10, 2021.

[19] The Apache Software Foundation, "Apache Mesos," 2021. http://mesos.apache.org/, Last visit April 10, 2021.

[20] Kubernetes Authors, "Kubernetes," 2021. https://kubernetes.io/, Last visit April 10, 2021.

[21] Jun Qin, "Using RocksDB State Backend in Apache Flink: When and How," 2021. https://flink.apache.org/2021/01/18/rocksdb.html.

[22] T. Akidau, *Streaming systems : the what, where, when, and how of large-scale data processing*. Beijing: O'Reilly, first edition. ed., 2018.

[23] The Apache Software Foundation, "Apache ZooKeeper," 2021. https://zookeeper.apache.org/, Last visit April 10, 2021.

[24] Stefan Richter, "3 differences between Savepoints and Checkpoints in Apache Flink," 2018. https://www.ververica.com/blog/differences-between-savepoints-and-checkpoints-in-flink, Last visit April 10, 2021.

[25] QOS, "Simple Logging Facade for Java SLF4J," 2019. http://www.slf4j.org/, Last visit April 10, 2021.

[26] Oracle, "Java Management Extensions (JMX)," 2021. https://www.ing.de/, Last visit April 10, 2021.

[27] Prometheus, "Prometheus - From metrics to insight," 2021. https://prometheus.io/, Last visit April 10, 2021.

[28] F. Hueske and V. Kalavri, "Stream processing with apache flink: Fundamentals, implementation, and operation of streaming applications," 2019.

[29] Yueh-Hsuan Chiang, "Column Families," 2018. https://github.com/facebook/rocksdb/wiki/Column-Families, Last visit April 10, 2021.

[30] Yue Meng, "In-depth analysis of flink job execution: Flink advanced tutorials," 2020.

[31] Apache Flink, "Physical Partitioning," 2020. https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/, Last visit April 10, 2021.

[32] L. Biao, "Metrics principles and practices: Flink advanced tutorials," 2020. https://www.alibabacloud.com/blog/metrics-principles-and-practices-flink-advanced-tutorials_596634.

[33] Viseca, "Aduno Group becomes Viseca," 2020. https://www.viseca.ch/en/corporate/viseca-holding/media/2020/aduno-group-becomes-viseca, Last visit February 21, 2021.

[34] The Apache Software Foundation, "Apache Avro," 2021. http://avro.apache.org/, Last visit April 10, 2021.

[35] Stefan Richter, "How to manage your RocksDB memory size in Apache Flink," 2020. https://www.ververica.com/blog/manage-rocksdb-memory-size-apache-flink, Last visit April 10, 2021.

[36] The Apache Software Foundation, "Apache Spark," 2021. https://spark.apache.org/, Last visit April 10, 2021.

[37] Docker, Inc, "Docker," 2021. https://www.docker.com/, Last visit April 10, 2021.

[38] J. Gama and P. Brazdil, "Cascade generalization," *Machine Learning*, vol. 41, pp. 315–343, 12 2000.

[39] The Apache Software Foundation, "Apache Cassandra," 2021. https://cassandra.apache.org/, Last visit April 10, 2021.

[40] A. Correa Bahnsen, D. Aouada, A. Stojanovic, and B. Ottersten, "Feature engineering strategies for credit card fraud detection," *Expert Systems with Applications*, vol. 51, pp. 134–142, 2016.

[41] ING Groep N.V, "ING Bank," 2021. https://www.ing.de/, Last visit April 10, 2021.

[42] D. Sanger and N. Perlroth, "Bank Hackers Steal Millions via Malware," 2015. https://www.nytimes.com/2015/02/15/world/bank-hackers-steal-millions-via-malware.html, Last visit April 10, 2021.

[43] K. AG, "KNIME - Machine Learning Models," 2021. https://www.knime.com/, Last visit April 10, 2021.

[44] Data Mining Group, "Persistent Model Markup Language (PMML)," 2021.

[45] M. Fowler, *Domain-specific languages*. The Addison-Wesley signature series, Upper Saddle River, NJ: Addison-Wesley, 2010.

[46] J. Traub, P. Grulich, A. R. Cuellar, S. Bress, A. Katsifodimos, T. Rabl, and V. Markl, "Scotty: Efficient window aggregation for out-of-order stream processing," in *Proceedings of the 34th IEEE International Conference on Data Engineering. IEEE International Conference on Data Engineering (ICDE-2018), April 16-20, Paris, France*, IEEE, 2018.

[47] Brice Bingman, "Poor performance with Sliding Time Windows," 2018. https://issues.apache.org/jira/browse/FLINK-6990, Last visit April 10, 2021.

[48] Jark Wu, "Improve performance of Sliding Time Window with pane optimization," 2020. https://issues.apache.org/jira/browse/FLINK-7001, Last visit April 10, 2021.

[49] Syinchwun Leo, "Lightweight Event Time Window," 2020. https://issues.apache.org/jira/browse/FLINK-5387, Last visit April 10, 2021.

[50] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl, "Cutty: Aggregate sharing for user-defined windows," in *CIKM 2016 - Proceedings of the 2016 ACM Conference on Information and Knowledge Management*, vol. 24-28-October-2016, (United States), pp. 1201–1210, Association for Computing Machinery (ACM), Oct. 2016.

[51] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams," *SIGMOD Rec*, vol. 34, pp. 39–44, Mar. 2005.

[52] debs.org, "DEBS 2013 Grand Challenge: Soccer monitoring," 2016. https://debs.org/grand-challenges/2013/, Last visit April 10, 2021.

[53] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[54] Flink, "Memory tuning guide," 2021. https://ci.apache.org/projects/flink/flink-docs-release-1.12/deployment/memory/mem_tuning.html/, Last visit April 10, 2021.

[55] J. Garman, *Kerberos : the definitive guide*. Beijing ;: O'Reilly, first edition. ed., 2003.

[56] Teng Hu, "Stagger TumblingProcessingTimeWindow processing to distribute workload," 2019. https://issues.apache.org/jira/browse/FLINK-12855, Last visit April 10, 2021.

[57] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm," in *AofA: Analysis of Algorithms* (P. Jacquet, ed.), vol. DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07) of *DMTCS Proceedings*, (Juan les Pins, France), pp. 137–156, Discrete Mathematics and Theoretical Computer Science, June 2007.

Abbreviations

MCC	Merchant Category Code
Auth	Authorisation
Trx	Transaction
CardId	Card Identification Number
ML	Machine Learning
RBT	Red Black Tree
ADT	Abstract Data Type
DSL	Domain Specific Language
TE	Temporal Event
BST	Binary Search Tree
CDF	Cumulative Distribution Function
LSM	Log-Structured Merge Tree
ETL	Extract, Transform, Load
UI	User Interface
DAG	Directed Acyclic Graph
IO	Input/Output
SDP	Streaming Data Platform
ACID	Atomicity, Consistency, Isolation, Durability
RDDs	Resilient Distributed Datasets
ATM	Automated Teller Machines
D0	Day Zero Windows
JSON	JavaScript Object Notation
HRK	High Risk Keys

Glossary

Fraud Scoring Solution that scores authorizations in near real-time using machine learning algorithms in order to detect fraud attempts.

Authorisation Is a check from the merchant, which verifies the chargeability of the credit card and makes a pre-booking of a certain transaction-related amount. For example, half, a quarter or the whole amount of a transaction can be reserved.

Transaction Booking of a purchase. Authorisations are provisional, transactions are a definitive debit of the customer account.

Apache Flink Stream Processing Framework used for this project.

Red Black Tree Self-balancing sub-type of a binary search tree.

Poisson Point Process a stochastic process named after Siméon Denis Poisson.It is a renewal process whose gains are Poisson-distributed.

Monad Algebraic structure used to isolate side effects and facilitate the composability of calculations and computations.

Fraud Scoring Fraud prevention system based on machine learning, which analyzes authorizations and assigns a certain probability of fraud to card usages.

Scala Functional and object-oriented programming language used for this project

Apache Avro Is a remote procedure call and serialization framework developed as part of Apache Hadoop. It uses a compact binary format for serialization.

Continuously Sliding Windows Custom windowing algorithm that allows to keep the last **x** time units (*e.g.,* seconds, minutes, hours, days) in a window at any time with millisecond precision

Unique Count Aggregate Aggregation function that counts the unique appearances of a defined attribute in a window.

Oldest/Recent Aggregate Aggregation function that determines the first and last occurrence of an event in a window.

Momentum Aggregate Aggregation function that incrementally calculates the higher order statistics (*e.g.,* sum, mean, standard deviation, skewness).

Split/Join Pattern Pattern used for combining calculated features with the initial event they were extracted from.

Delayed Stream Streams with different output / timing behaviour.

Two-Factor Authentication Access authorization is verified by several independent characteristics

List of Figures

Listings

Appendix A

Contents of the CD

The attached CD contains the following files and directories:

- `Abstract.txt`: Plain text version of the English abstract
- `Thesis.pdf`: PDF version of the thesis
- `Latex.zip`: Zip containing the LaTeX sources of the Report
- `Code.zip` Zip containing the code of the thesis
- `Zusammenfassung.txt`: Plain text version of the German abstract