



University of
Zurich^{UZH}

Design and Implementation of a Data Preparation System for Fraud Scoring

Dominik Bünzli and Noah Berni
Zürich, Switzerland
Student ID: 14-707-111, 14-728-166

Supervisor: Eder Scheid, Muriel Franco, Matthias Schwalbe
(Aduno Group)

Date of Submission: May 1, 2020

Zusammenfassung

Durch die immer wachsende Datenmenge und dem Bedürfnis nach Realtime oder Near-Realtime Verarbeitung eben jener, stoßen batchorientierte Systeme langsam an ihre Grenzen. Speziell im Gebiet der Kreditkartenbetrugsprävention spielt dies eine extrem wichtige Rolle. So muss die Überprüfung ob ein Einkauf getätigt werden kann innerhalb von einigen wenigen Millisekunden durchgeführt werden. Das Problem, welches diese Arbeit angeht, soll ich das effiziente und schnelle aufbereiten der für das Scoring benötigten Daten. Dieses Paper wird in Zusammenarbeit mit der Aduno Gruppe geschrieben, welche sich unter anderem auf das Herausgeben von Kreditkarten spezialisiert hat und somit starkes Interesse an einer effektiven Bekämpfung von Betrug hat. Das neuentwickelte Machine Learning Model welches momentan benutzt wird bezieht jedoch seine Daten über einen batchorientierten Prozess, welcher nur jede Stunde aktiviert werden kann aufgrund des hohen Ressourcenverbrauchs. Offensichtlich besteht hier ein klarer Konflikt zwischen den benötigten Millisekunden und der aktuell bestehenden Dauer von einer Stunde. Hier soll das vorliegende Projekt Abhilfe schaffen und die Datenaufbereitung für den Fraud Scoring Algorithmus mithilfe von Stream Processing implementieren. Aufgrund der Tatsache, dass jedoch keine produktiven Daten für diese Arbeit verwendet werden dürfen, besteht zudem der Bedarf nach Testdaten, welche eine hohe Korrelation zu den tatsächlichen Daten aufweisen. Diese sollen jedoch zufällig erzeugt werden. Zu diesem Zwecke soll ein System implementiert werden, welches mit Hilfe von Monaden und statistischen Verteilungen sinnvolle Testdaten generiert.

Abstract

Due to the ever growing amount of data and the need for real-time or near-real-time processing, batch-oriented systems are slowly reaching their limits. Especially in the area of credit card fraud prevention this plays an important role. Checks whether a purchase can be made must be carried out within a few milliseconds. The problem which this work addresses is the efficient and fast preparation of the data required for scoring of credit card usages. The paper is written in cooperation with Aduno Group, which is specialized in credit card issuing and thus has a strong interest in fighting fraud effectively. However, the newly developed machine learning model that is currently in use obtains its data via a batch-oriented process that can only be run every hour due to high resource consumption. Obviously, there is a clear conflict between the milliseconds required and the current duration of one hour. Thus, the goal of this work is to mitigate this problem and implement the data preparation for the fraud scoring algorithm with stream processing. Due to the fact that no productive data may be used for this work, there is also a need for test data that has a high correlation to the actual data, but is generated randomly. For this purpose a system is to be implemented which generates meaningful test data with the help of monads and statistical distributions.

Acknowledgments

We would like to thank many people for the opportunity to do this very interesting and educational work. First of all, we would like to thank the Aduno Group, namely the Head Business Analytics Services - Herbert Bucheli, Head Risk Analytics - Dr. Viton Vitanis, Head Datawarehouse Development - Pascal Simon and Business Manager Luca Huber for all their input and support. We would also like to thank Prof. Dr. Stiller - Head of the Communication Systems Group at the University of Zurich and Eder Scheid for the opportunity to write this paper at their chair, as well as for their continuous support and feedback. Last but not least, our greatest thanks goes to Matthias Schwalbe. Without his dedication and educational input for us, this work would simply not have been possible. It is invaluable to be supported by someone with such great knowledge, who you can call on a Sunday morning (while he's sitting in a cafe) and will spend an hour to answer any questions that arise. Many thanks for that.

Contents

Zusammenfassung	i
Abstract	iii
Acknowledgments	v
1 Introduction	1
1.1 Task Description	2
1.2 Report Outline	3
2 Background	5
2.1 Credit Card Fraud Prevention	5
2.2 Red Black Trees	7
2.3 Monads	9
2.4 FreeMonads	13
2.5 Agent-Based Modeling (ABM)	15
2.6 Multi-Agent System (MAS)	16
2.7 Stream Processing	17
2.8 Apache Flink	19
2.8.1 Keyed Streams	20
2.8.2 Windows	20
2.8.3 Triggers and Evictors	22
2.8.4 Event And Processing Time	23

3	Related Work	25
3.1	Multi-Agent System and Agent-Based Modeling	25
3.2	Data Generation	26
3.3	Snappy	27
4	Design	29
4.1	Architecture	29
4.2	Difficulty of Meaningful Test Data	29
4.2.1	Augmented Red Black Tree	31
4.2.2	Data Generation	35
4.2.3	Determinism of Data Generation	38
4.3	Data Aggregation	39
4.3.1	Sliding Windows	40
4.3.2	No Window Assigner	41
4.3.3	Global Window With Tumbling Window Slices	42
4.3.4	Comparison of the Methods	45
4.3.5	Summary of Problems	46
4.4	Data Reclassification	47
5	Implementation	49
5.1	Test Data Generators	49
5.1.1	Augmented Red Black Tree	49
5.1.2	Gen and TE Monad	57
5.1.3	Load Statistical Data Into The Structure	64
5.1.4	Poisson Point Processes	67
5.1.5	How to Enable Determinism	68
5.1.6	Programs	70
5.2	Data Aggregation	73
5.2.1	No Window Assigner	73
5.2.2	Window Slices With No Precise Main Window Length	76
5.2.3	Combination of the Generated Test Data and the Aggregations	82

<i>CONTENTS</i>	ix
6 Evaluation	85
6.1 Test Data Generators	85
6.1.1 Tree Operations	85
6.1.2 Resource Requirements	90
6.2 Generation of Events	92
6.2.1 Determinism	93
6.3 Evaluating Correctness of Window Aggregations	94
6.4 Measuring Performance of Window Aggregations	95
6.4.1 No Window Assigner	95
6.4.2 Window Slices Evicting Whole Slices	97
7 Summary and Conclusions	101
7.1 Future Work	102
Abbreviations	107
Glossary	109
List of Figures	109
List of Listings	112
A Contents of the CD	115

Chapter 1

Introduction

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, which require frequent 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, straightforward features include:

- The time difference between transactions,
- The average amount spent in the past 30 days,
- The ratio between current amount, previous amount, and,
- 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 fraud scoring at [2], to which the authors of this paper have access to, 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 data state. This requires significant investments in time and resources. Moreover, due to the sheer amount of data, maintaining 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 relying 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 as in the case where the scores are calculated every hour and only the subsequent fraudulent transactions are blocked.

1.1 Task Description

This work needs to ensure data preparation for the fraud prevention system using streaming technology. It is important to notice that for the master project, due to legal concerns, no work is performed on live data (*i.e.*, data on production). Instead, the students rely on the generation of test data that models general interactions and statistical correlations of production data. These correlations are automatically generated in a sub-step of the work. The present work is divided into the following three parts.

Part one is concerned with the generation of test data (multiple entities), based on monadic random generators described in [3]. The statistical models needed for these relations in the data will be provided by Aduno [2]. Moreover, besides the data produced based on [3], the students must extend the scope to support:

- Probability distributions
- Time models based on point processes
- Correlation within single entities
- Distribution profiles
- Correlation between many entities

Part two focuses on aggregating the data in sliding and tumbling windows. Taking the systematic of [4] as a basis, which describes a concept based on different classifications of workload profiles, the approach must be extended and adjusted where the available data does not meet the preconditions described in the paper. This work relies on the

partial aggregation of tuples to slices (fixed intervals) and takes up on the principle of “divide and conquer”. The optimization in comparison to existing algorithms lies in the fact that slices at the boundaries of these windows only have to be calculated once as the values can be shared. Therefore, just the pre-aggregated values have to be considered and not all (possible large number) events in a certain range. However, in contrast to this approach, the data required by Aduno [2] for its fraud prevention system has different characteristics, which makes it necessary to adapt the proposed algorithm.

- The data is based on sparse slices (*i.e.*, $|slices| \ll |tuples|$ is violated) and the possibility to aggregate a large number of tuples in a smaller amount of slices is not given
- The key-space/number of parallel windows is in a range of 10^7 to 10^8 (*i.e.*, not suitable for in-memory calculations)
- Integrated state-storage is missing from the presented approach.

Part three covers the reclassification of the data as it is not directly covered by part two. The reclassification, in the case of this master project, is a necessity for the data preparation. The reason for this is the occurrence of an authorisation or transaction initially classified as non-fraud, which is marked as fraud in a subsequent transaction. It must therefore be removed from the previously assigned window and sorted into the correct one. Moreover, it includes the recalculation of window aggregations in the affected windows. Thus, similar to the classifications in part two of this master project, it must cover:

- Distributive and algebraic aggregations, *e.g.*, count, mean, standard deviation, and skewness.
- Holistic aggregations, such as quantiles and unique count (*e.g.*, HyperLogLog [5]). The goal of these operations is to avoid being forced into saving all entities of a certain topic (*e.g.*, authorisations) in order to calculate the unique count. This would otherwise result in an unnecessarily large dataset. Instead, estimates are made with algebraic approximation for the actual count value.

1.2 Report Outline

The remaining of this work is structured as follows. The next Chapter will mainly focus on providing the necessary background information and give a brief introduction to the overall context that this paper is written in. Chapter 3 takes a look at some other papers on this topic that overlap with this work. Chapter 4 focuses on the design decisions taken. Chapter 5 describes the actual implementation and allows for a clear insight into the elaborated system. Finally, Chapter 6 is used to evaluate the solution at hand and in Chapter 7 conclusions are drawn.

Chapter 2

Background

This Chapter is intended to give the reader a basic understanding of the technologies and concepts used. In Section 2.1 a general introduction to the topic of credit card fraud prevention is given and the urgency of timely processes is discussed. Section 2.2 focuses on Red Black Trees, whereas Section 2.3 explains monads, which play an important role in the later course of this work. Subsequently, a look is taken at Agent Based Modeling (ABM) and Multi-Agent Systems (MAS), which are related to the first part of the project. Next, stream processing and the Flink framework used for it are introduced. Additionally, an insight is given into the Flink operations required in the second and third parts, i.e. keyed streams, windows, triggers and evictors. Finally, the differences between event and processing time are explained.

2.1 Credit Card Fraud Prevention

This academic work was written in cooperation with a credit card company and the goal was to develop a data preparation system for a fraud prevention algorithm. Inevitably, the question arose how fraud prevention was handled until now and what impact the new solution would have on the current situation. This question is to be answered in the following lines. As a basic introduction to the topic, it is certainly of interest to know what is meant by credit card fraud. One speaks of fraud when, for example, a customer initiates a payment at a terminal in Switzerland (card physically present) and only three hours later, another authorisation is initiated in America with the same card. Further examples would be the speed of triggered authorisations at a particular merchant or a pattern where test transactions are executed to see if a card is active. If it is indeed active, larger and larger transactions are made. This is where fraud prevention comes into play, as the goal of it is to detect and prevent such patterns. Yet, it is important to note that the issuer of the card does not always have to bear the cost of fraud. The issuer is responsible for card present and card not present usages (with activated 3DSecure protocol). However, if, for example, a customer violates the duty of care by storing the PIN and card in the same wallet, then the customer is liable if the wallet gets stolen and the card is used. Small merchants usually require 3DSecure from the issuer. Still, larger

providers (*e.g.*, Amazon) refuse to do so and often have their own fraud departments. One of the reasons why this is done is mentioned in [6]. It is estimated that 3D Secure can result in a 19% loss of revenue as customers do not know their password, are getting distracted by other things like a ringing doorbell and thus forget to buy. Other reasons that are mentioned are a clunky interface that makes customers feel less secure or just an overall bad user experience and frustration due to the extra authentication. As the merchants are interested in making a deal and get profit, they sometimes take the higher risk if the expected loss is higher than the potential damage.

As by now, the company this paper is written in cooperation with, relies on a widely implemented and tested rule engine of a third party provider. Such rule engines allow a user to set up certain policies that determine whether a card usage is authorized or rejected. For example, if a merchant category code (mcc) refers to a restaurant and a customer pays for something between two dates, the authorisation is most likely confirmed. However, if the same card is used in a restaurant to make three transactions of 50 or more in less than 10 minutes, the authorisation will be rejected. Since each and every authorisation request must be answered by the system within a maximum of 150 ms, performance is an important issue in this environment. The disadvantage of such a system is that it uses static rules and requires a large amount of personnel to maintain and adapt them if necessary. Of course, the department that takes care of this set of rules is also supported by a large number of reports from the company's internal data scientists. Nevertheless, a lot of manual work remains.

Another problem is that not all required attributes can be calculated in the basic system due to performance considerations. Furthermore, the system only has access to the data that the processor has available. The processor in this case is the provider hosting the fraud prevention system. Joining with other data, such as the company's own credit card app, is very difficult and time-consuming. However, it is precisely this additional information that would be interesting for an intelligent model. For example, the date and destination of a trip could be filtered out in the addendum records for flight bookings. If a transaction is then carried out in the destination country, there would, under certain circumstances, be no need to trigger a warning as it is known that the customer is really there.

To compensate for the disadvantages of this static but robust system, an in-house fraud scoring system was implemented which uses machine learning algorithms to assign a certain score to an authorisation. This score is then used to indicate the probability of it being fraud or not. In an internal study it was found that twice as much fraud could have been prevented as with the existing system (which is already able to prevent a considerable amount of fraud). However, the real problem is that the data preparation for training takes a long time. At the moment, this amounts to about four days for one month of data. In addition, the effective scoring can only be carried out every hour, as a large amount of data has to be touched during each run. That said, the current system relies on a scoring that should carry out its work in milliseconds. Yet, due to external circumstances, it can only be carried out every hour. Streaming could certainly be helpful, as there is no longer any need to perform batch calculations which can take hours. An additional possibility to use the streaming functionality in the area of fraud prevention refers to graph analysis, which can be used to determine the proximity of merchants to each other. Currently, the

merchants “googl.ch”, “google.ch” and “ggl.ch” are considered as individual merchants and the aggregations are set up in this way. However, the results of such aggregations would change dramatically if it was known that these different merchants are in fact the same. Such patterns are difficult for humans to recognize but with graph analysis it can get obvious. The need for an efficient and dynamic system for fraud prevention is obvious, but difficult to realise by traditional means. This is where this master project comes in to enable data preparation for such a machine learning model, which is supposed to operate in a millisecond range. This said, the paper will go on to explain the needed parts in order to successfully implement the individual tasks of this project. Starting off with the data structure that will play an important role in enabling the efficient generation of test data.

2.2 Red Black Trees

Data structures, *e.g.*, Red Black Trees [7], play a very important role in applications which have to handle large amounts of data. This also applies to the test data generation in the project at hand, where it is potentially needed to cover the quantity of all payment terminals around the globe in a data structure. In this section, the general functionality of such a tree and the differences to normal binary trees is explained. The specific implementation of the augmented Red Black Tree used in this project will be discussed in Section 5.1.1.

In principle, Red Black Trees can be described as self-balancing binary search trees with the following characteristics:

- Each node is either red or black,
- The root node is always black, and
- There are never two red nodes in a row

Still, the question that has to be answered is what the advantage of such a tree over a normal Binary Search Tree (BST) is. BSTs need $O(h)$ time for most operations, where h is the height of the tree. The problem is that in the worst case the tree can be skewed as seen in Figure 2.1. This means that the elements are inserted in a strictly growing order and thus, the tree only consists of a continuous path to the right. This finally leads to the tree performing operations like search and delete in $O(n)$ time instead of $O(h)$. To counteract this, the red and black tree balances itself whenever a node is added or removed. This ensures that the height of a tree with n nodes is smaller or equal to $2\log_2(n + 1)$. Thus, operations on the tree can be performed in $O(\log(n))$. Figure 2.2 shows the same sequence of values as in Figure 2.1, but with the rules of the red and black tree applied.

For an insert into the tree, the following steps must be applied. First, a normal BST insert is performed (`if(value > key) add right else add left`) and the new node is marked as red. If the new node is the root, it is marked black and the algorithm is

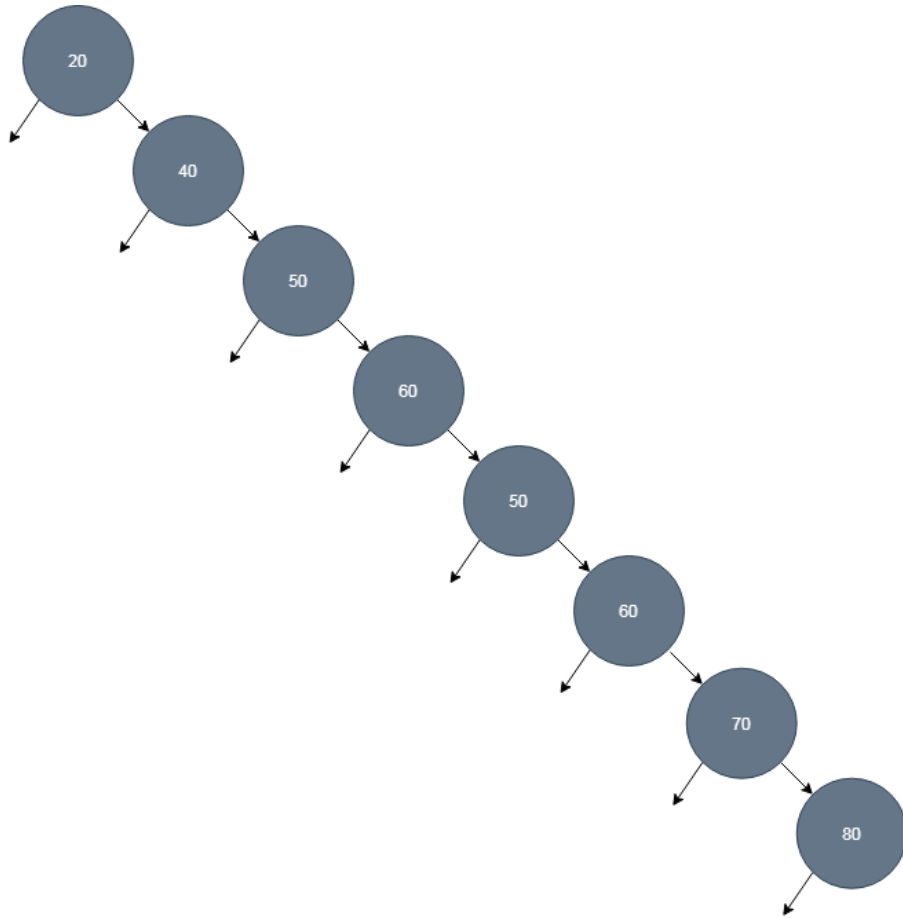


Figure 2.1: Skewed Binary Search Tree

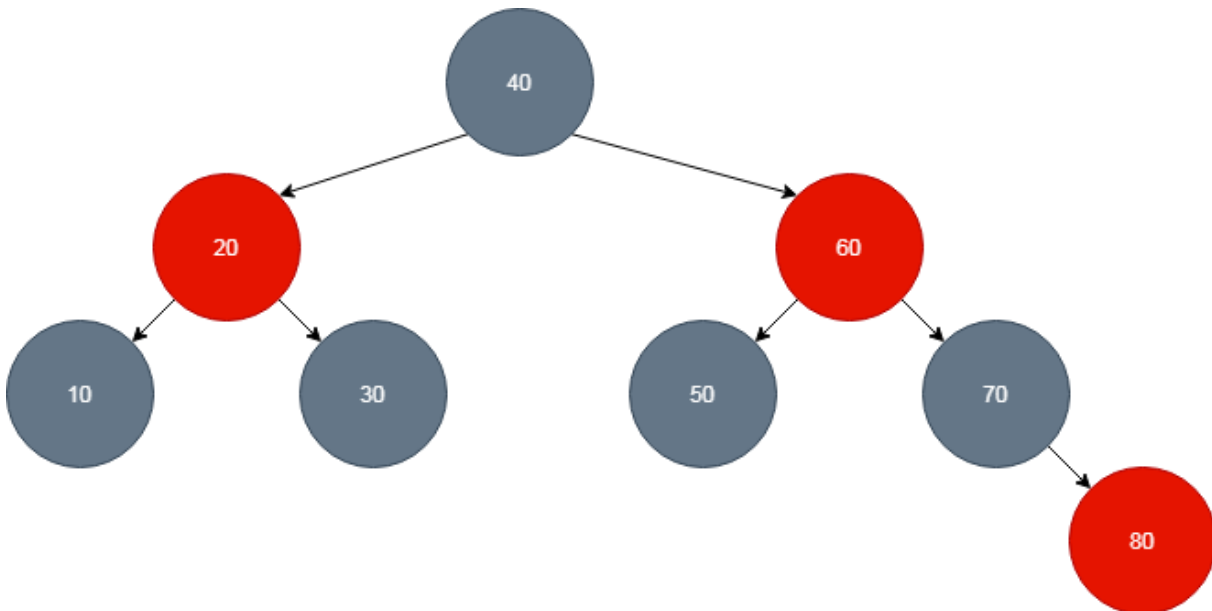


Figure 2.2: Same Values as in 2.1 but With Red Black Tree Algorithm

terminated. Otherwise, there is a case distinction depending on whether the uncle node of the newly inserted node is black or red. If it is red, the parent and uncle nodes are colored black and the grandfather node is marked as red. Then the algorithm is iteratively propagated upwards. If the uncle node is black, four rotation cases are distinguished, depending on the colors of the other nodes and the position (left/right) of the parent node. By following these steps and rotations of nodes, it is ensured that the same number of black nodes are found on the path from the root to the individual leafs. Therefore, a complexity of $O(\log(n))$ is achieved. Now that the basics of the data structure that will be used in the later course of this work are explained, the next essential topic can be introduced. In the following Section monads will be introduced, which facilitate composability of functions in a convenient way.

2.3 Monads

Let assume a mathematical value. An integer with value 3 shall serve as an example. This is an expression of a pure value, to which mathematical rules can be applied to. But if this value is to be brought into the real world, on a piece of paper or in the form of pixels on a screen, effects are needed that have an impact on the real world. This is where monads come into play. Monads are derived from the category calculus (subarea of algebra) and allow to isolate effects, combine them and put them in order. This should allow to bring the mathematical side and the effects side together to prove the correctness of a program, despite side effects. The monad laws, which will be discussed later in this section, are used for this. If a developer follows these laws, it can be proven that a program is still considered functional despite side effects.

The need for monads can be explained by a quotation from John Gall in 1975 [8], which states that "A complex system that works is invariably found to have evolved from a simple system that works. The inverse proposition also appears to be true: A complex system designed from scratch never works and cannot be made to work". But what exactly does this quote have to do with something as abstract as monads?

Simply put, monads are algebraic structures that facilitate the elegant composability of calculations. They form an essential part of the work at hand. In the following lines an introduction to this topic shall be given without going too much into detail. Monads were already introduced in 1958 by Godement [9], which implies that the concept is not something brand new. Yet, it is still a topic that is difficult to understand and even harder to explain. Or as stated in [10], "A monad is just a monoid in the category of endofunctors. What is the problem?". The problem seems to be that monads are based on a precise category theoretical definition, yet this definition does not help to understand monads in an easy way. It does also not provide the answer why they are so useful in programming either. Nevertheless, the following lines try to shed some light on the many benefits monads provide and explain them with simple examples.

First, however, two terms need to be explained in more detail. These are the functions `map()` and `flatMap()`, which play a crucial role in the later course. At this point, the

functions will be explained from general point of view in order to create a basic understanding of its overall purpose. Map on the one hand is needed to apply a function to all elements of a container. As an example, a list is used, to which a map operation could be applied to that multiplies each value by 2 ($x * 2$). The result would look like shown in Listing 2.1 and contain every value of the list with the provided function applied to it.

```

1  val a : List[String] = List(1,2,3,4)
2  val b = a.map(x => x*2)
3
4  val b => List(2,4,6,8)

```

Listing 2.1: Map Example

FlatMap on the other hand works by combining a map operation with a flattening. This means that a map is executed and returned on each element of a sequence. At the end these are "flattened" back into the original sequence.

As an example, a list of numbers and a function that returns another list instead of a value is considered. If one would only apply map to it, a sequence of sequences of numbers would be returned. By using flatten, this sequence of a sequence is converted into a simple sequence. Or simply put, the hierarchy level is flattened by one level with each application as shown in Listing 2.2.

```

1  def f (x : Int) = List(x,x+1)
2  val a : List[String] = List(1,2,3,4)
3
4  val onlyMapResult = a.map(x => f(x))
5  val flatMapResult = a.flatMap(x => f(x))
6
7  val mapResult => List(List(1,2),List(2,3),List(3,4),List(4,5))
8  val flatMapResult => List(1,2,2,3,3,4,4,5)

```

Listing 2.2: Difference Map and FlatMap

Now that the two terms map and **flatMap** have been introduced, the concept of monads can be explained. In [11], a simple definition derived from [12] is given which states that a monad is a triple consisting of a type constructor **M** and the operations identity and bind. Since this project is to be implemented with Scala, the terms **flatMap** for bind and pure for identity are used which represent the respective equivalent. The two terms are not self-explanatory in the context of monads and thus, a definition is given on the following lines.

Definition $\text{pure}[A](a: A): M[A]$

This definition states that an entity of type **A** is wrapped into an **M**, representing a monadic structure. For example, if **A** is assumed to be an **Int** and **M** is a **List**, then

`pure(Int) => List[Int]` or simply put, `a -> List(a)`. Thus, `pure` is defined by Cats [13] as an operation that wraps the value into the type constructor. In object-oriented languages, this concept is called generics.

Definition `flatMap[A, B](f: (A) => M[B])(fa: M[A]): M[B]`

This definition states that given a monad of type `A`, the result of the effect of `M[A]`, *i.e.*, `a's: A` applied to `f:(A) => M[B]` yield an `M[B]` which represents the effect of the `M[B]` returned by `f` when applied to the `a's`. In other words, a container of type `M[A]` is taken that performs a certain operation to produce a `M[B]`. Or as described by [13], `flatten` can be used to define `flatMap` by stating that `flatMap` is just a `map` followed by a `flatten` operation. The reverse statement also works, which claims that `flatten` is just a `flatMap` operation using the identity function.

What is still missing to define a monad is the `map` function. Since `flatMap` can be defined with the help of `map` and `flatten`, it will be provided here as well.

Definition `map[A, B](f: A => B)(fa: M[A]): M[B]`

Thus, `map` is used when a function `A => B` is present, as opposed to `A => M[B]`, and an operation shall be applied to every `A` in `M[A]`.

In [11], a metaphor for monads is offered which seems to be easily understandable. The article describes a monad as a type of container that contains values of type `A`. This is according to the source the intuitive understanding from a mathematical side which states that a monad embeds an object in an object with a richer structure. As an example, the `List` monad is mentioned. It can contain various types (*i.e.*, `String`, `Integer`) and allows to execute the essential functions `pure`, `map` and `flatMap` which were introduced before. Another important aspect of a monad is, that it allows reasoning about programs based on mathematical rules. Only the combination of the previously mentioned functions and these laws make a monad to what it is. These laws are, among others that derived from them, the following.

- **Left Identity** Assume a value `x`. If `x` is lifted into a monad with the `pure` operation and afterwards used in a subsequent monadic function, it is the same as if only `x` is taken and the given function is applied to it. Thus, condition `withMonad == onlyMethod` is satisfied (*cf.* Listing 2.3). The formal definition of what has just been written is given below, together with an example to further clarify the points mentioned.

```
leftId[A, B](a: A, f: A => F[B]) = F.pure(a).flatMap(f) <-> f(a)
```

```
1 def dupFunction(i : Int) : List[Int] = List(i*2)
2 val a = 1,2,3
3 val withMonad = List(a).flatMap(dupFunction)
4 val onlyMethod = dupFunction(a)
5
6 val withMonad => List(2, 4, 6)
7 val onlyMethod => List(2, 4, 6)
```

Listing 2.3: Left Identity Example

- **Right Identity** If there is a monadic value and the `flatMap` operator is used, the value returned will be the original monadic value. Thus, condition `flatMapOnMonad == plainValue` is satisfied (*cf.* Listing 2.4).

```
rightId[A](fa: F[A]) = fa.flatMap(F.pure) <-> fa
```

```
1 val b = List(1)
2 val flatMapOnMonad = b.flatMap(List(_))
3 val plainValue = b
4
5 val flatMapOnMonad => List(1)
6 val plainValue => List(1)
```

Listing 2.4: Right Identity Example

- **Associativity** If there is a chain of monadic functions it should not matter how they are nested. Thus, condition `seq == nested` is satisfied (*cf.* Listing 2.5). Again, the formal definition is given below, together with an example.

```
associativity[A, B, C](fa: F[A], f: A => F[B], g: B => F[C]) =
  fa.flatMap(f).flatMap(g) <-> fa.flatMap(a => f(a).flatMap(g))
```

```
1 def dupFunction(i : Int) : List[Int] = List(i*2)
2 def subFunction(i : Int) : List[Int] = List(i-1)
3
4 val c = List(1,2,3,4)
5 val seq = c.flatMap(dupFunction).flatMap(subFunction)
6 val nested = c.flatMap(x =>
7     dupFunction(x).flatMap(subFunction)
8 )
9
10 val seq => List(1,3,5,7)
11 val nested => List(1,3,5,7)
```

Listing 2.5: Associativity Example

Nevertheless, the questions remains what exactly the advantages of monads are, that should pay off for the work at hand. Often a developer is busy with complex tasks consisting of many steps, which can (and most of the time will) end up in a complicated structure. Monads allow the division of complex problems into manageable steps and put them together again afterwards. Therefore, one of the main advantages of monads is the composability by separating effects from pure functions. Objects can be put into containers, the required functions can be predefined and then run together like in a production chain without having to worry about composability. Simply put, it is just input in one end, a complication and calculation in between and the output in the other. This already exciting and powerful instrument can be made even more interesting by the FreeMonads pattern presented in the next section. They allow to specify models based on monadic structures that can be interpreted and executed at a later time.

2.4 FreeMonads

FreeMonads describe a construct which allows to build a monad from any functor. A functor is defined by [13] as a type class which can be used to abstract and map type constructors. Type constructors are defined as the process of constructing a new type from another (*i.e.*, $F[A]$ is constructed from A). Examples for this are `List` or `Array` where $\text{Int} \Rightarrow \text{List}[\text{Int}]$. As it was stated in the Section above, they would be called generics in object-oriented languages. It is important to note that by mapping over such a structure, the type can change, but the underlying structure remains the same. `Map` was already introduced in the previous section, but at this point an example will be given to clarify what is meant by the changing type but same structure. Consider an array with the values `val initialArray : Array[Int] = Array(1,2,3,4)`. The function `initialArray.map(x => s"Number: $x")` is then applied to the array. As result, an array (same structure) is returned but with the different type `String`. When printing this structure to the console the result as seen in Listing 2.6 is shown.

```
1 val initialArray : Array[Int] = Array(1,2,3,4)
2 val res = initialArray.map(x=>s"Number: $x")
3 println(res.foreach(x=>println(x)))
4
5 Number:1
6 Number:2
7 Number:3
8 Number:4
```

Listing 2.6: Output of a Map Operation

The FreeMonad structure allows to represent and manipulate computations in a pure way, which will be explained in the next lines. A good example can be found in [13] to further clarify this pattern. The goal is to implement a key-value store with the ability to get and put values into a storage, while isolating parts that are not purely functional (effects). The idea is to model the non-pure operations in an embedded Domain-Specific Language

(DSL) as a program and compile it afterwards. Besides the requirement to stay pure functional, the creation and execution of the program should also be separated. This will allow an easy exchange of the individual functions for execution.

A first step to achieve the goals mentioned above, is to create an ADT for the required effects. The ADT is represented by the case classes `Put[T]` and `Get[T]` of the sealed trait called `KVStore[A]`. The `Put` case class takes a key and a value (of type `T`) as an input, whereas `Get` only takes a key as shown in Listing 2.7.

```

1 sealed trait KVStoreA[A]
2 case class Put[T](key: String, value: T) extends KVStoreA[Unit]
3 case class Get[T](key: String) extends KVStoreA[Option[T]]

```

Listing 2.7: Definition of the ADT

As a next step, Listing 2.8 shows the creation of a type `KVStore[A]`, based on `Free[_]` and `KVStoreA[_]`.

```

1 type KVStore[A] = Free[KVStoreA, A]

```

Listing 2.8: Definition of the ADT Type

Afterwards, in Listing 2.9, smart constructors are implemented with the aid of the `liftF` function, that will lift `KVStoreA[_]` values into the `KVStore[_]` monad.

```

1 // Put returns nothing (i.e. unit).
2 def put[T](key: String, value: T): KVStore[Unit] =
3   liftF[KVStoreA, Unit](Put[T](key, value))
4
5 // Get returns an Option of a T value (None or Some(T)).
6 def get[T](key: String): KVStore[Option[T]] =
7   liftF[KVStoreA, Option[T]](Get[T](key))

```

Listing 2.9: Lifting of Functions Into the Monad

As a result, the afore created DSL can be used to construct programs using for-comprehensions as depicted in Listing 2.10. In this listing, the string "fraud-cases" shall be stored together with the value 2 and non-fraud-cases with 10. Finally, the number of fraud-cases will be returned.

```

1 def program: KVStore[Option[Int]] =
2   for {
3     - <- put("fraud-cases", 2)
4     - <- put("non-fraud-cases", 10)
5     n <- get[Int]("fraud-cases")
6   } yield n

```

Listing 2.10: KVStore Programm

This looks like a monadic flow, yet it is only a recursive data structure which represents a sequence of operations and does not produce anything yet. To make this work as intended, a compiler is needed which translates the new abstract language into an effective language and executes it. This last missing piece of the puzzle is to be found in the aforementioned compiler which contains the lifted functions represented by the corresponding case classes. Depending on which case class the pattern matching applies to, other functions are executed as shown in Listing 2.11. For instance, when the ADT case `Put` is being given to the compiler, it prints the key and value that will be written and puts a value into a map for the given key.

```

1 def apply[A](fa: KVStoreA[A]): Id[A] =
2   fa match {
3     case Put(key, value) =>
4       println(s "put($key, $value)")
5       kvs(key) = value
6       ()
7     case Get(key) =>
8       println(s "get($key)")
9       kvs.get(key).map(_._2.asInstanceOf[A])
10  }
11 }

```

Listing 2.11: FreeMonads Compiler

By using FreeMonads, it is possible to model complex processes in simple programs using a domain specific language. After introducing the two concepts of efficiently storing data sets in trees and assembling functions with monads, the next sections takes a look at how the two concepts can eventually be combined.

2.5 Agent-Based Modeling (ABM)

Since the first part of this thesis requires the creation of correlated test data and thus, the modelling of customer behaviour in the context of credit card usages, the question of a suitable approach arises. During the course of the research, one inevitably encounters the terms Agent-Based Modeling (ABM) and Multi-Agent System (MAS). The former is

described in the following lines, while the latter is discussed in more detail in the next section.

The ABM is described by [14] as a model consisting of many small units which have decision and action possibilities between the individual agents, the agents and the environment as well as various features of the environment. This allows a modeling between the individual actors on the micro and macro level. The agents themselves can be defined on different levels, including individuals or a collection of individuals and are characterized by various attributes and rules of behaviour that reflect the heterogeneity of a population. This allows an ABM to observe the effects of an intervention or policy by conducting what-if scenarios. The model can be made more precise by using large data sets. In addition, according to the rules of Big Data, the recognition of patterns, relations and rules can be facilitated. As a possible area of application of ABMs, [14] describes a study that models neighborhoods to explore the effects of the environment on the health behavior of individual actors, for example, physical activity related to obesity and diabetes.

2.6 Multi-Agent System (MAS)

As described by [15], multi-agent systems were an emerging area of distributed artificial intelligence in the 1970s and 1980s. The paper describes a system that is formed from multiple agents that can collectively solve a problem that would be difficult or impossible to solve for monolithic systems. Such MAS were found to be used in a wide range of applications. For example, they serve as heuristic solution methods for complex processes in ant colonies. What has never been clearly communicated is what an agent actually is. Agents are generally represented in theory as entities that:

- Are able to operate in an environment
- Are able to communicate with other agents
- Are driven by a number of personal tendencies and preferences to increase one's own benefit.
- Have own resources
- Are able to perceive its surroundings (but to a limited extent),
- But have only a partial representation of this environment (and maybe none at all)
- Have skills and can offer services
- Whose behavior is determined by the above-mentioned points in order to fulfill its own goals.

Furthermore, MAS are described as systems consisting of the following elements

- An environment E which consists of a space with a certain volume.
- A set of passive objects O which can be associated with a position in E at any time and can be perceived, created, modified or destroyed by the agents
- A set of agents
- A set of relations that link agents and objects together
- A set of operations that allows the agents to interact with the objects.

That said, the difference between a MAS and ABM is the way in which the solutions are viewed. A MAS focuses on solving a specific and complex problem with the help of independent agents and thus, implements a top-down approach. ABM instead takes a bottom-up approach and models the behavior of individual agents and their interactions with each other. This way, a complex system is formed without specifying behavior at the system level. As a next step, the following section will give an introduction to stream processing and the Apache Flink framework used for the project.

2.7 Stream Processing

As mentioned in Section 2.1, the aim of the project is to use stream processing to efficiently process and analyze the data from the credit cards. But what exactly is stream processing? Principally it can be described as the processing of data in motion or simply the processing of streams. Instead of storing incoming data in a database first, stream processing computes on data directly when it is received, optimally directly when it is produced. Compared to applications which use a database, the application logic is not applied on request but continuously. The application reacts to each received event and may trigger an action like returning a new value or updating the current aggregated value. A stream processing application has to face different challenges to be able to deliver the expected results in near real-time. [16] lists 4 key characteristics, which should be fulfilled by a stream processing application.

- Instant reaction when a new event happens. Optimally there should be no time gap between when the event happens and when the action is taken by the application. Otherwise, the state is reflecting a data set, which already passed its validity.
- Stream processing is able to handle much larger data volumes than other data processing systems, as in most cases not all events have to be stored, but only a meaningful subset.
- Stream processing models the continuous and timely nature of most data. When possible stream processing computes its result in an incremental way when a new event happens instead of a periodic re-computation of all the events. (Batch Processing vs Stream Processing)

- Stream processing enables a decentralization and decoupling of the infrastructure as it reduces the need for large and expensive shared databases. Instead, each application maintains its own data and state, which makes it naturally fit in a micro-service architecture.

The 4 key characteristics above pose a solid basis for the understanding of the requirements a stream processing application should fulfill. However, they are optimistic and the first two characteristics are often not achievable due to high complexity. The instant reaction to new events as stated is a general problem of stream processing, which will be explained in Section 2.8.4. Furthermore, the second characteristic is hard to hold for use cases where an incremental methodology is not available. This problem will be discussed in more detail in Section 4.3. Besides these limitations, still a lot of components are missing to really understand stream processing and how it is applied. By just considering the 4 characteristics above there is one unbounded stream, which is continuously processed and will never end. An example of such a stream is shown in Figure 2.3.



Figure 2.3: One Input Stream [17]

By looking at the use case of fraud detection, it gets obvious that the stream processing application should not just continuously aggregate all newly arriving authorisations. It should distinct the events between the card numbers, merchants or other features of the data (*i.e.*, having one stream per card would improve the possibilities to analyze the events of a single customer). This requirement can be met with the concept of keyed streams, which make it possible to split a stream according to a specified key. This is explained in more detail in Section 2.8.1. Furthermore, events which happened a long time ago are in most cases not relevant anymore and should therefore not be considered. Thus, the question arises how the application will decide which events are relevant for the next computation and which are not.

Another important point regarding optimization is to ensure that calculations are not applied to every stored event, but just for the ones that are relevant for a query and happened in a certain period of time. In stream processing, this problem can be solved with the usage of windows, which will be explained in Section 2.8.2. In general, a stream as in Figure 2.3 can be split vertically using keyed streams and horizontally using windows.

In order to be able to efficiently handle all these requirements, a framework is needed that manages all these concerns. For this, stream processors can be used. The way such a system works is described in Section 2.8, where Apache Flink - the framework used in this project - is introduced.

2.8 Apache Flink

Apache Flink was chosen as stream processor for the project at hand, because it is a powerful, mature and open source stream processing framework that fulfills the requirements mentioned in Section 2.7. The functionalities it offers to process a stream will be explained in this Section. Listing 2.12 presents the general structure of a windowed Flink program. It is an adaption of the structure shown by [18].

```

1 stream
2     //only if pocessed in event time
3     .assignTimestampsAndWatermarks(...)
4     [.keyBy(...)]           <- optional: else no keyed streams
5     [.window(...)]         <- optional: else no windows
6     [.trigger(...)]        <- optional: else default trigger
7     [.evictor(...)]        <- optional: else no evictor
8     [.allowedLateness(...)] <- optional: else zero
9     .reduce/aggregate/fold/apply/process()
```

Listing 2.12: General Structure of a Flink Program

The input to such a Flink program can be any stream, which contains a set of events of the same type. Each event needs to have a set of attributes and a timestamp, which represents its event time. Whether a stream has to be processed based on event or processing time is defined in the characteristics of the input stream. The difference between event and processing time will be further clarified in Section 2.8.4. In order to be able to process a stream in event time, a watermark assigner is used. Watermarking is used as a mechanism to signal progress in event time. It tells the system up to which point it can be assumed that all data has been received.

After that, the input stream is optionally split into a keyed stream by the *keyBy(...)* operator in line 4. The attribute given as parameter is the key to split the stream. As a next step, a window assigner can be defined by applying the function *.window(...)*. For window assigner that consider a certain time interval, the function *.timeWindow(...)* is used. As a parameter, any window assigner type can be picked. Keyed streams will be explained in Section 2.8.1 and windows in Section 2.8.2. In line 6 and 7 of Listing 2.12, a trigger and an evictor can be assigned to the stream. Per default, a trigger that fires at the end of a window is chosen and no evictor is used. Triggers and evictors will be explained in more detail in Section 2.8.3. Line 8 defines the allowed lateness, which describes how long a window should stay alive after its lifetime was passed. This parameter is also optional and in case that it is not defined, the allowed lateness is set to 0 and every event that would fall into this window is dropped as soon as the windows end time is passed. As a final step, the function that is called when a trigger fires is assigned in line 9. This function is user defined and has to consider that it takes a list of elements of the input stream as input. Examples for such a function are shown in Section 5.2. Next, this Chapter is dedicated to the individual functions of Flink which play an existential role in the further course of this work.

2.8.1 Keyed Streams

In order to be able to analyze the events which are connected to a specific attribute, keyed streams can be used [18]. Each keyed stream handles all events of a specific key value. For example, if an input stream contains events of three customers and it is keyed by this attribute, there will be three new keyed output streams like the ones shown in Figure 2.4. It shows the events of the input stream, where the x-axis describes the time and the y-axis the customer, which was involved in the event. Each line is a separate keyed stream.

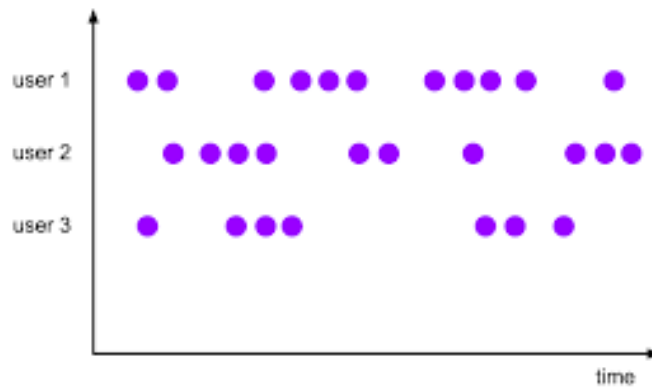


Figure 2.4: Keyed Stream by the User Id as shown by [18]

It is possible to use any attribute of an event as key and even combine them. Keyed streams also allow to be processed in parallel by multiple tasks, since each keyed stream is independent of another keyed stream. All elements referring to the same key still have to be processed by the same task. This additional advantage of keyed streams can improve the performance significantly. Of course it is also possible to have streams which are not keyed, but since it does not make much sense to process all elements together in the present use case, only keyed streams will be considered.

2.8.2 Windows

Besides the splitting of the input stream on the y-axis as shown in Figure 2.4, stream processing also allows to split an unbounded stream in the x-axis, considering the time, into an unbounded set of windows [18]. Windows are one of the most important tools of stream processing and they are at the heart of processing infinite streams. They define when and what element has to be processed. Compared to an infinite stream, a window is a bounded stream with a specific start and end time. Computations are applied only to the elements which are stored in an according window and by default, they are applied when the time passes the end of the window. Another possibility to trigger the computations is by a customized trigger, which will be explained in Section 2.8.3. A new window is created when the first event of a certain time interval arrives. Subsequently, it will be removed when the time (event or processing time) passes its end time plus a certain

lateness, defined by the developer. As defined earlier, the allowed lateness defines how long a window and its values should be kept after its end time passed. It can also be used to facilitate *late events*, which will be defined in Section 2.8.4.

Each element in the stream is assigned to a set of windows by a so called window assigner [18]. How many windows are created and how much time they cover is dependent of the window assigner type and the user specific parameters. Flink offers 4 pre-defined window assigners for the most common use cases. Namely tumbling windows, sliding windows, session windows and global windows. These window assigners, except the session windows, will be used in this project. Thus, they will be introduced in the following paragraphs.

The global window type is simple to understand, because all elements with the same key are assigned to the same single global window. Actually, there is no added distinction compared to the keyed stream shown in Figure 2.4, where no windowing was used. Since the end timestamp of the global window will never be reached (no natural end time) computations will never be applied by default. Thus, it only makes sense to use a global window if a custom trigger is defined.

Tumbling windows are windows of fixed size, which are adjacent to each other but do not overlap. Each element belongs to exactly one window. A tumbling window with size of one hour will evaluate the current window and create a new window at every full hour. Figure 2.5 shows an example of a stream divided into tumbling windows. Compared to Figure 2.4, also the x-axis is split into windows. A possible usage for keyed tumbling windows could be to get the count of events per customer per day.

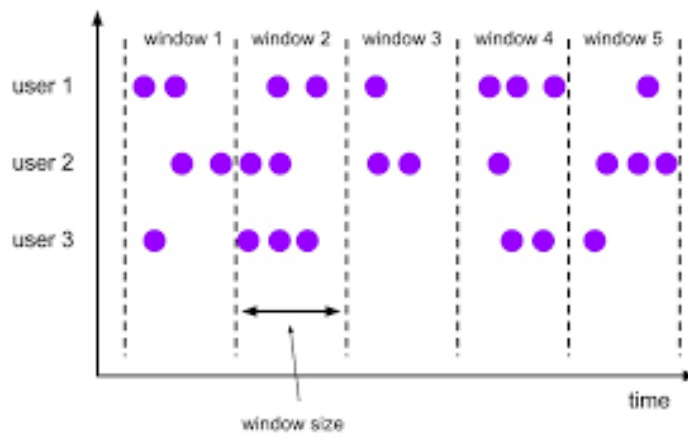


Figure 2.5: Tumbling Windows shown by [18]

Sliding windows have a fixed size and a new sliding window is created at a predefined frequency called window slide. In sliding windows, elements are assigned to $windowSize/windowSlide$ windows in average. If the window slide parameter is equal to the window size, the windows are created equally as in the tumbling window case and each element belongs to exactly one window. If the slide parameter is smaller than the window size the sliding windows are overlapping and the elements are assigned to one or more windows. If it is bigger, there will be elements which do not belong to any window.

Figure 2.6 shows an example, where the window slide parameter is smaller than the window size, so there are overlapping windows. Since the slide parameter is half as big as the window size, each element belongs to exactly 2 windows. The case where the slide parameter is smaller than the window size is the typical one and it is used the most. A possible usage for sliding windows could be to get the count of events of the last 24 hours per customer per hour.

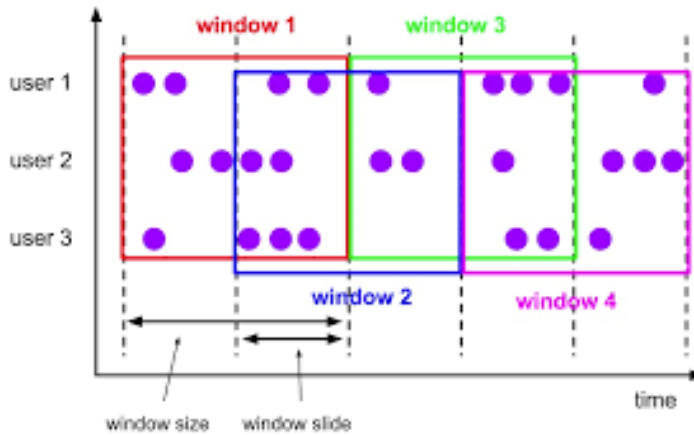


Figure 2.6: Sliding Windows [18]

In all of the explained window types, each window can have its own window state, where window specific variables can be stored in [19]. The window state can be read and adapted during processing only by the specific window. Beside the window specific state there can also be a global state, which can be read and adapted by each window.

2.8.3 Triggers and Evictors

By default, computations are applied to a window when the end time of it is reached [18]. However, it is possible that a specific application needs to apply its computations more than once while the window is active. In case a window should execute its computation at an arbitrary point in time, a trigger can be defined. A trigger itself can be activated in different cases, for instance when a new event arrives or when a specific time is reached. If such a case applies, the trigger decides which action will be taken. For example, the default trigger always fires its computations when the time reaches the end of a window.

Another possible use case is that some elements in a window are not valid anymore while the window is still active. In such cases, an *evictor* can help out. It is able to remove elements before or after computations are applied by checking for different conditions. It can for example remove all elements in a window, which happened between 2 am and 3 am. By using customized triggers and *evictors* new options are available to adapt a window and its element.

2.8.4 Event And Processing Time

As a last step of the introduction to Flink, a look on the differences between event and processing time will be taken. The understanding of the time dimension is one of the most important things in stream processing and crucial to be able to apply the concepts to specific use cases. The event time can be defined as the time, at which an event was generated by the source [20]. Speaking in a credit card context, this is for example the time when a customer pays at a credit card terminal. The processing time on the other hand represents the current system time of the machine processing the events. The processing time is moving forward in real time. As previously stated in the characteristics of a perfect stream processing application, there should be no gap between when the event happens and when an action is taken by the application. In a perfect world, an event would be processed by the application directly when it happens. As already mentioned before, such a scenario is currently not feasible, especially for applications where the data is collected from different places. First of all, the events have to be transmitted to the processing machine, which will lead to a delay depending on the location of the event. Furthermore, it is possible that for instance a terminal has bad or no internet connection at some time and can not transfer its data immediately. This leads to even bigger delays, which can vary in their duration. Considering these stumbling blocks, it can be stated that the processing time is always bigger than the event time. Since the event delays can vary a lot, it is also likely that the ordering of the incoming events does not conform with the ordering when the events actually happened. Since it is important to process elements in correct order, the lack of a strict ordering is an additional challenge. Incoming events can be processed either by focusing on processing or event time. Considering the default trigger, which fires when the end time of a window is reached, the two variants can be analyzed. The processing time variant is simple, the application just returns its results after the window's end time is passed in real time. The event time variant is more complicated. The progress of time depends on the data and not on any wall clock. The event time can progress independently of processing time and it is not known in advance when the end time of a window is passed. By looking at the incoming data it has to be defined which event time is currently valid. In stream processing this problem is handled with so called watermarks, which measure the progress in event time. Watermarks flow as part of the data stream and carry a timestamp t , which defines that time t in the current stream has been reached. This means that there should be no more elements from the current stream with a timestamp $t' \leq t$ [20]. Hence, there should be no incoming events with a timestamp older than the watermark timestamp. The watermark timestamp t is increased according to a specified function while the stream is continuously processed. How this watermark function should work is dependent on the incoming stream and on the goal of the application. Assuming that the events arrive in order, the watermark timestamp can be equal to the time of the last processed element. This scenario is shown in Figure 2.7, where the watermark just moves with the event timestamp of the last element.

However, the arrival of the events will most likely not be in order. Hence, the time of the event that arrived last can not be picked as the watermark timestamp. This is due to the fact that later on an event may be delivered with an earlier timestamp, but the trigger already started the computations. How the watermark timestamp should be set

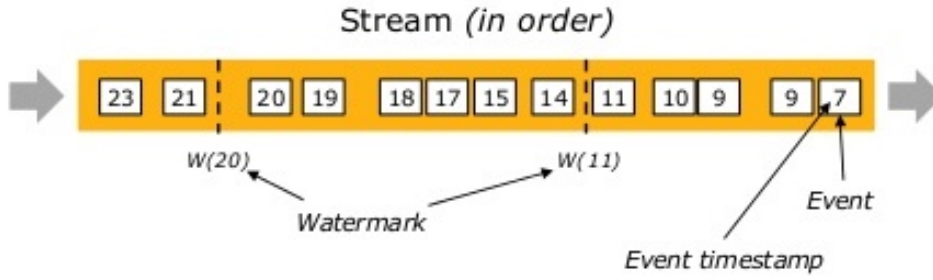


Figure 2.7: Watermark for a Stream in Order as Shown by [21]

is dependent on the goal of the application. Figure 2.8 shows an example of a watermark assigner function with an out-of-order stream. The watermark is always set equal to the timestamp of the element, where no elements with a smaller or equal timestamp will appear later.

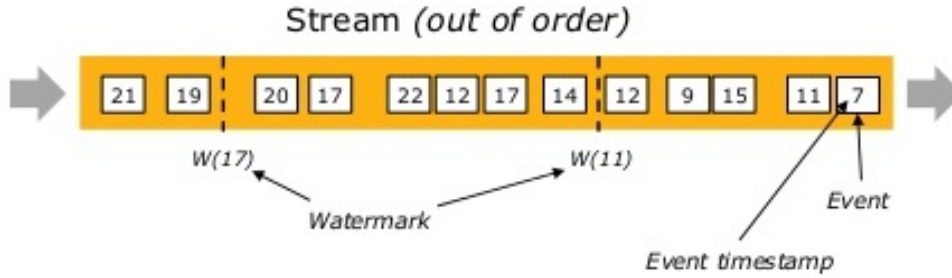


Figure 2.8: Watermark for a Stream Out of Order [21]

Since it is not known what elements will arrive later in the stream, the assigner function used in Figure 2.8 is not applicable in practice. In practice it has to be estimated how much a stream is out of order. Considering these estimations, a specific assigner function can be chosen. However, it is never guaranteed that there will be no elements after a specific time. It is always possible that there are incoming events with an event time that already passed the watermark timestamp. These events are called *late events*. The handling of these events is dependent on the goal of the application and if it is important that every event is considered. In some applications *late events* are processed equally as in time events and a trigger can be fired that recalculates the state. In other applications late events may just be skipped. The probability of *late events* is dependent to the watermark assigner function and the entities generating the data. If the watermark timestamp is set to the highest appearing timestamp in any event, there are a lot of *late events* in out of order streams. Yet, the results are returned faster as the event time is advanced and the trigger fires earlier. If it is set to a quite small value, *late events* are getting less likely. Nevertheless, the event time is delayed and the first output is returned later. More details regarding the handling of late events regarding the present work is shown in Section 4.3.5.

Now that the needed background information for this project has been provided, a look can be taken at related work in the next Chapter.

Chapter 3

Related Work

In this Chapter, the reader is introduced to work that is related to the thesis described here. The first part discusses both an Agent-Based Modeling and Multi-Agent System. Afterwards a look is taken at two papers that focus on test data generation in a streaming environment. Finally, a more detailed description of the Snappy paper is provided, which deals with the aggregation of streaming data into slices to reduce the recalculation of data required for windowing.

3.1 Multi-Agent System and Agent-Based Modeling

This section is dedicated to two papers that implement a MAS and ABM respectively. From this work, conclusions are to be drawn as to whether a MAS or ABM would best meet the requirements for the test data generation task of this project. The work presented in [22] deals with the implementation of a medical diagnostic system using a MAS. This paper originates from India and deals with the problem that many people live in remote areas where there is no adequate medical care and there is a blatant lack of trained doctors. It is also mentioned that many people are afraid to consult a doctor, even if only for a diagnosis, because it could lead to high costs. Another issue mentioned is that many people go to a local unauthorized healer. This in turn can lead to misuse of medication based on half-knowledge and guessing. Statistics from the WHO have revealed that such misuse of antibiotics often makes bacteria resistant to the first line of drugs and therefore the stronger and more expensive ones are needed.

The core concept is that a user enters the symptoms of a disease as raw data to a User Agent (UA) via an interface. The User Agent in turn converts the data into knowledge using the Master Agent (MA) and analytical methods. The knowledge is then stored in the Global Knowledge Base (GKB) and is available to all other agents. The main task of the MA is then to pass on the enriched cases to a selected specialist for the individual case. After a case has been solved by a specialist, the case and the solution are stored in a solution knowledge base. For example, if a user complains of high fever and headache, the User Agent can check via interface if the user also feels generally weak. If this is the

case, this information is aggregated and written to the global knowledge base. The MA then uses this knowledge to detect a high probability of malaria. This way a doctor can be notified and find a solution for the problem. Subsequently, a short summary has to be written into the solution knowledge base. Finally, a Report Agent (RA) can access the solution knowledge base and create reports from this data

This paper shows a system which divides the individual tasks for diagnosing diseases to individual agents and lets them work independently. The common goal is the efficient and safe diagnosis of diseases, which the individual agents work on in collaboration by writing and updating their knowledge in a central location.

In order to evaluate and test the quality of an Internet of Things (IoT) environment, a large amount of data is required, which often requires a lot of time and resources to create. Therefore, [23] is working on the development of an easy to use simulator and autonomous agent generator that simulates human activity in a household to generate such data. The simulator offers a 3D Graphical User Interface (GUI) that can simulate spatial configuration and virtual sensors that simulate real sensors. Various sensors can be defined and distributed throughout the house via a GUI, which react differently to interactions. For the actual agents in the house, certain scenarios with different behavioral consequences are defined that imitate human behavior. An agent-specific factor is always added, which either strengthens or weakens the action. Thus, for example, a distinction is made between sleepiness (low) and sleepiness (high), which then either trigger a short lie-in or to go to bed. The individual actions are subdivided into levels which are based on the order of need. For example the needs hunger and thirst are at a higher level than learning and watching TV.

The differences between a MAS and ABM approach to a similar problem are again highlighted by the two papers presented. A MAS focuses on the interaction of different agents to solve a central problem, while ABM is more concerned with modelling the interaction of individual agents in an environment. For the first part of this paper it will be of great interest to implement a system that adheres to the principles of ABM, since it is about modeling human behavior when using credit cards.

3.2 Data Generation

In [24] a system is explained, which in principle addresses a similar problem as the first sub task of this thesis. The aim is to be able to test a streaming system with the help of generated, reproducible data. However, the authors of the presented paper focus on varying the arrival of the events and thus the generation of out of order events. First the out-of-order characteristics of a source data set are examined and the min / max delay and fraction of such events are examined. For the source data set it is important that the fields are only flat format and that there is an event time field. Then the data generator modifies this out-of-order factor by setting delays on some events. To enable reciprocity, the generator operates deterministically using a three-step algorithm. This algorithm first performs a pre-processing by analyzing the out-of-order factor. Secondly, an out-of-order ingestion time is generated. This out-of-order ingestion time then changes the time that

the event should be consumed by the system. As a third step the final data stream is sorted by ingestion time. This ensures that the temporal data distribution of the event time is not changed but only the time at which the event arrives.

[25] describes a system to automatically generate synthetic data in order to enable analyzers to cope with insufficient data for machine learning models. The introduction mentions a problem similar to the present paper. Namely that the data needed for a project is insufficient or cannot be used due to data protection or similar reasons. In the present situation the latter is also the case. As a requirement for generated data, the authors identify two principles. On the one hand, the data must be statistically similar, on the other hand, there must also be structural similarity. As a solution, statistical distributions and a k-nearest neighbour algorithm are used. The elaborated algorithm is divided into six different steps. First all features with a categorical or date type are pre-processed to continuous numerical types. From these numerical values a k-NN model is generated to capture the correlations between the features. Then the estimated probability distributions are used to generate random samples. These samples are then validated against the previously generated k-NN model. Finally, the samples are post-processed, which is the reverse process to pre-processing. In contrast to the system introduced by [25], the present work goes a little further. No existing data sets are required, but only a statistical distribution. In addition, attention should also be paid to the temporal aspects of data generation. Also, a main focus of the paper by [25] is to generate data with a high correlation to the real data. This is not the main task of the work at hand. It is more about the simulation of a potential system load on certain keys or for elaborated test cases.

After a long and thorough search, the author could not find a paper to the best of his knowledge and belief that already covered the sub task of this thesis, which is the generation of statistically distributed data for a streaming system with millions of concurrent entities.

3.3 Snappy

The considerations made in the paper by [26] play an important role in the second and third part of the project. These considerations however could not be directly adopted due to the differing characteristics of the data involved. Nevertheless, the approach remains highly interesting and is certainly worth mentioning at this point. In [26], Traub et al. describe the first general stream slicing technique for windowing operations. As a major driver for performance problems, they identify large overlapping computations in sliding windows and concurrent queries. The proposed technique is based on the fact that individual events are pre-aggregated into so-called slices and then only these events have to be used for the calculation and not all values. The approach is described in Figure 3.1. When evaluating the proposed system, the authors of the paper found that the throughput was higher by an order of magnitude with the help of slicing than without it. However, an important point to note with regard to the work introduced by [26] is, that it is assumed that many individual events can be combined into slices. This assumption does not apply to the present paper. For credit card transactions aggregated by card number, it cannot be assumed that the benefit of pre-aggregating the sparsely populated keys into slices

would yield a great benefit. Nevertheless, the technique must be looked at and evaluated carefully with respect to this paper. The breakdown into slices could possibly be of great interest for aggregations by country or individual merchants.

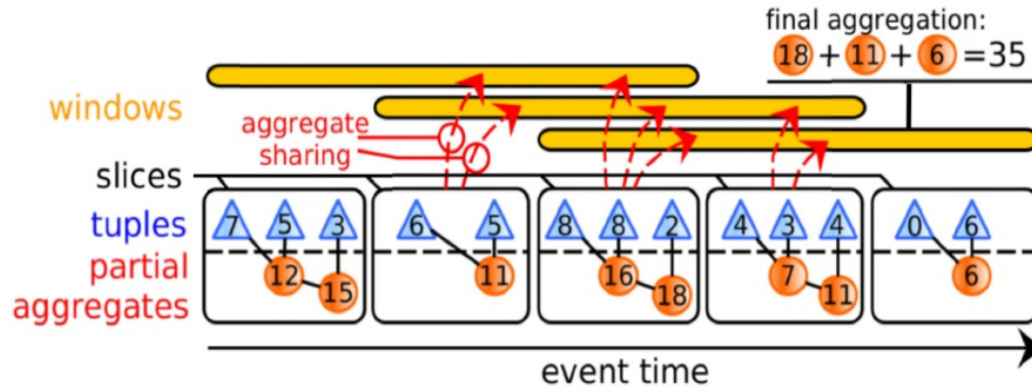


Figure 3.1: Aggregation of Tuples Into Slices as Proposed by [26]

Chapter 4

Design

This Chapter explains the design decisions taken for the final solution. In the first Section 4.1, the general architecture of the approach is discussed. Afterwards, a look is taken at the difficulty of test data generation 4.2 and how the problem of dependent random variables will be solved by means of augmented Red Black Trees 4.2.1. In a third Section 4.2.2, the approach to create random generators is explained and the link to the tree structure mentioned earlier is made. Additionally, this Chapter also describes the chosen approach to enable the data generation to be deterministic in Section 4.2.3. Finally, the design decisions taken for the aggregations of the data in Flink are discussed in more detail.

4.1 Architecture

The general architecture of the proposed solution includes the three parts described in Section 1.1. This involves the generation of statistically distributed test data, the aggregation of the same in sliding and tumbling windows as well as the reclassification of data from windows. The latter is necessary because it can happen that an authorisation or transaction has been classified as non fraud, but will be reclassified as fraud at a later time. Therefore it needs to be removed from the corresponding fraud/non-fraud windows. An overview of the required data flow can be found in Figure 4.1. The numbers in the diagram correspond to parts 1 to 3 of the project work.

4.2 Difficulty of Meaningful Test Data

When talking about generating test data, one probably thinks of simple numbers or strings of characters which have no further correlations. However, if one thinks of a credit card company, which like any other company has the need to test its systems with data that is similar to production, it slowly gets understandable how complex this topic is. The next Section will therefore provide an overview of the problem at hand.

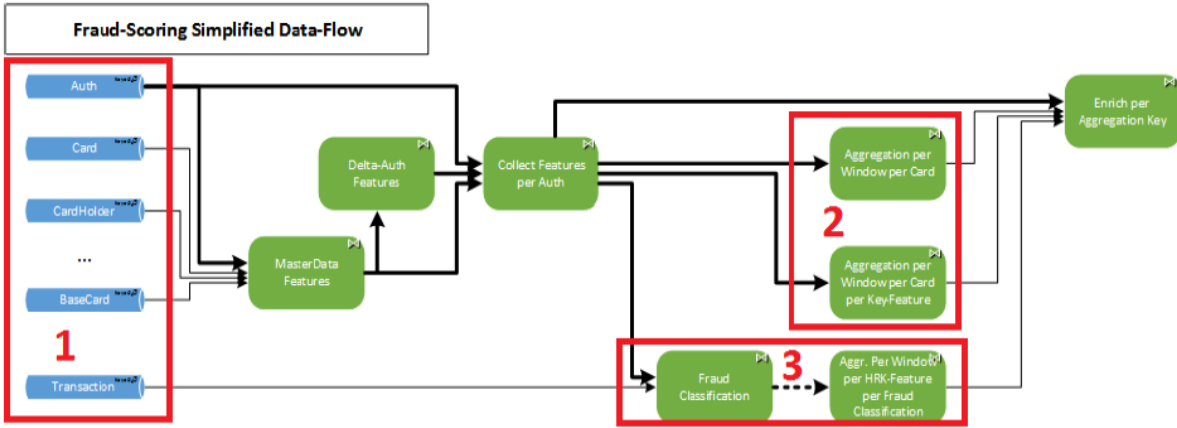


Figure 4.1: Dataflow Diagram of the Data Preparation for Fraud Scoring

The following challenges arise when considering the test data generation. Assumed a customer owns one or more cards, which one will be used for transactions? A customer lives or works at a certain place. Therefore, most of the transactions will be made near that location and only with little probability or never will something be bought on the other side of the world. The customer also has certain preferences and the current location determines at which merchant to make these transactions. Still, not every place has a merchant that sells the assortment to meet the specific needs. A customer can of course also be on holiday and thus be able to make transactions anywhere in the world. During holidays, the preferences will (in most cases) be different from those at home or work. These are just a few of the issues that make the problem complex. Even finding a data structure that can map dependent random variables is a thought-intensive task, which is explained in more detail in the Section 4.2.1. Furthermore, attention must be paid to performance. If considered that every single credit card terminal in the world (as well as internet transactions) have to be covered, several million objects will need to be handled efficiently.

Therefore, the research questions (RQ) which arise are the ones mentioned below. The proposed solutions to these RQs are described in the subsequent sections.

- RQ1** *How can dependent and independent statistical distributions be efficiently stored?*
- RQ2** *How can an easy-to-understand system based on composition be constructed that allows random variables to be generated from the previously mentioned distributions and put them into a temporal context?*
- RQ3** *How can determinism be enabled in a complex system that relies on randomness?*

4.2.1 Augmented Red Black Tree

In general, a distinction can be made between dependent and independent random variables. Independent variables can be described as random variables that have no influence on the other random variables in an experiment. This means that if an event occurs, it has no influence on the probability of the occurrence of subsequent events. An example for this are coin tosses, where there can be either heads or tails, but the individual tosses are independent of each other. Formally, independent random variables can be defined by the conditions below, which state that knowing the value of x says nothing about the distribution of y . The first equation defines that for all values of x and y the probability that x follows y is the probability of x alone. The second equation states that the probability of x and y is the multiplication of each others probability. Dependent variables on the other hand are the opposite and affect the probabilities of other random variables.

$$P(x|y) = P(x), \text{ for all values of } x \text{ and } y$$

$$P(x \cap y) = P(x) * P(y), \text{ for all values of } x \text{ and } y$$

Since potentially many dependent probabilities have to be stored and queried, the necessity for a well thought-out data structure is evident. The basic needs were to be able to store a key together with an arbitrary value and a probability. This raises the question of how a random value can be drawn from a data structure based on its probability. If looking for possible solutions to this problem, one will only find trivial solutions that are unsuitable for the case at hand, as they are not dynamic enough or scale badly. The simplest solution that is mentioned in various blog entries is to have a long and nested `if` or `match case` statement, which triggers a different action depending on which number is drawn (*cf.* Listing 4.1).

```

1 rnd = Random.nextInt(0.100)
2 if(rnd == 0 && rnd < 20) println("case 1")
3 else if( rnd >= 20 && rnd < 70) println("case 2")
4 else if( rnd >= 70) println("case 3")

```

Listing 4.1: Probabilities With If Else Structure

Needless to say that this is not a feasible option as it is far too static and the solution should be able to handle any datatype without the need to change the structure. Another approach, proposed by [27], suggests to multiply the probabilities by a constant multiplier to get rid of the fractions. Then, an array with the accumulated size of the multiplier should be created. After that the values are placed into the individual fields of the array based on their weights.

$$P(1) = 2$$

$$P(2) = 3$$

$$P(3) = 5$$

Therefore, the array needs to have a length of 10 and contains the following data

$$\text{Array}(P) = (1,1, 2,2,2, 3,3,3,3,3)$$

This allows to draw a random index from the array and get the corresponding value. But if it is now assumed that millions of entries and arbitrarily large weights should be stored, it quickly becomes clear that this cannot be a possible solution.

[28] provides an easy to understand metaphor for further thought games. The values A, B, C, and D are given with the following probabilities.

$$P(A) = 0.1$$

$$P(B) = 0.3$$

$$P(C) = 0.4$$

$$P(D) = 0.2$$

This means that in 10% of the cases one would like to get an A, in 30% an B, 40% an C and 20% a D. If one now imagines these probabilities on a bar, the resultant pattern will be the same as depicted in Figure 4.2.



Figure 4.2: Metaphorical Stick With Intervals

Subsequently, the individual probabilities have to be summed up as shown in Figure 4.3. Finally, it is possible to get a random number between zero and the maximum stick length, which will yield a key based on it's probability. For example, if one chooses $p = 0.3$ as shown in Figure 4.3, B would be returned as a result. In the presented example, absolute values (*i.e.*, 0.3, 0.2, 0.4, 0.1) were used. The same principle would of course also work for relative probabilities if they were normalized to the unit of the structure.

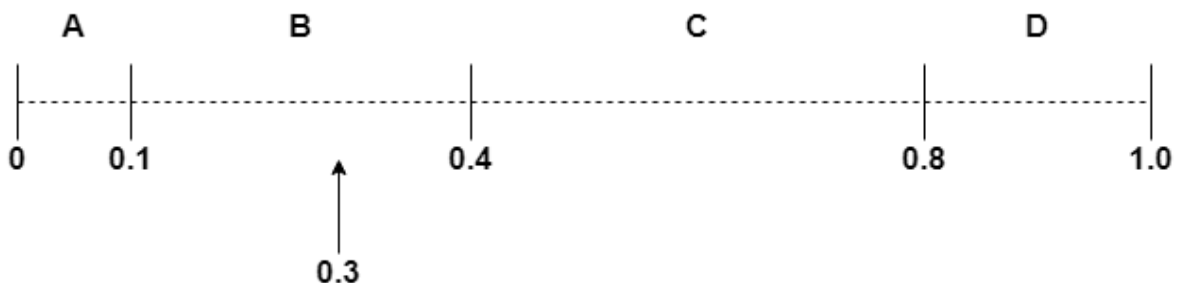


Figure 4.3: Metaphorical Stick With Summed Probabilities

These first ideas led to the consideration of using the already existing TreeMap in Scala. This data structure builds a Red Black Tree [7] and allows to store a value and a key on

a node. A possible approach would be to use a key/value pair, where the key could be a tuple that makes up a probability (*e.g.*, customer name = “Ruckstuhl”, nationality = “Switzerland”). The value would accordingly be a number representing the probability. In the previous example 90% felt. The key/value pair is now turned around and the probability is taken as key and the former tupled key (*e.g.*, name = “Ruckstuhl”, nationality = “Switzerland”) as value. While building the tree, a running sum of accumulated weights is kept. Every inserted node then adds its weight to this running sum and takes the result as the new key. This finally yields ranges that correspond to the probability of the added node. An example for such a Tree can be found in Figure 4.4 where nodes with weights 10, 180, 110, 200, 300, 200 and 300 are inserted. On each insert, the weights of the nodes are accumulated with a running sum and then inserted into the tree.

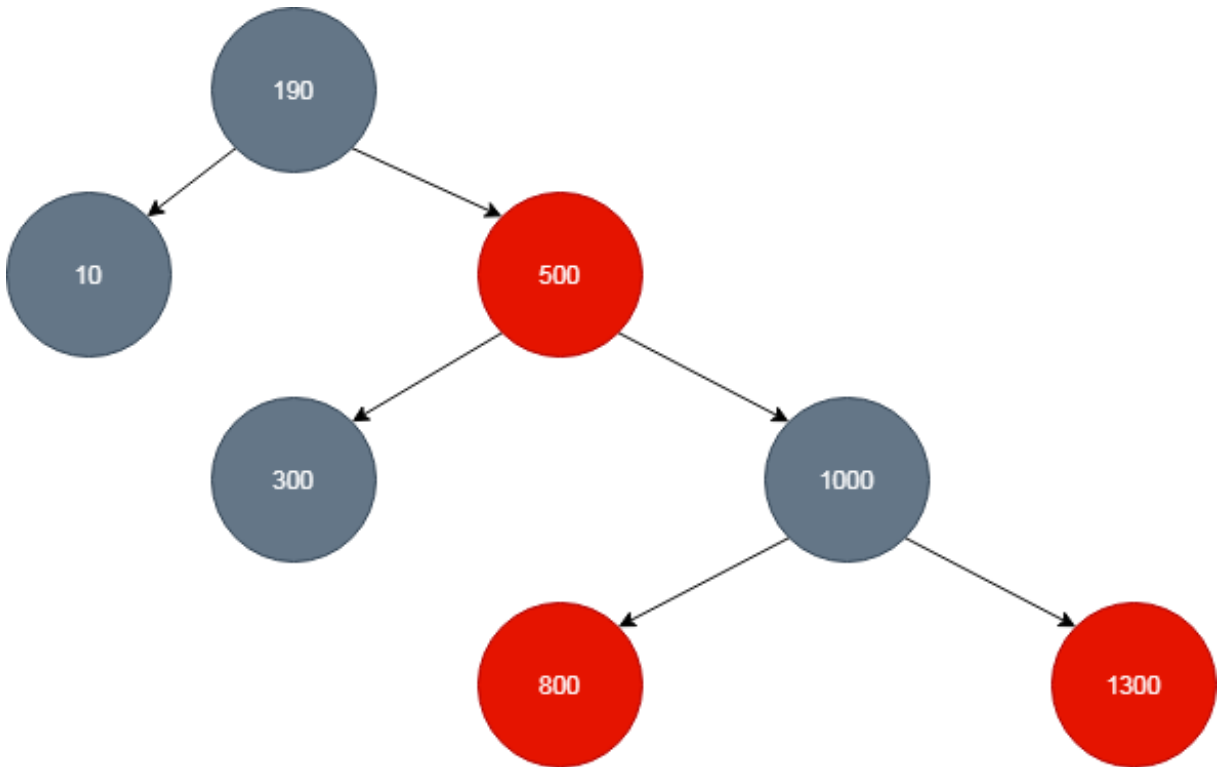


Figure 4.4: Example of a Scala TreeMap With Weights as Keys

At first glance, this seems to be exactly what is needed to represent dependent probabilities. For each nesting depth a 2, 3, ..., n tuple could be created with the pre-calculated probabilities. However, after further consideration, it is clear that searching for single original keys is no longer possible and only searches with $\log(n)$ for prefixes can be performed. Especially for queries which have to return all objects of a certain region or a certain segment this would be absolutely necessary.

So what is missing is a kind of second index on the trees. The tree must be sorted by key, but also allow a prefix search. Since no out-of-the-box solution for this problem was found even after a lengthy search, an independent solution was designed.

For this purpose, the basic ideas of the Red Black Tree implemented in Scala are extended to meet the requirements which include:

- The basic properties of Red Black Trees are met
- New nodes are sorted by an arbitrary key
- The nodes can have generic objects as values.
- The tree facilitates efficient $O(\log(n))$ prefix searches
- Possibility to execute range queries

The exact implementation of the tree is discussed in detail in Chapter 5. At this point, the design decisions made for the tree are explained. Meeting the first three properties is trivial and has already been implemented in various examples. What remains is the augmentation by weights. Each node has its own weight and the aggregated weight of its child nodes. If a new node is added, the newly aggregated weights are recursively passed upwards to the root node. This allows to perform a prefix search. A random number between 0 and the aggregated weight of the root node is chosen. It is then checked if the searched number is greater than the aggregated weight minus the weight of the node. If so, this node is returned. If not, it is checked whether the searched value is greater than the aggregated weight of the left node. If not, the left side is considered, otherwise the right node. The steps are repeated recursively until either a leaf node is reached or the value is found. This way the same effect is achieved as with the previously mentioned stick with the summed probabilities as ranges. However, the structure remains dynamic and keeps the actual key as index. An example of such a tree is depicted in Figure 4.5. It shows a tree with the summed weights and the final sum in the root node. The previously explained algorithm is shown in 4.2. Note that when the right child node is considered, either the aggregated weight is subtracted from the prefix or the prefix is taken directly. The reason behind this is described in the implementation in Section 5.1.1.

```

1  def getNodeWithPrefix(tree: RBT[A, V, W], prefix: W): RBT[A, V, W] =
    {
2      if (prefix >= tree.aggregatedWeight - tree.weight) tree
3      else if (tree.left != null && tree.left.aggregatedWeight >
        prefix)) getNodeWithPrefix(tree.left, prefix)
4      else {
5          if (tree.right != null) {
6              if (tree.left != null) {
7                  getNodeWithPrefix(tree.right, prefix -
                        tree.left.aggregatedWeight)
8              } else getNodeWithPrefix(tree.right, prefix)
9          } else tree
10     }
11 }
```

Listing 4.2: Prefix Algorithm

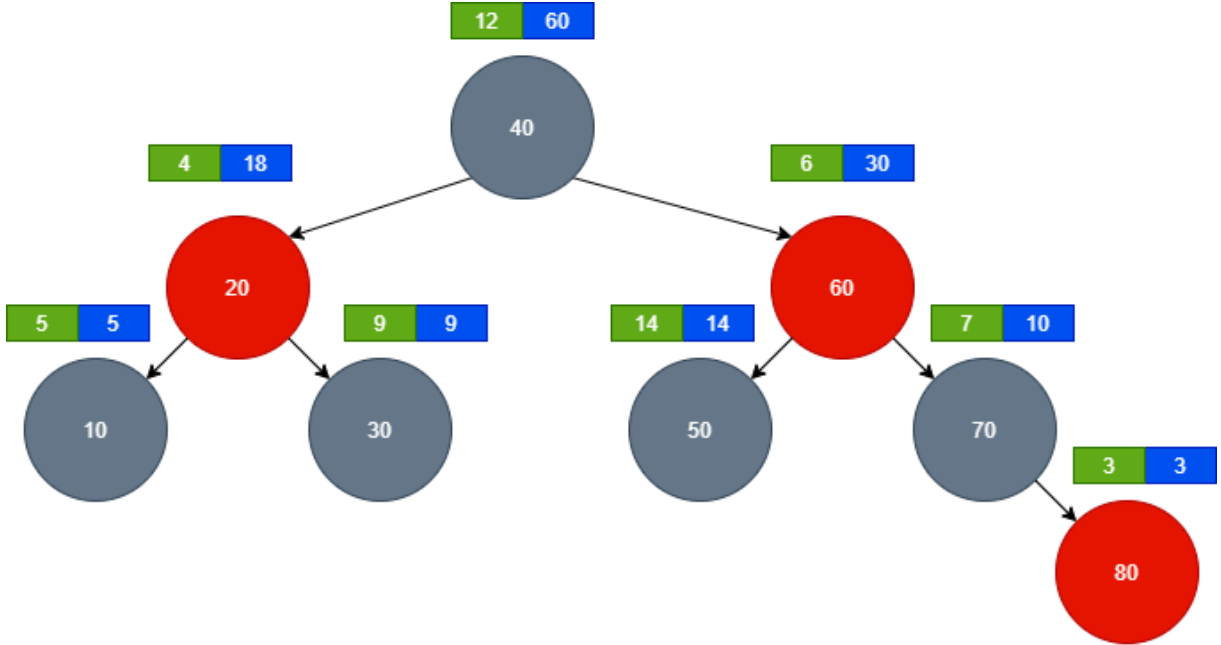


Figure 4.5: Example of a Tree With an Aggregated Weight of 60 in the Root Node

After clarifying how dependent random variables should be stored efficiently, the report now focuses on generating entities based on statistical distributions.

4.2.2 Data Generation

In the first part of the project, the data generators are responsible for the actual creation of the random entities. They use the concept of monads, which were introduced in Section 2.3. As a reminder, they can roughly be defined as complications and calculations in a chain of operations. The monad allows (among others) to model statistical distributions and compose them to larger distribution models expressing causal dependencies. In the end, random objects respecting the model can be generated.

As an example for such a generator, the following lines describe the creation of a merchant and its usage by a randomly created customer. For this, the statistical distribution of the merchant category codes (mcc) stored in the elaborated data structure and a random region are needed. In a first iteration, these regions are limited to Switzerland, the EU and the world. The selection of a region will be discussed in more detail in Section 5.1.6. To create a new merchant, a random mcc from the tree of merchant distributions is drawn as well as a random region in the world. These two probabilities are not yet dependent in the current model, but with further complication they could become so. As an example, it could be stated what the probability of a certain mcc is to occur in a specific region. Greenland will clearly not have as many merchants for swimming suits and beach towels as Brazil. After selecting an mcc and a region, a certain card usage patterns is assigned to a merchant. This has to be done as the number and content of authorisations and transactions that get emitted from a petrol station are different to a refunding scenario

or a standard card usage at a terminal. Currently, the following four different cases are considered:

- **Petrol** An authorisation is emitted with a standard reservation amount. When the customer is done with fueling, another authorisation is emitted that contains the difference between the reserved and actually used amount. This authorisation is used to cancel the unused amount off the previous authorisation. After some time a transaction is emitted that contains the effectively used amount from the second authorisation. This happens when the transaction is booked to the customer account during overnight processing.
- **Card Validity Check** An authorisation with a small amount is emitted to check if the card is valid. Shortly after there is another authorisation following that cancels the first one. This is mostly done by online merchants.
- **Standard** An authorisation is emitted, followed by a transaction.
- **Refund** Authorisation and transaction like in the standard case, but after some time a transaction is emitted that reverts the spent amount from the first transaction.

Again, a tree is built up that holds the probabilities of the individual patterns for a specific mcc. In the present simplified model, the differentiation between refunding and standard case is made by looking at the statistical distribution of transactions with `amount < 0` and `amount > 0`. The ones smaller than 0 are refunding transactions, the one bigger than 0 are standard ones.

Now that the merchants can be created, there also need to exist customers to do the transactions. The special feature of the customers is that they should exhibit an independent behaviour, which is determined by a simplified model. This means that they must be able to change their location (in reasonable time intervals and regions), change their preferred merchants as well as cards in case they expire or get blocked due to fraud or loss. These processes are shown in Figure 4.6 and make it obvious that some structure is needed which can handle timelines concerns. This means that the whole system needs to be able to delay an operation based on model time. Still, this operation must not stop the other processes running concurrently. To ensure this, a monadic structure should be implemented that allows to describe the generation of test data, temporal concerns, concurrency and synchronization as well as facilitate a global state and handle general side effects. Therefore, the in Section 2.4 introduced FreeMonads will be used, as they offer a pattern for such needs. The elaborated Temporal Event (TE) monad will contain the ADT cases below, which allow to cover every operation needed in order to meet the requirements of project part one.

- **Pure[T](t: T)** Lifts the supplied value T into the TE monad.
- **Now()** Returns the current model time as TE[Long] where Long is the timestamp in epoch milliseconds
- **Delay(millis: Long)** Delays further execution of the steps in the program by the specified milliseconds.

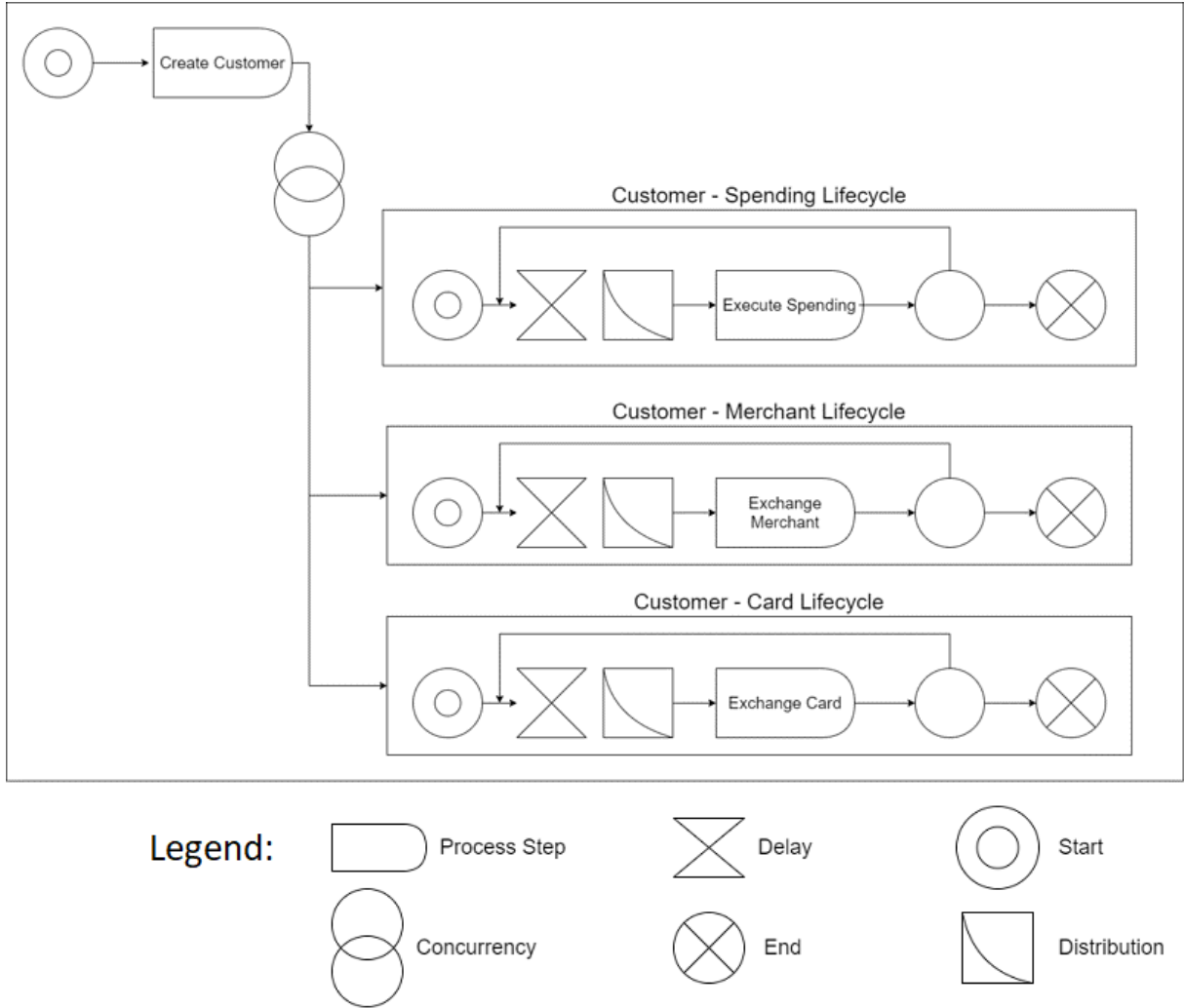


Figure 4.6: Diagram of a Customer and its Lifetime Cycle

- **Concurrent[T](run: TE[T])** Runs a specified program concurrently
- **Emit[T](channel: String, t: T)** Emits an event to a stream
- **Generate[T](gen: Gen[T])** Executes the generator supplied as a parameter and generates a random value from the given distribution. Determinism plays an important role here, which is discussed in more detail in Section 4.2.3.
- **Effect[T](effect: () => T)** Executes a piece of code that contains side effects. This allows side effects to be isolated.
- **PutToStateBag[T](t: Symbol, element: T)** Operation to write a value into a global state bag similar to the example in 2.4
- **GetFromStateBag[T](t: Symbol)** Operation to get an object from the state bag of the compiler.
- **SetSeed(s: Long)** Operation which sets the initial seed of the pseudo random generators. This is discussed in detail in Section 4.2.3.

The exact functionality of the TE monad is explained in Section 5.1.2. After implementing this structure, it will be possible to write programs that will generate test data when interpreted. What is missing is the possibility to make the generation fully deterministic. For this purpose, the approach described in the next section has been designed.

4.2.3 Determinism of Data Generation

An important aspect of any system development is certainly the testability of the written code and the resulting artifacts. Guaranteeing this for random data generation is an integral part of the further work. To ensure that the same initial random value generates the same final state, a seed object will be required. It stores the current random value and, if necessary, calculates a new value based on a mathematical function. The difficulty is that every time a function is called that requires a random value, the seed object needs to be passed on. This will be done with the aid of the envisioned Temporal ADT compiler and the Gen monad. The Temporal ADT will offer a case called Generator, which executes a Gen monad that is passed as function parameter. At the same time a new random value is calculated which can be used directly by the next call. Furthermore, the compiler contains the method SetSeed, with which the seed can be set initially. As a second essential part, the previously explained characteristics of the TE monad are needed. By creating a temporal context and thus a total order, it can be ensured that all operations for an identical starting value always produce the same output. This total order also makes it possible that side effects can be executed without getting race conditions, since side effects are also executed in this sequence. This is of course under the condition that all operations to generate random data are in the chains of the TE monad. Now that all questions regarding data generation can be answered, it is possible to draw a conclusion about the potential solutions to the initially presented problems.

RQ1 How can dependent and independent statistical distributions be efficiently stored?

- *Answer:* With augmented Red Black Trees as shown in Section 4.2.1.

RQ2 How can an easy-to-understand system based on composition be constructed that allows random values to be generated from the previously mentioned distributions and put them into a temporal context?

- *Answer:* With the Temporal ADT and the Gen monad as presented in 4.2.2.

RQ3 How can determinism be enabled in a complex system that relies on randomness?

- *Answer:* With seed objects and the TE monad presented in Section 4.2.3.

4.3 Data Aggregation

Taking the generated test data as input for the aggregations, a way need to be found to extract different features from the data as fast as possible and as memory preserving as possible. This section describes the investigated methods to update the implemented features continuously as soon as new events are delivered. As already mentioned previously, Apache Flink is used as stream processing framework in the project. Possible example features, which need to be extracted from the incoming data, are listed below.

- The average amount of the authorisations of a specific customer in the last 24 hours.
- The number of authorisations processed by a specific merchant in the last 30 days.
- The standard deviation of the authorisation amounts in a merchant category in the last 24 hours.
- The number of days a specific customer did a transaction during the last 12 months.

By a first observation of these features, one can see that each one of them focuses on a specific key (*i.e.*, customer, merchant, merchant category code) and a time interval (*i.e.*, 24 hours, 30 days, 12 months). This is the case for most of the needed features considered in this project.

Regarding that the end time of such a time interval is the current event time and the time interval encompass 24 hours its start time is exactly 24 hours before the current event time. Since the event time is dependent of the processed data, the event time, which will be after a new event is processed, is not known before the new event actually arrived. Hence the exact start and end time of the new time interval are not known before a new element is processed and therefore an according window for the time interval can not be created in advance. Since the events are randomly distributed, there is also no general regularity when the events happen and it can not really be estimated when a window is necessary. Considering these aspects, it is not possible to have a window assigner, which fits the incoming data perfectly. Thus, the question arises how it can be assured that only relevant events are contained in the window.

Having in mind that the goal is to incrementally process the incoming events, incremental functions have to be found to calculate the features. Furthermore, assuming that the incoming events are not in total order and there are late events, the elaborated functions also needs to be commutative. This gets clear by looking at the equation 4.3, where the left side of the calculation is a stream in order with the events a, b and c and the right side is a stream with the same events but event a is a late event. The events are processed in a different order, but the final result has to be the same.

$$Event(a) + Event(b) + Event(c) = Event(b) + Event(c) + LateEvent(a) \quad (4.1)$$

By looking at the 4 features listed before in this section and having a commutative and incremental function in mind, some more observations can be made. For the first and the second feature it is easy to find such a function. Especially for counts, where the current aggregated value just has to be increased by one when a new event arrives. For the third feature, Welfords method can be used [29]. However, for the fourth feature it is actually not possible to incrementally calculate the final value. This is due to the fact that in a count distinct problem every passed value has to be known to decide if the final value has to be increased due to the current event. Therefore, a solution has to be found that handles aggregations, where no commutative and incremental function exists.

The following list summarizes the previously detected challenges.

- RQ1** How can a set of windows efficiently cover time intervals, when their start and end time is not known exactly in advance?
- RQ2** How can it be ensured that only relevant data is considered for the calculations?
- RQ3** How will late events be handled?
- RQ4** How are features handled, where no commutative and incremental process function is available?

The final solution has to solve all these challenges and still be able to operate efficiently. In the following section, the investigated methods, which solve the first and the second challenges are presented. The methods are described along with their advantages and disadvantages. All the explanations will be based on the feature that calculates the average amount of the transaction of a specific customer in the last 24 hours. Subsequently, the other two challenges are discussed.

4.3.1 Sliding Windows

Considering the given feature, a first opportunity to solve the problem is to use sliding windows with a window size of 24 hours and a new slide every second. When a new event is received, a window over the last 24 hours from the new event time with a precision of 1 second exists. This is probably the easiest approach to understand and to implement, as mostly out-of-the-box features of Flink can be used. An example for this method is shown in Figure 4.7 with a window size of five seconds. Assuming that the elements will arrive at the time points 3, 5 and 8, the continuous aggregated values of window C, E and H can be taken as result for the feature without doing anything more.

However, as stated in Section 2.8.2, in sliding windows an element is assigned to $windowSize/windowSlide$ windows. Hence, one element is assigned to five windows in this example. For each added element five windows have to be updated concurrently which can be considered inefficient. Additionally, windows A, B, D, F, G, I, J are created. Yet, their value is not considered. In this example, 70 % of the created windows are never used later on. Considering a window size of 24 hours and one slide per second, 86400

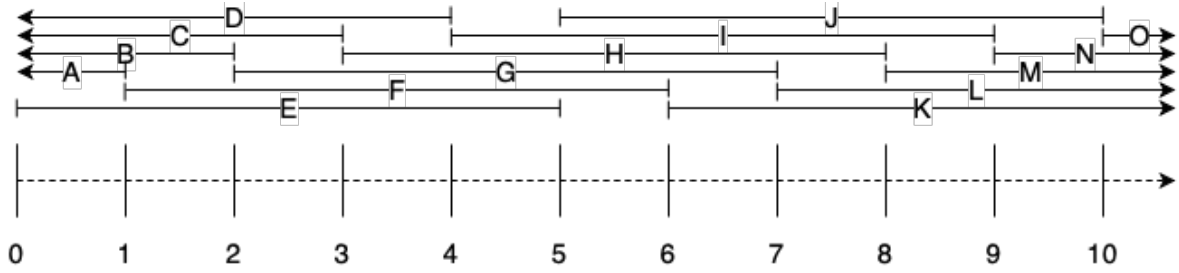


Figure 4.7: Sliding Window With a Window Size of 5 Seconds and one Slide per Second

windows need to exist concurrently. Although only a small part of them is needed later, each aggregation value needs to be updated when a new event occurs. Additionally, it is not possible to calculate different values when there are more than one event in the same second. Considering these disadvantages, this approach can not be used in the present work as performance is one of the main concerns.

4.3.2 No Window Assigner

Another option is to use no window assigner and just use a trigger, which fires each time an event arrives. Additionally, something like an evictor is used to remove all events that are not relevant anymore. This leads to the state containing all events together with the time when they should be deleted. In Figure 4.8, three stages of such a global window are shown. The event time is always updated to the time of the newest event. The green elements are the elements, which are currently contained in the window. The red elements are already evicted and the blue ones are the ones which did not arrive yet.

In Subfigure (a), it is 12 o'clock and on the right side two new events are arriving. In Subfigure (b) the state after the first element was processed is shown. At the left side two elements (older than 24 hours) are evicted. In Subfigure (c) the second event was processed and nothing had to be evicted. By doing so the aggregated value just needs to be adapted with the newly added and removed values. This can be done incrementally without looking at the events, which are currently in the window. However, in order to know which events have to be evicted the condition $eventTime - 24hours$ must be checked. During the processing of the second event in Subfigure (c) no element needs to be evicted. Thus, it gets obvious that it is not needed to iterate through the whole list whenever a calculation is triggered. To avoid unnecessary iterations, a timer is registered firing at $eventtime + 24hours$. When such a trigger fires, the event matching the deletion time is removed. To efficiently remove the correct value, a map is used. The time to delete is taken as key and the value is the event itself. Hence, only a simple `map.get(eventTime)` request has to be processed and the value stored in the map can be removed from the aggregated value. When the first element is processed in Figure 4.8, two timers are reached and the explained process is therefore executed twice. It does not matter which one of the two elements is removed first since the function is commutative. When the second element is processed no timer is reached and therefore no element is removed.

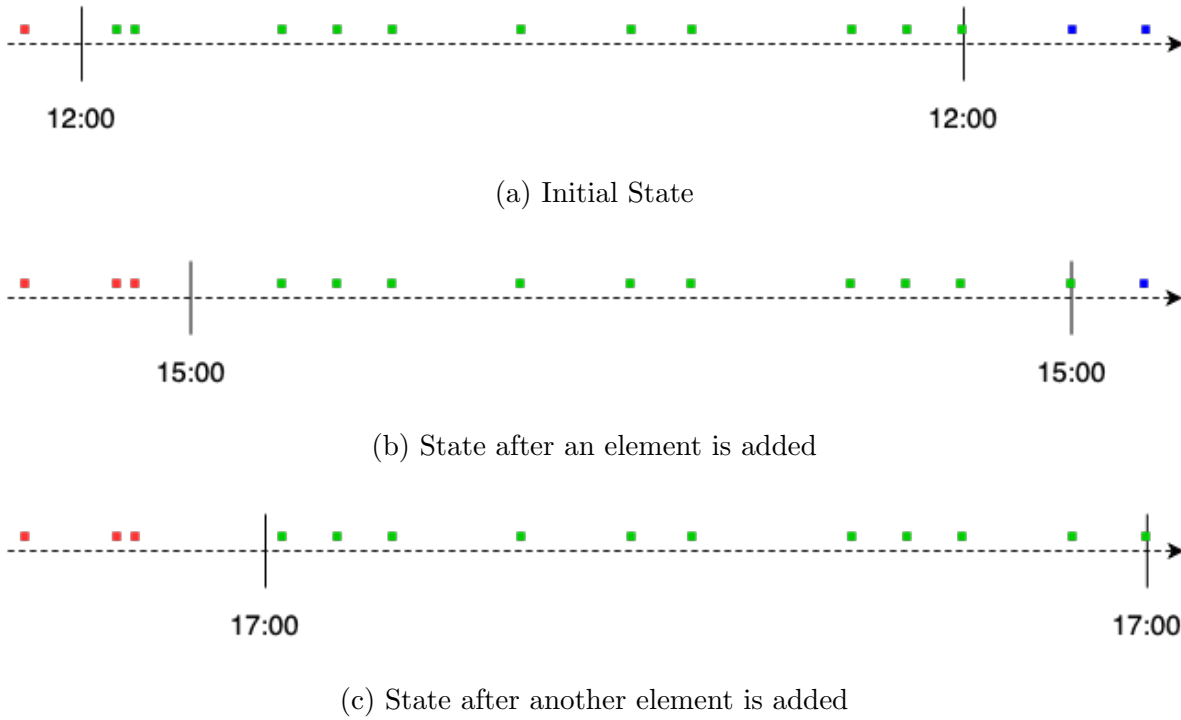


Figure 4.8: Only an Evictor, Which Deletes non Relevant Transactions

This approach contains some complexity, since timers need to be used in order to delete the correct elements. Regarding performance it constitutes an improvement compared to the first method. However, each event has to be stored during its lifetime. This leads to high control over the elements, but each one needs to be kept in memory, which is not desirable. This contradicts the second desired characteristic of stream processing mentioned in Section 2.7. Nevertheless, under certain circumstances it can be necessary to use this approach in order to get satisfying results.

4.3.3 Global Window With Tumbling Window Slices

The third investigated approach is an extension of the second one explained in Section 4.3.2. It tries to improve the performance by pre-aggregating subsets of the elements in so called window slices. The main idea of it was introduced by [26] mentioned in Section 3. In this method the input stream is split into tumbling window slices with a defined length. Each tumbling window slice processes its elements and forwards the results to a global window. The global window then combines all the values from the slices and calculates the final value. When a value in a slice is added or removed only the slice the element belongs to has to recalculate its pre-aggregated result. Subsequently, the new result is forwarded to the global window, which then can also recalculate the final result with the updated value of the changed slice.

As in the previous method, each value has to be stored to be able to remove outdated values, which is not desirable. However, when it is acceptable that the main time window does not exactly comprise the defined time window this method offers great possibilities.

Instead of removing single outdated events it can remove whole outdated slices. In the following subsections, two possible approaches are described. In the first one the main window comprises exactly the specified window size. In the second it comprises approximately the specified size and removes whole slices.

Window Slices With Exact Main Window Length

As mentioned in the introduction of this section, window slices can be used to apply some pre-aggregation function to its elements and forward the aggregated value. As an example, the slices could calculate the average value of its elements and forward it together with the number of elements contained. The global window can then aggregate all the values from the slices to a final output.

Considering that the period of the last 24 hours is split into 1 hour tumbling window slices, the global window consists of 25 slices of 1 hour tumbling windows. There are 25 slices, as the first and the last slice do not consist of a full hour. Considering a new event comes in at 13:20, then the first window slice consists of the time between 13:20 and 14:00 on the previous day. The last window slice consists of the time between 13:00 and 13:20 of today. So together the first and the last slice encompass a full hour. The other 23 window slices are all exactly one hour.

When a value is added or removed from a window slice the aggregated value changes has to be forwarded to the global window. The global window then removes the aggregated value from the time window and only includes the new one in its calculation. As a new event only has impact on the window slice it belongs to, the other slices stay the same. Thus, only the changing slice needs to be updated. The global window then aggregates the new value together with the existing ones. Since the final result should not be dependent of the partition in the window slices, the process function has to be associative and the equation 4.2 has to hold.

$$(Event(a) + Event(b)) + Event(c) = Event(a) + (Event(b) + Event(c)) \quad (4.2)$$

Figure 4.9 shows the concept of this method again with the same example as used before. The numbers in brackets over each slice are the aggregated values per slice, namely the number of elements in a slice and the average value of them. The main global window just aggregates all these intermediate aggregations and calculates the final result. When a value is removed or added to a slice the aggregated value is updated as seen in state (b) and (c). The global window then just calculates the result based on the updated values.

For the removing of elements, event timers are used. They signalize that an element has to be evicted from a specific slice as similarly done in the previous method. Again, each relevant element has to be stored in memory. For simple use cases as the calculation of the average this method adds just unnecessary complexity and the method without a window assigner is more efficient.

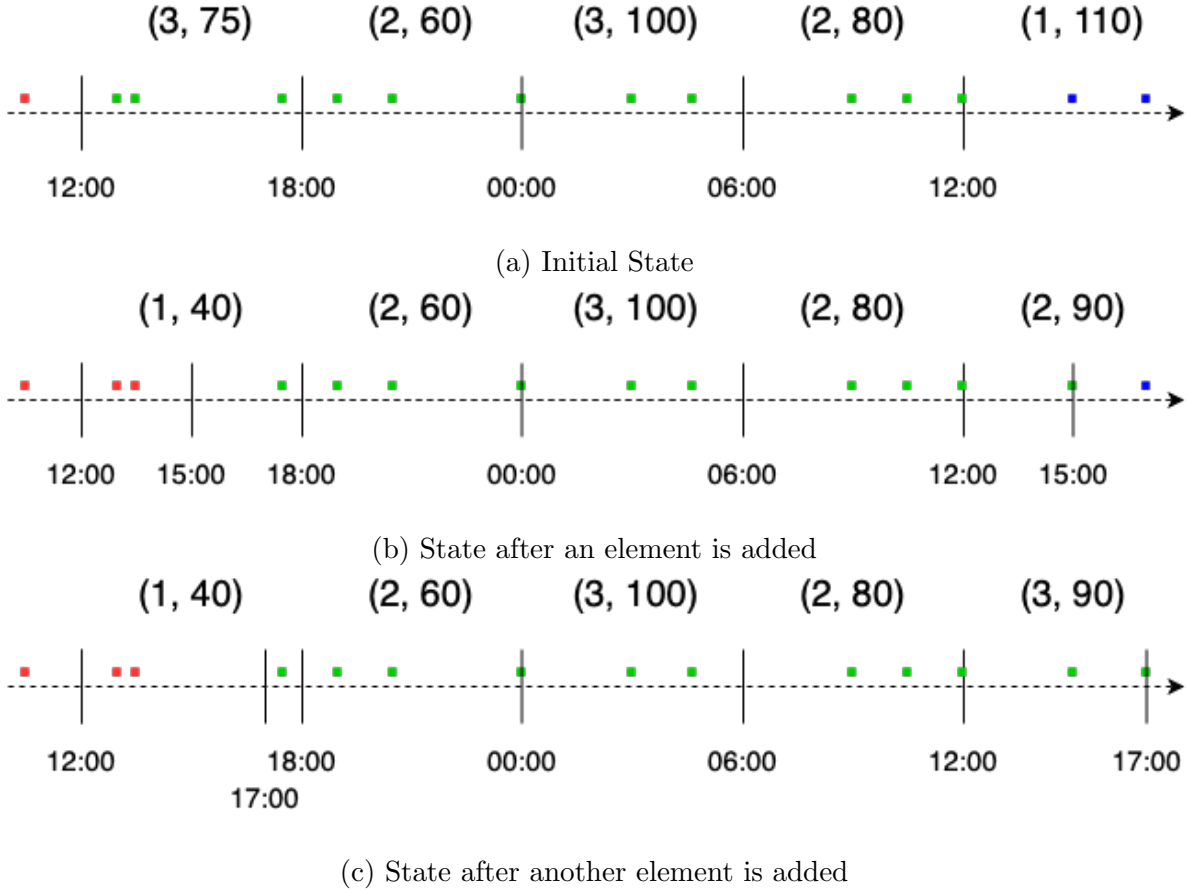


Figure 4.9: Global Window Split in Moving Slices

Window Slices Where Main Window Length is Not Exact

Considering that there is a set of window slices and one global window, which combines the relevant window slices, there is another opportunity than the aforementioned. It solves the performance and storage issues in cost of precision. Instead of deleting each element in a window slice, the global window just deletes the whole slice when its time is over. Having again a period of the last 24 hours split into 1 hour tumbling windows, the global window consists of 25 slices of 1 hour tumbling windows as before. However, instead of making the first slice smaller when time moves on it stays one hour long. Thus, at 13:20 the first slice contains the data between 13:00 and 14:00 of yesterday, instead of just between 13:20 and 14:00 as it was before. The last window slice still consists of the time between 13:00 and 13:20 of today. In total the slices comprises 24 hours and 20 minutes in this case.

The same example case as before is used in Figure 4.10 with the concepts of the fixed slices. In the transmission from state (a) to (b) two elements were evicted in the previous method. In this method the first slice is not touched and comprises the elements of the full hour, hence no element is evicted at this point. During the time of a slice elements are just added. At the time the slice becomes irrelevant, the whole slice will be evicted. In this example, the first window slices stays relevant until 18:00 the next day. Like this no single element has to be removed and hence only the current aggregated value per slice

needs to be known.

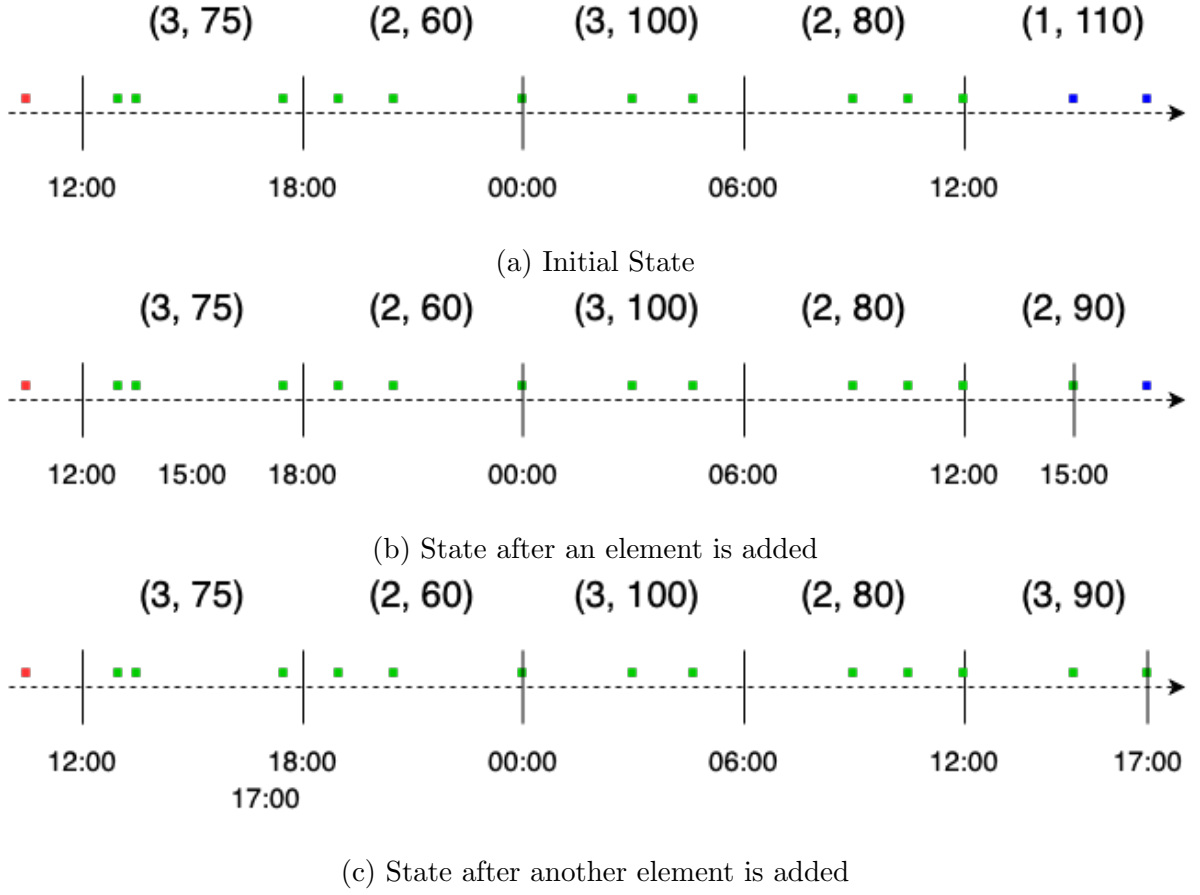


Figure 4.10: Global Window Split in Fixed Slices

This method is fast and does not need much space in storage. Yet, it does not exactly calculate the values of the specified time. The precision grows with the number of windows. In the case of the 24 hours, window precision is considered important. Otherwise it can happen that a time, when a customer is very active, falls into consideration twice. However, by looking at other features, where the time windows go over many days, the minimal loss of precision is not that important. For example when there is a window of 180 days and slices with a length of 1 day, the global window would comprises 180 days plus some hours of the first day. This additional events should not affect the result significantly and hence this method is plausible for such feature types.

4.3.4 Comparison of the Methods

As a final step, the previously investigated methods will be compared in different areas. Subsequently, it will be decided which method will be used later. Table 4.1 provides an overview of the different methods along with the applied operations. At a first glance, the sliding window variant has a promising complexity for storage as well as the addition and

Method	Storage	Adding An Element
Sliding Windows	Current Aggregated Value	Incrementally
No Window Assigner	Each Element	Incrementally
Window Slices Evicting Single Elements	Each Elements in Slices Aggregated Values in GW	Incrementally in Slices Not Incrementally in GW
Window Slices Evicting Whole Slices	Aggregated values in Slices Total Aggregate in GW	Incrementally in Slices Not Incrementally in GW

Method	Removing an Element	Parallel Aggregations
Sliding Windows	Not Needed	windowSize / windowSlide
No Window Assigner	Incrementally	1
Window Slices Evicting Single Elements	Not Incrementally in Slice Not Incrementally in GW	1
Window Slices Evicting Whole Slices	Not Needed	1

Table 4.1: Comparison of the Methods

removal of elements. However, it applies a lot of parallel aggregations. Due to the fact that it has to store the current aggregated values and add a new element for each parallel aggregation, the approach seems unfeasible for the present use case.

The method without a window assigner offers good performance for addition and deletion operations. Additionally, it only needs to run one parallel aggregation. The main disadvantage is the need to store each element.

The window slices method, which evicts single outdated events, is generally worse than the method with no window assigner. Thus, it will not be considered in more depth.

The last method is the best in regards of performance. It is able to incrementally add values to the aggregator of the slices and it does not need to remove single values. Hence, it does not need to store each event. However, this method is not 100 % accurate regarding precision.

Now that the methods have been compared the decision is made which method will be picked for the different problems. The sliding window method and the exact window slices method are the worst regarding performance. The second window slice method is not exact. Thus, the method without a window assigner fits best for the problems where the precision of the time interval is important. For problems where only roughly correct precision is needed, the slicing windows method with whole slice eviction is chosen.

4.3.5 Summary of Problems

Besides the two problems tried to solve in the previous subsections, there were two more problems mentioned in the introduction of this section. The first of these considered the handling of late events and how the program should react when a late event arrives. Generally, a late event has to be processed as an event in time and the aggregated values have to be updated as usual. Only if the late event is so late that it is already not relevant anymore it is dropped directly.

The second problem is concerned with the handling of features, where no commutative, incremental process function is available (*i.e.*, count distinct). To solve this problem, two possibilities exist. The first approach is to store all passed values in a set and always add a new value if it does not exist yet. Since every value has to be stored and checked when a new value arrives this approach is very inefficient when there are a lot of values. The second possibility is to only approximate the exact value. To do that a HyperLogLog algorithm can be used [5], which is an algorithm that allows to incrementally process the incoming elements. The HyperLogLog estimates the number of different values in a proof of work kind. It stores the hash of each value and the longest chain of leading zeros in the hashed values gives an estimation how many values are approximately present.

As a solution to all problems mentioned in the introduction has been given, a short overview of these problems along with their solutions is provided.

RQ1 How can a set of windows efficiently cover time intervals, when their start and end time is not known exactly in advance?

Answer: By applying one of the investigated methods, which are compared in table 4.1.

RQ2 How can it be ensured that only relevant data is considered for the calculations?

Answer: By using event timers and an according evictor.

RQ3 How will late events be handled?

Answer: Late events will be handled as events in time when they are still relevant, otherwise they are directly deleted.

RQ4 How are features handled, where no commutative and incremental process function is available?

Answer: By using the HyperLogLog algorithm, which estimates the count distinct and can be calculated incrementally.

4.4 Data Reclassification

The project part on data reclassification is about removing authorisations or transactions from individual windows and including them in others. This is done because some card uses, which were originally marked as non-fraud, are subsequently marked as fraud. The difficulty is manifold. On the one hand, numerically stable operations must be found which allow values to be removed from windows and added to others. On the other hand, the data must be adjusted only in certain windows and in some cases it is no longer of interest. It is also necessary to stay efficient in terms of time and space complexity and thus find a way not to store all events of the last half year (worst case). Due to the challenges presented in the previous sections, the given time was unfortunately not sufficient to cover this part in all its depth. It was therefore not possible to devise and implement a presentable solution that would have met the requirements. Accordingly, this part was postponed to future work. This circumstance is also mentioned again in chapter 7.

Chapter 5

Implementation

This Chapter deals with the details of the implementation of the test data generation, as well as the data aggregation with Apache Flink. In a first part, the functionality of the Red Black Tree is examined. Then, the Gen and Temporal Event (TE) monad are introduced in Section 5.1.2. Afterwards, the loading of the statistical models into the tree structures 5.1.3, as well as the poisson point processes 5.1.4 and the enabling of determinism 5.1.5 is discussed in detail. Section 5.1, dedicated to test data generation, is rounded off with an explanation of the TE programs used and how out-of-orderness of the events was achieved. In a second part, the data aggregations in Flink are discussed. It first focuses on the basic structure of Flink programs, continues with the creation of windowing slices and describes the aggregation of values in the aforementioned slices.

5.1 Test Data Generators

The test data generation is an important and interesting part of this work. Without reliable and statistically correct distributed data the system load cannot be checked thoroughly. Due to performance requirements, this part contains some stumbling blocks which will be discussed in the following Sections. Furthermore, determinism in the generation of test data facilitates repeatability of stream processing experiments involving big amounts of data. This is especially important for regression tests against code changes, changes in timeliness of data or changes in configuration for optimization of runtime behavior. Therefore, a subsection will be dedicated to explain how this goal was achieved.

5.1.1 Augmented Red Black Tree

This Section discusses the implemented augmented Red Black Tree [7] for storing statistically distributed objects. The basic structure of the individual nodes is described first. Afterwards, the functions that make up the tree are examined in more detail. These are the search for keys and weights, the insertion and deletion of nodes with automatic re-balancing as well as range queries.

Basic Structure

Each node in the tree is set up to contain two pointers to its child nodes, or `null` if no child node exists. In addition, each node contains an object of type `Color`, which has the available subtypes `Red` or `Black` that denote the color. Since the tree should consist of a triplet with key (A), value (V), and weight (W), each node also needs these attributes. They are intentionally implemented as generics, such that every conceivable type can be used. It is important to note that the keys must be sortable (*i.e.*, implement `Orderable[A]`) and the weights must be numeric (*i.e.*, implement `Numeric[W]`). The structure of such a node therefore looks like shown in Figure 5.1. Furthermore, the two calculated attributes `aggregatedWeight` and `height` should be mentioned. The aggregated weights will play an important role later in the prefix search, while the height is very interesting for testing to ensure that the tree meets the requirements for node depth ($\max. 2 * \log_2(n + 2) - 2$). These two attributes are intentionally chosen as *val* and not as a function. This has the consequence that the values are guaranteed to be calculated only once when the node is instantiated and are unchangeable afterwards. When a new node is added and a re-balancing takes place, only the nodes to be touched have to be recalculated and not recursively the whole tree. Furthermore, a function in this area would recursively calculate the entire tree for each of the aggregated values used, which would no longer meet the requirement for logarithmic complexity.

```

1  abstract sealed class Color
2  case object Red extends Color
3  case object Black extends Color
4
5  case class RBT[A,V,W](
6    color: Color,
7    key: A, value: V, weight: W,
8    left: RBT[A,V,W], right: RBT[A,V,W]
9  )(implicit aOrderable: Orderable[A], wNumeric: Numeric[W]) {
10
11    val aggregatedWeight: W = {
12      var sum = wNumeric.zero
13      if(left != null) sum = wNumeric.plus(sum, left.aggregatedWeight)
14      sum = wNumeric.plus(sum, weight)
15      if(right != null) sum = wNumeric.plus(sum, right.aggregatedWeight)
16      sum
17    }
18    val height: Int = {
19      if (left == null && right == null) 1
20      else if (left == null) right.height + 1
21      else if (right == null) left.height + 1
22      else math.max(left.height, right.height) + 1
23    }
24  }

```

Listing 5.1: Basic Structure of a RBT Node

Addition

In order to add new nodes, an automatic re-balancing must be performed. This requires the use of the five different cases of the Red Black Tree algorithm [30]. At this point, it is not intended to explain each of the cases in detail, as they are already formulated precisely in various other sources. Instead, in 5.2 a single case is taken out to explain the systematics behind the function and the pattern matching used.

First, an add operation calls the function `balanceAdd` 5.2, which recursively arranges the node in the tree based on whether the key to be inserted is greater or smaller than the current node key. After that, a left or right balance is executed and the `balanceAdd` function is called again for the next level. This creates a new RBT object which only needs to adapt one child node. The unaffected child node is taken over unchanged. Finally, the `blacken` function recolors the root node of the tree black to adhere to the Red Black Tree rules.

```

1 def add(tree: RBT[A,V,W], g: A, vg: V, wg: W): RBT[A,V,W] = {
2   def balancedAdd(t: RBT[A,V,W]): RBT[A,V,W] = {
3     if (t == null) RBT(Red, g, vg, wg, null, null)
4     else if (aOrderable(g) < t.key) balanceLeft(t.color, t.key,
5         t.value, t.weight, balancedAdd(t.left), t.right)
6     else if (aOrderable(g) > t.key) balanceRight(t.color, t.key,
7         t.value, t.weight, t.left, balancedAdd(t.right))
8     else t
9   }
10  }
11  blacken(balancedAdd(tree))
12 }
```

Listing 5.2: Addition of a Node

The balancing functions are divided between a function for the right and left side. As an example, only the code for a left-uncle-red case (left node) has been selected here. This case is as shown in Figure 5.1 and depicts a scenario where a new node shall be added as a left child, while the parent and its sibling node (hereafter called uncle) are both red. If this is the case, the colors of the father and uncle must be recolored to black and the root node to red. The newly added node is also highlighted in red. After that, the root node is recursively balanced to check if the recoloring had an impact on the next higher level.

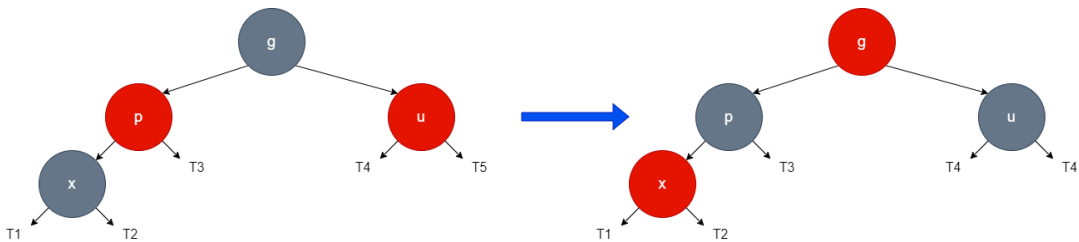


Figure 5.1: Visualization of a Left Uncle Red Case

Listing 5.3 uses Scala pattern matching for these complex operations. Here the tree nodes `l,r` passed into the function can be checked for a certain pattern. This is ideal for distinguishing cases, because it is easy to check if for instance, the root node is black and its two children (uncle and father nodes) are red. The individual values can be given parameter names, which can then be reused for the return of the new object. Important to mention for this example is the naming of the individual values. For this purpose, ***g*** was chosen for the grandfather node, ***p*** for the parent node, ***u*** for the uncle node and ***x*** for the new node.

```

1  def balanceLeft(c: Color, g: A, vg: V, wg: W, l: RBT[A,V,W], r:
    RBT[A,V,W]): RBT[A,V,W] = (c, l, r) match {
2      case
3          (
4              Black,
5              RBT(Red, p, vp, wp,
6                  RBT(Red, x, vx, wx, left_x, right_x),
7                      right_p
8                  ),
9              RBT(Red, u, vu, wu, left_u, right_u)
10             )
11         =>
12             RBT(Red, g, vg, wg,
13                 RBT(Black, p, vp, wp,
14                     RBT(Red, x, vx, wx, left_x, right_x),
15                         right_p
16                     ),
17                 RBT(Black, u, vu, wu, left_u, right_u)
18             )
19     }

```

Listing 5.3: Left Balancing of an Uncle Case Node

Deletion

For the deletion of nodes, the algorithm presented by VirtusLab in [31] was taken as a starting point. It is based on the deletion algorithm for Red Black Trees by Stefan Kahrs [32] who implemented a Haskell version of Okasaki's Red Black Tree [33]. This algorithm was then adapted to the proposed augmented Red Black Tree. In order to give a comprehensible insight into the algorithm, the individual functions and their contents were not listed in detail. Instead, Listing 5.4 only shows the signature and explains the use of the function.

First, a tree and a key to be deleted must be passed to the function. After that, line 14-16 check whether the key to be deleted is located in the left, right or current node and then triggers the appropriate action. A distinction is made between `delLeft`, `delRight` and `append`. These functions are implemented recursively and allow elegant traversing of the

tree and deleting with $O(\log(n))$ complexity. The two `delLeft` and `delRight` functions are called when a value is less or greater than the current node key. The `append` function is called when the key to be deleted is found and the child nodes need to be reassembled. Joining after deleting is enabled by the `balLeft` and `balRight` functions, which each call the `balance` function (with their values based on the general algorithm steps of red and black tree deletion). The last method listed is the `subl` help function, which copies a node and changes its color from black to red. This function is needed for re-balancing.

```

1  def del(tree: RBT[A,V,W], k: A): RBT[A,V,W] =
2  if (tree eq null) null
3  else {
4      def balance(g: A, gv: V, gw: W, tl: RBT[A,V,W], tr: RBT[A,V,W])
5      def subl(t: RBT[A,V,W])
6      def balLeft(x: A, xv: V, xw: W, tl: RBT[A,V,W], tr: RBT[A,V,W])
7      def balRight(x: A, xv: V, xw: W, tl: RBT[A,V,W], tr: RBT[A,V,W])
8      def delLeft
9      def delRight
10     def append(tl: RBT[A,V,W], tr: RBT[A,V,W]): RBT[A,V,W]
11
12     if (k < tree.key) delLeft
13     else if (k > tree.key) delRight
14     else append(tree.left, tree.right)
15 }
16 }
```

Listing 5.4: Deletion From an RBT

Range Queries

A core requirement of the data structure is the ability to evaluate range queries. This should, for example, enable to filter out all nodes of a certain country or other grouping within the key structure. The difficulty here is that the result of such a range query should again be a tree in order to be able to carry out tree operations in $O(\log(n))$. The difficulty which arises here is shown in Figure 5.2. Individual values that fall within the desired range can end up on the other "side" of a tree. These nodes must not be ignored. In addition, there is the problem of merging the individual nodes into a correct red-black tree. In the following lines it is first discussed how to filter out the individual nodes and then a look is taken at the solution to merge potential sub-trees. The core of the implementation consists of the function `doRange`, which is shown in Listing 5.5. The function uses recursion and has the following steps. At the beginning of the function it is checked if a leaf node or empty tree was returned. If this is the case, `null` is returned. Otherwise it is checked if the current key is smaller than the minimum searched value. If this is the case, the function continues with the right child node. Otherwise the next condition checks if the current key is greater than the maximum that is searched. If this is the case, the function continues with the left child node. These two steps are essential to get to a point from which to start returning nodes.

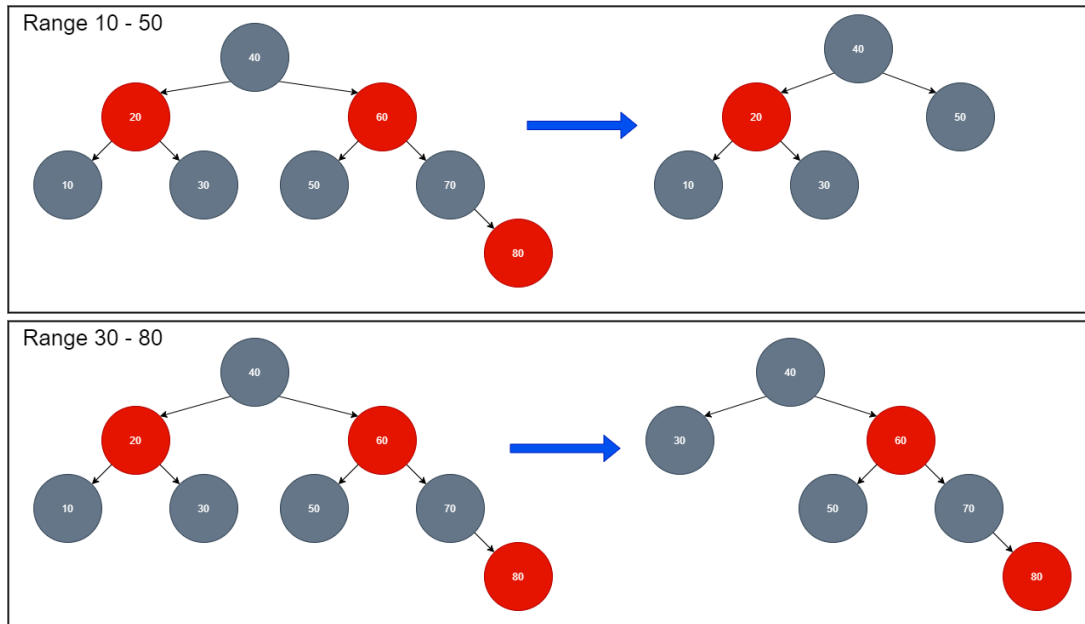


Figure 5.2: Visualization of the Problem When Using Range Queries

Afterwards, the two functions `doFrom` and `doUntil` are used to create a new left and right subtree respectively. In these two functions all nodes are filtered out which are either too small or too large for the searched range. For the `doFrom` function, the left subtree of the previously elaborated node is used, for the `doUntil` function the right subtree. In the best case the tree is returned without changes. Otherwise, child nodes are moved from left to right (or vice versa). This depends on whether one of the nodes no longer fits into the desired range and must therefore be omitted or re-balanced.

```

1 private[this] def doRangeImpl(tree: RBT[A, V, W], from: A, until:
  A): RBT[A, V, W] = {
2   if (tree eq null) return null
3   if (aOrderable(tree.key) < from) return doRangeImpl(tree.right,
    from, until)
4   if (aOrderable(until) <= tree.key) return doRangeImpl(tree.left,
    from, until)
5   val newLeft = doFromImpl(tree.left, from)
6   val newRight = doUntilImpl(tree.right, until)
7
8   if ((newLeft eq tree.left) && (newRight eq tree.right)) tree
9   else if (newLeft eq null) addImpl(newRight, tree.key,
    tree.value, tree.weight)
10  else if (newRight eq null) addImpl(newLeft, tree.key,
    tree.value, tree.weight)
11  else rebalance(tree, newLeft, newRight)
12 }

```

Listing 5.5: Core Method of the Range Queries

The function which facilitates the re-balancing (*cf.* Listing 5.6) is based on the implementation of the Red Black Tree of Scala library [34] itself. It consists of two steps. First, it is checked if the tree is already on the same level. If this is the case, the left and right sub-tree can be combined simply via the root node. If one of the two is bigger than the other, a zipping must take place. The first check is which of the two subtrees is smaller with regard to depth. The smaller of the two is then sorted into a list node by node. This list is then taken and each node is merged into the larger subtree. The merge operation starts by taking the largest value (head) of the list as the starting value for the left or right child of the root node. This is done to determine where the next elements have to be inserted. Then the smallest value is taken from the list and inserted into the tree.

```

1  private[this] def rebalance(tree: RBT[A, V, W], newLeft: RBT[A, V,
    W], newRight: RBT[A, V, W]) = {
2
3      val blkNewLeft = blacken(newLeft)
4      val blkNewRight = blacken(newRight)
5      val (zipper, levelled, leftMost, smallerDepth) =
        compareDepth(blkNewLeft, blkNewRight)
6
7      if (levelled) {
8          RBT(Black, tree.key, tree.value, tree.weight, blkNewLeft,
            blkNewRight)
9      } else {
10         val zipFrom = findDepth(zipper, smallerDepth)
11         val union = if (leftMost) {
12             RBT(Red, tree.key, tree.value, tree.weight, blkNewLeft,
                zipFrom.head)
13         } else {
14             RBT(Red, tree.key, tree.value, tree.weight, zipFrom.head,
                blkNewRight)
15         }
16         val zippedTree = NList.foldLeft(zipFrom.tail, union: RBT[A, V,
            W]) { (tree, node) =>
17             if (leftMost)
18                 balanceLeft(node.color, node.key, node.value, node.weight,
                    tree, node.right)
19             else
20                 balanceRight(node.color, node.key, node.value,
                    node.weight, node.left, tree)
21         }
22         zippedTree
23     }
24 }

```

Listing 5.6: Rebalancing of Range Queries

Key Search

The algorithm, which is shown in Listing 5.7 allows to find a specific key in the tree and is similar to a normal search in a BST. This means that the tree is traversed recursively from the root downwards until the node is reached whose value matches the searched key. In order to enable traversing and to decide whether to continue the search in the left or right sub-tree, a check is made in the active node whether the value searched for is greater or smaller than the current node key.

```

1 private def getNodeByKeyImpl( tree: RBT[A,V,W] , key: A): RBT[A,V,W] =
    {
2     if (tree.key == key) tree
3     else if (tree.left != null && aOrderable(key) < tree.key)
        getNodeByKeyImpl( tree.left , key)
4     else if (tree.right != null && aOrderable(key) > tree.key)
        getNodeByKeyImpl( tree.right , key)
5     else null
6 }

```

Listing 5.7: Search For Key Algorithm

Prefix Search on Aggregated Weights

Another core requirement of the data structure is the possibility to specify the relative probabilities of individual nodes and to be able to query their data efficiently in an uniform structure. The algorithm which makes this possible is the prefix search presented in this paragraph, which operates on the aggregated weights of the nodes.

In a first step (*cf.* Listing 5.8) it is checked if the prefix is within the range of the current node. For this the total weight minus the weight of the node itself is calculated. If the searched prefix is in this range, the value is returned. If not, the left child node is checked if it is not empty and if the aggregated weight of the child node is greater or equal to the prefix. If this is the case, the next node to check is the left child and the recursive algorithm starts again. If none of this is true, the only remaining option is to go to the right child node. Here the left child node is first checked to see if it is empty. If not, one has to go to the right node and a new prefix has to be calculated by taking the existing prefix minus the aggregated weight of the left child node. This must be done as otherwise the range of the values is not consistent. That means, if the total weight is 60 (with the weight of the root node being 12), in the left child node 18, and in the right child node 30, then the actual range for the left node is [0-18], for the right node [18-48] and for the root node [48-60]. Thus, if one went to the right without subtracting the aggregated weight of the left child node, then one would operate in a range of [0-30], but not in a range of [18-48]. A special case is when the left child node is empty. In this case, zero aggregated weight is assumed for this node and therefore nothing is deducted from the prefix.

```

1  def getNodeWithPrefixImpl(treeIn: RBT[A, V, W], prefix: W): RBT[A,
    V, W] = {
2      if (wNumeric.gteq(prefix,
        wNumeric.minus(treeIn.aggregatedWeight, treeIn.weight)))
            treeIn
3      else if (treeIn.left != null &&
        wNumeric.gt(treeIn.left.aggregatedWeight, prefix)){
4          getNodeWithPrefixImpl(treeIn.left, prefix)
5      } else {
6          if (treeIn.right != null) {
7              if (treeIn.left != null) {
8                  getNodeWithPrefixImpl(treeIn.right,
                    wNumeric.minus(prefix,
                        treeIn.left.aggregatedWeight))
9              } else {
10                 getNodeWithPrefixImpl(treeIn.right, prefix)
11             }
12         } else {
13             treeIn
14         }
15     }
16 }

```

Listing 5.8: Search For Prefix Algorithm

In the previous sections, the implementation of the augmented Red Black Tree was discussed in detail. It showed the implementation of such an immutable tree in pure functional style, which fulfills all requirements and can still operate in $O(\log(n))$. An in-depth evaluation of the individual operations can be found in Chapter 6. Monads are another critical element of the test data generation. For this project, the Gen monad and the Temporal Event (TE) monad are of particular importance. Therefore, the next Section is dedicated to these two elements.

5.1.2 Gen and TE Monad

For the creation of data, not only the effective generation (Gen) but also the placing of the generated events in a temporal and causal sequence is required. Therefore, it must be possible to delay a certain action for some time without blocking the other concurrent processes. For this purpose the Temporal Event (TE) monad was elaborated. It was already mentioned and described in Chapter 4. In the following Subsection, the Gen monad is roughly described and its functions and benefits are explained. After that, the TE Monad and its implementation is shown in more detail.

Gen Monad

The Gen monad is contained in Scalacheck [35], which is a test framework for Scala. The monad should help to generate random data for recurring test cases. It will be extensively used in Section 5.1.6 in conjunction with the TE monad to generate the actual test data. Some examples of the functionalities used are listed in the following paragraphs. The examples refer to the code snippets listed in Listing 5.9.

In example 1, a model that will yield two random numbers on execution is defined. `Gen.choose(-180, 180)` models a uniform distribution variable in int on an open interval $[-180, 180]$, `Gen.choose(-90, 90)` analogously for interval $[-90, 90]$. The `val randomCoordinates` models a distribution of tuples composed of the two independent distribution variables longitude and latitude mentioned before.

Example 2 models the generation of an integer on an open interval $[0, 100]$, which is then used as input for the next random drawing of a number. Finally, the function `Gen.const()` can be used to lift a value into the monad so that it can be used in this context. As output value of the second example a tuple with three attributes can be seen, which consists of the previously calculated x,y and z. The line `y <- Gen.choose(2*x, 200)` is also an example for a dependent distribution.

Example 3 shows the frequency function, which makes it possible to get evenly distributed values from a given list. Another possibility of frequency is to pass tuples consisting of a weight and the value as parameters, where the values are then drawn based on the weights. In this example, "Foo" would occur with a probability of 10%, while a "Bar" would be delivered in 90% of the cases.

Example 4 shows a model to create a list with n elements, where the elements of the list may depend on another Gen (in this case randomNumbers).

Example 5 describes the application of arbitrary, which allows to lift arbitrary values and calculations into the monad and execute them there. Here the conversion from a timestamp in milliseconds to the day of the week is shown.

Example 6 shows the sampling of a distribution variable (randomCoordinates). This means that a concrete value is generated but with respect to a given distribution. It also shows the difference between `sample` and `pureApply`. Every time sample is called, a new random seed is created in the background using the `Random` object. With `pureApply`, however, an existing seed object must be provided. This is important to make the whole process deterministic and is explained in detail in Section 5.1.5.

```

1 //Example 1
2 val randomCoordinates : Gen[(Int,Int)] = for {
3   longitude <- Gen.choose(-180,180)
4   latitude <- Gen.choose(-90,90)
5 } yield (longitude,latitude)
6
7 //Example 2
8 val randomNumbers : Gen[(Int,Int,Int)] = for {
9   x <- Gen.choose(0,100)
10  y <- Gen.choose(2*x,200)
11  z <- Gen.const(10)
12 } yield (x,y,z)
13
14 //Example 3
15 val drawOneWithFrequency : Gen[String] =
16   Gen.frequency((10,"Foo"),(90,"Bar"))
17
18 //Example 4
19 val listOfIntTuples : Gen[List[(Int,Int,Int)]] =
20   Gen.listOfN(10,randomNumbers)
21
22 //Example 5
23 def getDayOfWeekFromMillies(millis: Long) = Arbitrary({
24   val instant = Instant.ofEpochMilli(millis)
25   val zonedDateTimeUtc = ZonedDateTime.ofInstant(instant,
26     ZoneId.of("UTC"))
27   zonedDateTimeUtc.getDayOfYear()
28 }).arbitrary
29
30 //Example 6
31 val result : Option[(Int,Int)] = randomCoordinates.sample
32 val resultPure : (Int,Int) =
33   randomCoordinates.pureApply(parameters, seed)

```

Listing 5.9: Gen Monad Examples

TE Monad

The Temporal Event (short TE[T] or TE) monad can be described as one of the core elements of test data generation. It is used to model time sequences and enables concurrency of processes. The implementation of the monad is based on FreeMonads, which were explained in Section 4.2.2. The approach to describe the implemented functionality depends on the individual elements of the ADT. Thus it will be explained by describing the operations used in the compiler. As a reminder, the ADT models a finite set of impure effects with regard to model time and progression. Thereof, sampling of distribution variables, emission of streaming events, concurrency and synchronization as well as global state, and general side effects. Additionally, the TE monad also adds causality, *i.e.*, strict

order of the execution of aforementioned ADT side effects. The explanations will follow the listing from Section 4.2.2. First, however, some general facts have to be explained. This concerns on the one hand the IO monad [36] used for isolating side effects and on the other hand the Sync- and ContextTimer, which control the time model.

Basically it can be said that the IO monad [36] is used to encode side effects as pure values. Accordingly, IO values are pure and unchangeable values and therefore maintain the referential transparency of the functional programming while allowing side effects to be added to the mathematical model that monads represent. Furthermore, IO uses a technique called trampolining [37], in order to convert functional recursion into iteration to achieve stack neutrality.

The actual time model is controlled by SyncTimer, which is an implementation of the ContextTimer interface. The function `sleepModelTime` can be used to adjust whether delays should be waited for, or whether the delay should only be taken as a value and run through without an actual delay. This is achieved by replacing the function seen in Listing 5.10 with an empty function.

```

1 protected def sleepModelTime(now: Long, until: Long) = {
2     if (now < until) Thread.sleep(until - now)
3 }

```

Listing 5.10: Sleep Model Time

The SyncTimer also acts as queue for delayed operations. This means that the SyncTimer contains a priority queue which stores the individual continuations and their timestamps. This means that when a new delay operation is registered, it is written into the queue together with the time of its future activation as shown in Listing 5.11. When the stored time is reached, the associated continuation function is called. One of the most important elements of the SyncTimer is the line `val tick = (when, currentTick)`. With `when` and `currentTick` a total order of executed effects can be built, where `when` is the model time of the effect and `currentTick` is a monotonously growing time-stamp of when the delay/sleep operation was registered. Due to this order, it is possible to get an order in which effects are applied and therefore it is possible to enable determinism.

```

1 override def sleep(duration: FiniteDuration): IO[Unit] = {
2     clock.realTime(MILLISECONDS).flatMap(now => {
3         val when = now + duration.toMillis
4         IO.async[Unit]((cb: callBack) => {
5             val tick = (when, currentTick)
6             currentTick += 1
7             queue.enqueue((tick, cb))
8             ()
9         })
10    })
11 }

```

Listing 5.11: Enqueueing Continuations With Sleep

The `TE[T]` compiler implements the abstract ADT operations, which are explained on the following lines. The cases are ordered by topic. First, the cases are explained in terms of model and time progression, followed by the sampling of distribution variables and seed setting, then comes the emission of streaming events, followed by global state management and the handling of general side effects. Finally, concurrency is explained.

- **Pure** The most simple ADT case is pure. It allows to lift a value into the monad and use it in subsequent calculations. If a parameter `t` of type `T` is passed to the function, a `TE[T]` is returned which contains the parameter value.

```
1 def pure[T](t: T): TE[T] = Free.pure(t)
```

Listing 5.12: Implementation of Pure

- **Now** This ADT case is used to get the current model time, which takes its value from the implicitly given `ContextTimer` (`timer`).

```
1 case Now() => {
2     timer.clock.realTime(MILLISECONDS).asInstanceOf[IO[T]]
3 }
```

Listing 5.13: Implementation of Now

- **Delay** The delay ADT case uses the sleep method, which is delegated by IO to the previously mentioned `SyncTimer`. A continuation is registered and waits until it can be executed.

```
1 case Delay(millis) => {
2     IO.sleep(millis.milliseconds).asInstanceOf[IO[T]]
3 }
```

Listing 5.14: Implementation of Delay

- **Generate** The purpose of the Generate ADT case is reported in more detail in Section 5.1.5. For now, it suffices to say that Generate is used to execute a Gen monad value, `gen`, and samples the distribution variable `gen` by means of the deterministic `incrementSeed`.

```
1 case Generate(gen) => {
2     IO {
3         gen.pureApply(genParameters, incrementSeed)
4     }
5 }
```

Listing 5.15: Implementation of Gen

- **SetSeed** Why the seed object is essential for enabling determinism in random data generation is discussed in more detail in Section 5.1.5. At this point, it is to be mentioned that this ADT case is responsible for setting an initial seed for the generation of pseudo random numbers in the compiler. Ideally, this is the first command called during initialization.

```

1 case SetSeed(s) => {
2     IO {
3         seed = Seed.apply(s.toLong)
4     }.asInstanceOf[IO[T]]
5 }

```

Listing 5.16: Implementation of SetSeed

- **Emit** The Emit ADT case determines the current model time and then emits the value `t` to a Flink streaming context (selection of the target Flink `DataStream[T]` is not shown here).

```

1 case Emit(s, t) => {
2     timer.clock.realTime(MILLISECONDS)
3     .flatMap(now => {
4         IO {
5             ctx.collectWithTimestamp(t.asInstanceOf[OUT], now)
6         }
7     })
8 }

```

Listing 5.17: Implementation of Emit (Simplified)

- **PutToStateBag** The basic problem that is solved by modeling a global state bag is, that there was a need to find a central location for the created data structures. Therefore, a simple static object was created first, which contained the trees as lazy vals. When the tree was needed for the first time, it was created on the fly. This worked well until determinism was needed and the creation of structures had to be passed to the TE monad and executed via the `gen` command of the ADT and not via the `sample` function of the Gen monad. The solution to the problem was to have the needed objects created at the beginning of the data generation and write them into the state bag via the `PutToStateBag` ADT case. The state bag is represented by a map which takes a symbol as key (unique string) and an object of type `Any` as value. First of all, as shown in Listing 5.18, it is checked whether such an object type already exists with the supplied symbol. If so, this entry is deleted first and the new one is added. If nothing already exists, the value is written to the bag. Furthermore, it is important to note that `PutToStateBag` is part of the total time order modeled by `TE[T]` and the ADT.

```

1  case PutToStateBag(t, element) => {
2      IO {
3          val e = stateBag.get(t);
4          e match {
5              case Some(ob) => {
6                  stateBag = stateBag.-(t)
7                  stateBag = stateBag.+((t, element))
8              }
9              case _ => {
10                 stateBag = stateBag.+((t, element))
11             }
12         }
13         stateBag.get(t).get
14     }.asInstanceOf[IO[T]]
15 }

```

Listing 5.18: Implementation of PutToStateBag

- **GetFromStateBag** Of course it must also be possible to get values from the state bag. For this purpose the GetFromStateBag ADT case was implemented. As seen in Listing 5.19, it is first checked whether an object with the desired symbol and type is available. If so, this object is casted to the target type and returned. Otherwise None is returned.

```

1  case GetFromStateBag(t) => {
2      stateBag.get(t) match {
3          case Some(ob) => {
4              IO {
5                  Some(ob.asInstanceOf[T])
6              }
7          }.asInstanceOf[IO[T]]
8          case _ => IO{None}.asInstanceOf[IO[T]]
9      }
10 }

```

Listing 5.19: Implementation of GetFromStateBag

- **Effect** helps to lift an arbitrary effect function **effect**, which contains side effects, into the monad in order to not violate the conditions of the mathematical model. An example could be changing the preferred merchants of a customer, where the merchant structure has to be adjusted due to changing preferences during runtime.

```

1  case Effect(effect) => {
2      IO {effect()}
3  }

```

Listing 5.20: Implementation of Effect

- **Concurrent** The **Concurrent** ADT case is a feature that allows TE programs, as the name suggests, to be concurrent. As return value the user gets a fiber, which is a kind of lightweight thread. A fiber represents a pure result of an asynchronous datatype like IO and can be either aborted or joined (synchronized == wait for completion, result extraction) to other fibers. This is achieved with the two functions `cancel` or `await`, which are shown in Listing 5.21.

```

1  case Concurrent(run) => {
2    val runIO = run.foldMap(this)
3    runIO.start
4    .map(ioFiber =>
5      new TemporalFiber[Any] {
6        override def cancel: TE[Unit] = {
7          val res: TE[Unit] = liftF [TemporalEventA,
8            Unit] (Suspend(ioFiber.cancel))
9          res
10         }
11       override def await: TE[Any] = {
12         val res: TE[Any] = liftF [TemporalEventA,
13           Any] (Suspend(ioFiber.join))
14         res
15       }
16     }.asInstanceOf[T]
17   }

```

Listing 5.21: Implementation of Concurrent

At this point it should be mentioned that the composability of monads was deliberately omitted and currently different monads / concerns are combined in one ADT. These are the time modelling and progression, the sampling of distribution variables, the emission of streaming events, concurrency and synchronization as well as global state and the handling of general side effects. [38] suggests to compose these monads and thus, allowing to also compose compilers thereof. Still, this approach has not yet been implemented in this project due to its overhead. The presented solution can nevertheless be refactored accordingly should the need arise. In the previous lines, the TE monad was explained in more detail and its functionality described. What is now needed to successfully create a program that can generate events based on statistical distributions is the creation and filling of the data structure that was shown in Section 5.1.1.

5.1.3 Load Statistical Data Into The Structure

One question that certainly remains is how the statistical distributions talked about earlier can be loaded into the elaborated tree structure. There are two different ways to achieve this goal. On the one hand, distributions can be estimated from productive data and

imported from a comma-separated values (CSV) file. This method is shown in Listing 5.22. For legal reasons, however, the current data or its structure cannot be shown here. Instead, placeholder column names are used to show the principle. As a second option, entities created from the productive distributions (such as merchants) can be stored in a tree. This is described in detail in Listing 5.23.

Loading data from a CSV file is quite self-explanatory. To add an element to the tree, a tuple of the corresponding type must be created. This tuple consists of three different values. Namely the key of type A, the value of type V and the weight of type W. If a distribution of different merchants (mccs) per region is to be loaded into a structure, an ordered key must first be defined as a case class. In the example shown in Listing 5.22, this is represented by the `DistinctMerchantsByCountryTreeKey`, which takes an `mcc` and the name of a region as input. This key will be used as the first element of the node tuple. The actual payload (merchant itself) of the new node must be placed second in the tuple. Finally, the count supplied in the csv, which acts as a weight, is stored in the third part of the tuple. In the end, an RBT tuple $[A,V,W]$ is constructed. After creating the tree, *i.e.*, a prefix search can be performed to sample a specific `mcc` and region based on the distributions observed in the productive systems.

```

1  def instantiate() :
    RBT[DistinctMerchantsByCountryTreeKey, CountryMcc, Double] = {
2      val distinctMerchants =
        CsvReader.getListFromFile(Source.fromResource(path))
3      .map(line => {(
4          DistinctMerchantsByCountryTreeKey(line(0), line(1))
5          , CountryMcc(line(0), line(1))
6          , tryToDouble(line(2)).getOrElse(0.0)
7      })}
8  )
9
10     for {
11       tree <- RBT.generateTree(distinctMerchants)
12       state <- putToStateBag('DMBCT, new
        DistingMerchantsByCountryTree(tree))
13     } yield state
14   }
15 }
```

Listing 5.22: Loading a Tree From a CSV

Creating entities that have dependencies to existing probability trees is not much different. In the basic principle again, either a list of nodes must be created or nodes are created incrementally and added to the tree. For the creation of the merchants the latter was chosen, as it does not make sense in this case to first save everything into a list and then to merge it into a tree.

In the following example, the merchant structure is set up. As before, the starting point is the `instantiate` function. The number of merchants to be created and the symbol as which

the structure is finally to be stored in the status bag are specified here. What follows is a recursive creation and storage of the merchants in the target structure. The individual nodes are created in the function `createMerchantTreeNode` of Listing 5.23. This function in turn uses the instantiation function of the merchant itself, which can be found at the bottom of Listing 5.24. This is also the first time a get from the state bag is visible. With the function `State.getDistinctMerchantStructure('DMBCT')` the corresponding structure is retrieved from the state bag and is available for further calculations. This function is syntactical sugar for executing `TemporalEventInstances.getFromStateBag()` and should make the process more readable.

```

1  object MerchantTree{
2    private def createMerchantTreeNode(merchantId: String) :
      TE[( MerchantTreeKey , Merchant , Double)] =
3    for {
4      x <- createMerchant(merchantId)
5      weight <- gen(Gen.choose(0,10000))
6    } yield ( MerchantTreeKey(x.region.alpha3Code ,... ,... ,...) , x,
      weight)
7
8    private def addNodeToTree(maxAmount : Long, id: Long, treeIn :
      Option[RBT[ MerchantTreeKey , Merchant , Double]]) :
      TE[ Option [RBT[ MerchantTreeKey , Merchant , Double ]]] = for {
9      newNode <- createMerchantTreeNode(id.toString)
10     treeOut = treeIn match{
11       case Some(t) => Some((t.add _).tupled(newNode))
12       case None =>
13         RBT.generateTree((newNode._1 ,newNode._2 ,newNode._3))
14     }
15     tree <- createListOfMerchantNodes(maxAmount, id+1,treeOut)
16   } yield tree
17
18   private def createListOfMerchantNodes(maxAmount : Long, id: Long,
19     treeIn : Option[RBT[ MerchantTreeKey , Merchant , Double]]) :
20     TE[ Option [RBT[ MerchantTreeKey , Merchant , Double ]]] = for {
21     merchants <- if (id < maxAmount)
22       addNodeToTree(maxAmount,id ,treeIn) else pure(treeIn)
23   } yield merchants
24
25   def instantiate(amountOfMerchants : Long,symbol:Symbol) :
26     TE[MerchantTree]= {
27     for {
28       tree <- createListOfMerchantNodes(amountOfMerchants,0L,None)
29       state <- putToStateBag('MerchantTree,new MerchantTree(tree))
30     } yield state
31   }
32 }
```

Listing 5.23: Creation of a Merchant Tree

```

1 object Merchant {
2     def createMerchant(merchantId: String) : TE[Merchant] = {
3         for {
4             distinctMerchantsTree <-
5                 State.getDistinctMerchantStructure(DMBCT)
6             selectedRegionAndMerchant <-
7                 distinctMerchantsTree.genRandomWeightedNodeValue()
8             regionTree <- State.getRegionStructure('RegionTree)
9             regionKey =
10                 RegionTreeKey.build(selectedRegionAndMerchant)
11             region <- regionTree.genGetNodeValue(regionKey)
12             ...
13         } yield Merchant(region, ... , ... , ...)
14     }
15 }

```

Listing 5.24: Creation of a Merchant

The previous section described how the data structure can be filled and used to handle statistical distributions. What is still missing is a function for the stochastic delay of operations. This will be discussed in the next Section.

5.1.4 Poisson Point Processes

At this point it is deliberately avoided to offer a mathematical proof or a formula to explain poisson point processes. Instead, an easy to understand introduction to the topic and its benefits for the present project is provided.

[39] describes a poisson point process as a model for a series of discrete and independent events where the average time between events is known but the exact timing is random. The occurrence of an event is therefore independent of the preceding events. As an example of the scope of this thesis, the average occurrence of a credit card transaction is an undisclosed frequency of transaction events per second. However, the range varies between some seconds and could, in extreme cases, even be years (which is very unlikely).

An uniform poisson point process on the time axis is characterized by an average rate of events per time unit. The probability of a specific number of events observed in time intervals of specific events follows a discrete poisson distribution. [40] elaborates that the distance between two consecutive events follows an exponential distribution function parameterized by the rate λ . In order to sample such a time distance, a random variable is applied to the inverted CDF of this exponential distribution. If it is now assumed, for the sake of discussion, that a transaction is made on average every two minutes. Accordingly, a rate parameter λ can be defined which reflects this fact. In this case, this would be $\lambda = 1/120$. The rate parameter is therefore a reflection of the frequency with which a certain event occurs per time unit. What is interesting to answer are questions like:

”What is the probability that a transaction occurs after 100 seconds?”. To answer this question the cumulative distribution function (CDF) for exponential distributions can be used. This function is defined as follows

$$F(x) = 1 - e^{-\lambda * x}$$

The CDF has a characteristic as shown in Figure 5.3. It can now be seen that the probability of a transaction arriving after 120 seconds is approximately 63%. Thus, it can be said that the longer it takes, the greater the probability of the event occurring gets. On the other hand, the probability that no transaction will occur tends to zero the more time passes.

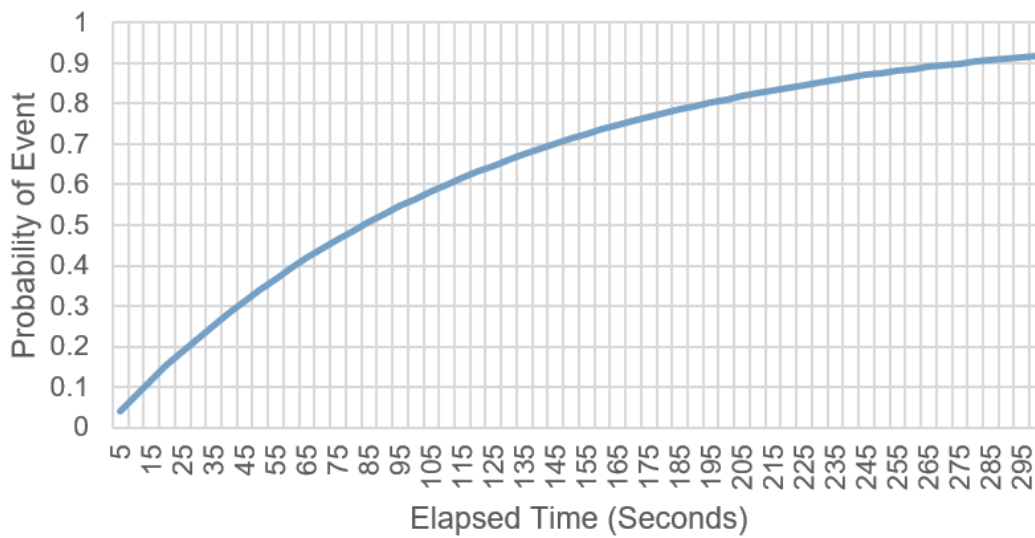


Figure 5.3: Visualization of a CDF of an Exponential Distribution

It should have become obvious by now why the poisson point processes are of immense importance for the random generation of test data. If a pattern is to be generated which must reflect reality as accurately as possible, it is essential to have random intervals. It is not sufficient to have a constant number of transactions every day or that customers change their location or card in the same intervals. Up to now there has been a lot of talk about randomness, but how can it be ensured that the randomness is calculable and repeatable in order to create tests and evaluate the whole system? This question will be discussed in the next Section.

5.1.5 How to Enable Determinism

One of the main objectives of test data generation was to enable determinism. This means that the same initial input should generate the same output. This is especially important for testing the aggregations, because this way different algorithms can be evaluated and it is always clear which final state is expected. It is also important to be able to make statements about the correctness of the random generation.

This goal was achieved with the Scalacheck[35] **Seed** object. This object contains four long values, which amount to 256 bits of entropy and determine the random number. When calling the function `next()`, these values are changed by rotation operations in order to get a new seed. Of course, a start value can also be given during the initial creation of the seed.

The random operations used by the Gen monad all use such a seed object. The problem is the following. In order to get a result from the monad, the function `sample()` can be called (*e.g.*, `Gen.choose(0,10).sample()` to get an integer value between 0 and 10). What happens now in the background is shown in Listing 5.25.

```

1  def sample: Option[T] = doApply(Gen.Parameters.default ,
    Seed.random()).retrieve
2
3  def random(): Seed = apply(scala.util.Random.nextLong)

```

Listing 5.25: Called Functions When Using Sample

The two lines mean that each time `sample()` is executed, a new random seed object is created based on `scala.util.Random.nextLong`. Thus no determinism of the operations is possible. The problem is further complicated by the fact that different entities end up performing concurrent operations, each requiring a random value. The solution to this problem is to create a central location where a seed object is stored. In this paper this location is the compiler needed for the TE monad. The compiler contains a private variable `seed` and the private functions `setInitialSeed` and `incrementSeed`. They can only be used by the compiler itself. The only point of interaction is the setting of an initial seed via the ADT case `SetSeed`. Accordingly, a new ADT case named `Generate(gen : Gen)` had to be created, which gets a Gen as parameter to be executed. If this ADT case is used and passed to the compiler, a simple but clever operation is performed as shown in Listing 5.26.

```

1  IO{
2      gen.pureApply(genParameters , incrementSeed)
3  }

```

Listing 5.26: PureApply That Allows For Determinism

With every usage of the Generate case, a `pureApply` is executed on the passed Gen, with the current seed stored in the compiler. Additionally, the `next()` function is executed on the same seed object to get the subsequent random value. This way, the same output can always be obtained for an initial seed. This forces the user to use the TE monad for all operations that should generate random values. Nevertheless, it has the advantage of allowing determinism, which would otherwise be difficult or almost impossible. At this point it is mentioned again that the `TE[T]` monad guarantees a total order of events. Especially the sampling of Gen objects if and only if the compiler interpreting `TE[T]` programs respects the semantics of the ADT operations. Now that all the elements are in place to write the actual programs for test data generation, the next Section is dedicated to these and explains how they work.

5.1.6 Programs

As the goal of the first part of this thesis was to offer a generalized and simplified possibility for users to develop a coherent model for test data generation without having to deal with the underlying complex mechanisms or optimizations. This includes, for example, modelling the time dimension or providing an efficient data structure for probability distributions. In the end, only single, simple building blocks have to be put together to write down complex issues in a simple and understandable way.

As an example, the creation of customers and their individual processes is described in the following lines (*cf.* Figure 4.6 in Section 4.2.2). As in Listing 5.27, the procedure begins with the `genStructure()` function, which calls the previously explained ADT commands. First, an initial seed is set (*cf.* line 2 in 5.27) and the globally required data structures are generated (*cf.* line 3-7 in 5.27). The instantiate functions of the structure objects also contain the command to write the trees into the state bag. Consequently, after initialization, these structures can be accessed at all points in the code when using the TE monad. As a next step, the program `genCustomers` is called (*cf.* line 9 in 5.27), which generates a certain number of customers and starting their lifecycle. The lifecycle for a customer currently includes the periodic exchange of cards, the exchange of merchants and the use of cards.

```

1  def genStructure(): TE[Unit] = for {
2    _ <- setSeed(10L)
3    _ <- MccToGroupMapping.instantiate('MccToGroupMapping)
4    _ <- MccTree.instantiate('MccTree)
5    _ <- RegionTree.instantiate('RegionTree)
6    _ <- DistinctMerchantsByCountryTree.instantiate(DMBCT)
7    _ <-
      MerchantTree.instantiate(Config.numberOfMerchants, 'MerchantTree)
8    _ <- effect(println("Generated Dependent Structures"))
9    _ <- concurrent(genCustomers(0))
10 } yield ()
11
12 def genCustomers(id: Long): TE[Unit] = for {
13   customer <- Customer.create(id)
14   _ <- concurrent(customer.customerLifetime())
15   _ <- if (id < Config.numberOfCustomers + 1)
       concurrent(genCustomers(id + 1)) else unit
16 } yield ()

```

Listing 5.27: Generation of Datastructures and Customers

As it is now known how the customers are created, a look can be taken at their independent activities, more precisely the use of a card as seen in Listing 5.28. A closer look immediately reveals a pattern. First, the TE program calls the `concurrent` function `spend()` (*cf.* line 5 in 5.28). As an initial step, a delay is generated by using the function `nextPoisonPoint()` (*cf.* line 10 in 5.28). The returned time in milliseconds is then used to

delay the continuation (*cf.* line 11 in 5.28). When the specified time is reached, one of the four currently implemented spending patterns (*cf.* Listing 5.29) is selected based on the probabilities of the selected merchant (*cf.* line 12 in 5.28). This pattern is nothing else than another TE program, which is executed on line 13 and will subsequently emit events based on the specified procedure. Finally, the spend function is called again as long as no cancelFlag is set. Again, it is important to note that this is just a model. The actual execution of the program only happens when it is interpreted by the compiler.

```

1  def customerLifetime(): TE[Unit] =
2    for {
3      _ <- concurrent(addMerchant())
4      _ <- concurrent(addCard())
5      _ <- concurrent(spend())
6    } yield ()
7
8  private def spend(): TE[Unit] =
9    for {
10     nextSpendingTime <-
11       gen(nextPoisonPoint(Config.avgTimeUntilNextCardUsage))
12     _ <- delay(nextSpendingTime.toLong)
13     spendingPattern <- pure(cardUsage())
14     _ <- concurrent(spendingPattern)
15     _ <- if(cancelFlag) unit else spend()
16   } yield ()
17
18  def cardUsage(): TE[Unit] = for {
19    merchant <- merchants.getRandomWeightedNodeValue
20    card <- cards.getRandomWeightedNodeValue
21    pattern <- merchant match {
22      case Some(m) => {
23        for {
24          usageTypes <-
25            m.cardUsagePatterns.getRandomWeightedNodeValue
26          prog <- usageTypes match {
27            case Some(p) => p(card.get, this, m);
28            case None => unit
29          }
30        } yield prog
31      }
32      case None => unit;
33    }
34  } yield pattern

```

Listing 5.28: Start of a Customer Lifetime and Card Usage

In the code snippet shown in Listing 5.29 a *Simple Case* pattern is handled. This means that when a customer pays with a card at a point of sale (online or onsite), an authorisation is generated. Then, after a certain delay, a transaction for the same amount is

automatically initiated to make the booking. In addition to the case just presented, there are other variants mentioned in Section 4.2.2 which have been implemented so far. Of course, other cases can be added when needed.

```

1  def single(card: Card, customer: Customer, merchant: Merchant):
    TE[Unit] = for {
2    authTimeRes <- now
3    authRes<- gen(Authorisation.newAuthorisation(card.cardId, "res",
        "Single", customer, merchant, merchant.amountGen, authTimeRes))
4    authResEmitted <- emit(authRes)
5    tevDelay <- pure(Time.days(1).toMilliseconds)
6    _ <- delay(tevDelay.toLong)
7    trxTimeBkg <- now
8    trxBkg <- gen(Transaction.newTrx(authResEmitted, "bkg",
        "Single", trxTimeBkg))
9    _ <- emit("trx", trxBkg)
10 } yield ()

```

Listing 5.29: Card Usage Pattern, Simple Case

Out-Of-Orderness

It is now possible to generate authorisations and transactions using the monadic structure and emit them as a stream. In the current state, the events are now always emitted at the point in time at which they actually happened. However, this does not really correspond to reality, where, for example, some events can only be transmitted at a later point in time.

For this purpose a `KeyedProcessFunction` from Flink was implemented, which is supposed to delay single events based on some distribution variable for the delay. In the implementation at hand an exponential distribution was chosen for this delay. It is important to note that the `KeyedProcessFunction` can only operate on a keyed stream. The core of the function is relatively simple. For implementation, a new class must be created, which is derived from `KeyedProcessFunction`. In this class, the two functions `processElement` and `onTimer` must be overwritten. The `processElement` function handles all new incoming events for a specific key. In the present case, the events are to be stored in a map state, where the event time plus a stochastic delay is taken as the key. After that a new timer is registered on this time which was specified as key. If this time is reached and the timer is fired, the `onTimer` function is called. In this function the map state is checked and the event with the corresponding timestamp is retrieved and emitted using the original (non-delayed) event time. Thus, it was possible to delay events for a certain time. However, now it can also happen that the target time of delayed events collide or that an event that was initially emitted earlier only arrives after a later one. Both cases must be intercepted.

What has been shown in the previous Section 5.1.6 is the breaking down of complex problems into a few program statements that hide the actual complexity of the problem

itself. As a result, statistically distributed events are generated that can simulate the load and partitioning of a production system in a credit card environment. The evaluation of the implemented system follows in Chapter 6.

5.2 Data Aggregation

In Section 4.3 different concepts were introduced, which solve the problem of having aggregations over all elements in a window. In a final comparison it was decided that the method without a window assigner is the most suitable one when the window has to contain the last 24 hours. Sometimes it is acceptable that the calculations are made over a window where also some events outside of the relevant time range are considered. In this case, the method with tumbling windows slices (*i.e.*, evict whole slices at once) is the most suitable regarding performance. In this Section a possible implementation for both concepts is explained. Afterwards, in 5.2.3 an example is shown which combines the generated test data and the aggregations.

The examples made will be based on the feature "The average amount of the authorisations of a specific customer in the last 24 hours" for both methods. Instead of only searching for the plain averages also the distribution of the amounts will be looked at. These are namely the standard deviation, the skewness and the curtosis. The set of these values combined will be called distribution statistics in the following.

The goal of both implementations is to process an input stream of `Authorisation` events and return a new stream of `DistributionStatisticsOverPeriod` elements. Listing 5.30 shows the composition of such an element. It consists of the distribution statistics along with the key and time window the statistics belong to. The distribution statistics comprises count, mean, standard deviation, skewness and curtosis as shown in Listing 5.31. In general, such an element stores the distribution of the amounts spent by the specified key in a window.

```
1 case class DistributionStatisticsOverPeriod(key: String, statistics:
  DistributionStatistics, window: CustomTimeWindow)
```

Listing 5.30: `DistributionStatisticsOverPeriod`

```
1 case class DistributionStatistics(count: Long, mean: Double,
  standardDeviation: Double, skewness: Double, curtosis: Double)
```

Listing 5.31: `DistributionStatistics`

5.2.1 No Window Assigner

The base function for the method with no window assigner is quite simple and consists only of a single Flink program, hence the input and output is a data stream. The program

is shown in Listing 5.32. Two values are given as input parameter. These are the stream of `Authorisation` elements and the `timeOfRelevancyInMs`. The latter defines the time after when an element is not relevant anymore and hence should be removed. For the present example, the `timeOfRelevancyInMs` is 24 hours.

The output of the function is a stream of `DistributionStatisticsOverPeriod` elements as mentioned in the introduction of Section 5.2.

```

1  def calculateStatisticsNoWindow(authStream:
    DataStream[Authorisation], timeOfRelevancyInMs: Long):
    DataStream[DistributionStatisticsOverPeriod] = {
2      authStream
3      .keyBy(_ . customerId)
4      .process(new
        GetStatisticsOverTimeProcessFunction(timeOfRelevancyInMs))
5  }
```

Listing 5.32: Flink Program With no Window Assigner

First, the input stream is split into a set of keyed streams in line 3. As the goal is to know the statistics of each specific customer, the stream is keyed by the attribute `customerId`. After that the `ProcessFunction` `GetStatisticsOverTimeProcessFunction` is applied, which will process all incoming elements and generate output elements with the type `DistributionStatisticsOverPeriod`. The variable `timeOfRelevancyInMs` is given as parameter, which denotes the time when events have to be removed. The code of the function is shown in Listing 5.33. The function is an extension of a `KeyedProcessFunction`. The types in square brackets in the function header represent the type of the key, the type of the elements in the input stream and the type of the elements in the output stream.

As explained in Section 4.3.2 the state of the function has to store each currently relevant value in a map with the time to remove the specific value as key. Since later on also the key, which split the stream into keyed streams, has to be known, the key of it is stored along with the value in a type called `ValueAndKey`. Besides that, the current aggregated value has to be stored as well. The values are of type `MomentumStatisticsAggregator`, which has the same attributes as the type `DistributionStatistics`. Since the distribution statistics can not be calculated in an incremental way, the momentum statistics are used for this. The state of the function is set up in lines 4 and 5. The initialization of the state itself is simplified with the method `setupState()`. This method initialize an empty Map for the `valueToRemoveState` and null for the `currentAggregatedValueState`.

In lines 8 - 35 the `processElement` function is defined, which is applied each time a new event arrives in the stream. As parameters the new value, the current context of the environment and the collector are passed. As a first step, the `currentAggregatedValueState` is initialized. The new value can then be incrementally added to the value state. This is done with the function `addValueToStat`, which applies the statistical computations. These computations are not in the focus of the project and will not be described here. They are derived from [29]. After aggregating the new value it is returned along with the context information in a `DistributionStatisticsOverPeriod`. Since the aggregated

value is an element of type `MomentumStatisticsAggregator` it has to be transformed into an element of type `DistributionStatistics`. This is done by applying the function `getStatisticalValues`.

As a last step in the process function, a timer is set to the time when the element has to be deleted again. This is equal to *eventTime* + *timeOfRelevancyInMs*. Besides that, the processed element is stored in the state.

```

1  class GetStatisticsOverTimeProcessFunction(timeOfRelevancyInMs:
    Long) extends KeyedProcessFunction[String, Authorisation,
    DistributionStatisticsOverPeriod] {
2
3  // The state that is maintained by this process function
4  lazy val valuesToRemoveState: MapState[Long, ValueAndKey] =
    setupState()
5  lazy val currentAggregatedValuesState:
    ValueState[MomentumStatisticsAggregator] = setupState()
6
7  override def processElement
8  (
9    value: (Authorisation),
10   ctx: KeyedProcessFunction[String, Authorisation,
    DistributionStatisticsOverPeriod]#Context,
11   out: Collector[DistributionStatisticsOverPeriod]): Unit = {
12
13   if(currentAggregatedValuesState.value() == null){
14     currentAggregatedValuesState.update(
15       MomentumStatisticsAggregator(0, 0, 0, 0, 0))
16   }
17
18   val elementValue = value.amount.toDouble
19   currentAggregatedValuesState
20     .update(addValueToStat
21       (currentAggregatedValuesState.value(), elementValue))
22
23   // return the new aggregated values
24   out.collect(DistributionStatisticsOverPeriod(
25     ctx.getCurrentKey,
26     getStatisticValues(currentAggregatedValuesState.value()),
27     CustomTimeWindow(ctx.timestamp() - timeOfRelevancyInMs,
28       ctx.timestamp())))
29
30   // set timer to remove event and store it in the MapState
31   lazy val timeToRemoveElement = ctx.timestamp() + timeOfRelevancyInMs
32   valuesToRemoveState.put(timeToRemoveElement,
33     ValueAndKey(ctx.getCurrentKey, elementValue))
34   ctx.timerService.registerEventTimeTimer(timeToRemoveElement)
35 }

```

```

34
35 // event timer is passed and a value has to be removed
36 override def onTimer
37 (
38     timestamp: Long,
39     ctx: KeyedProcessFunction[String, Authorisation,
40         DistributionStatisticsOverPeriod]#OnTimerContext,
41     out: Collector[DistributionStatisticsOverPeriod]): Unit = {
42     val valueToRemove = valuesToRemoveState.get(timestamp)
43     if (valueToRemove != null) {
44         valuesToRemoveState.remove(timestamp)
45         currentAggregatedValuesState.update(
46             removeValueFromStat(currentAggregatedValuesState.value(),
47                 valueToRemove.value))
48         out.collect(DistributionStatisticsOverPeriod(
49             valueToRemove.key,
50             getStatisticValues(currentAggregatedValuesState.value()),
51             CustomTimeWindow(timestamp - timeOfRelevancyInMs, timestamp)))
52     }
53 }
54 }

```

Listing 5.33: GetStatisticsOverTimeProcessFunction

In order to implement the `KeyedProcessFunction`, `onTimer` also has to be overwritten. This function is called when the timer registered in the process function is passed. Instead of the event itself it has the timestamp of the timer as input parameter. With this timestamp the corresponding value in the map is retrieved. The `currentAggregatedValuesState` is then updated as before, but this time the current value is removed from the aggregated value and the pair is removed from the state. Finally, the distribution statistics are calculated based on the newly aggregated value and returned together with the needed context information.

As a result, the output stream is of type `DistributionStatisticsOverPeriod`.

5.2.2 Window Slices With No Precise Main Window Length

The second method is split in two parts. The first part generates the slices, whereas the second part aggregates the slices as explained in 4.3.3. This two parts are combined in the function `calculateStatisticsFixedSlicingWindows`, which is the main function of this method (*cf.* Listing 5.34). Beside the two parameters `authStream` and `timeOfRelevancy`, this method has another input parameter called `sliceLength`. It defines how much time a slice should cover.

```

1 def calculateStatisticsFixedSlicingWindows(
2     authStream: DataStream[Authorisation],
3     timeOfRelevancy: Time, sliceLength: Time
4 ) : DataStream[DistributionStatisticsOverPeriod] = {
5     val centralMomentsPerSlicePerKey:
6         DataStream[SliceMomentumStatistics] =
7         getAggregatedSlices(authStream, timeOfRelevancy, sliceLength)
8     val distributionStatistics:
9         DataStream[DistributionStatisticsOverPeriod] =
10         combineAggregationsOfSlices(centralMomentsPerSlicePerKey,
11             timeOfRelevancy.toMilliseconds)
12     distributionStatistics
13 }

```

Listing 5.34: CalculateStatisticsFixedSlicingWindows

The first part processes the input stream and creates a stream of `SliceMomentumStatistics` elements. It is then processed in the second part to generate the final stream with elements of type `DistributionStatisticsOverPeriod`. The type of the final stream is already known from the first method in Section 5.2.1. Yet, the type of the elements in the intermediate stream is unknown. Such a `SliceMomentumStatistics` element includes all information of the corresponding slice needed in the second part. Its composition is shown in Listing 1st:sms. In general, it is similar to the `DistributionStatisticsOverPeriod`, but is used for another concern. The `key` and the `window` parameter describe to which key and time window the calculated momentum statistics (stored in the `aggregator` parameter) belong. The information about the key and the time window is needed in order to remove elements that are not relevant anymore. As a fourth parameter the `currentTime` is needed, which is equal to the time of the newest element in the window.

```

1 case class SliceMomentumStatistics(
2     key: String,
3     aggregator: MomentumStatisticsAggregator,
4     currentTime: Long,
5     window: CustomTimeWindow
6 )

```

Listing 5.35: SliceMomentumStatistics

Creating Window Slices

The part which is responsible for creating the slices is located in the function `getAggregatedSlices` (*cf.* Listing 5.36). The function needs the same input parameters as the base function. As the goal is to find out the distribution statistics of the amount spent per authorisation, the incoming stream is again keyed by the `customerId`. After that the stream is divided into tumbling windows with a length defined by the input parameter.

For a relevant time of 24 hours a slice length of half an hour is chosen. The tumbling windows represent the window slices. Next, the allowed lateness of a window is defined. As the slices are used for the final calculations, it has to be active at minimum for the length of the global window. Hence, the parameter `timeOfRelevancy` is assigned as the allowed lateness here.

```

1 def getAggregatedSlices(
2     authStream: DataStream[ Authorisation ] ,
3     timeOfRelevancy: Time,
4     sliceLength: Time
5 ): DataStream[ SliceMomentumStatistics ] = {
6     authStream
7     .keyBy( _ . customerId )
8     .timeWindow( sliceLength )
9     .allowedLateness( timeOfRelevancy )
10    .trigger( CountTrigger . of( 1 ) )
11    .aggregate( new
12        IncrementalMomentumStatisticsAggregateFunction() ,
13        new MomentumStatisticsOfASlicePWF()
14    )
15 }

```

Listing 5.36: Flink Program to Create the Window Slices

In a next step, it is defined when a slice should trigger its process function. As the goal of the intermediate step is to tell the global window when an event is added and what the new aggregated value is, it has to create a `SliceMomentumStatistics`. For such needs Flink provides a standard trigger called `CountTrigger`. It always triggers computations when a number of elements arrived. As the trigger should fire once for each event, the number of elements is set to 1 as seen in code line 8 in Listing 5.36.

Since this method does not remove individual elements inside a slice, no evictor is needed. After all the settings and components of the Flink program are initialized, a process functions needs to be defined. As by default, a `ProcessWindowFunction` stores each of its element in memory. Thus, an `AggregateFunction` has to be used which combines the incoming values to one aggregated value. Subsequently, only this value is used by the `ProcessWindowFunction`. The most important parts of the `AggregateFunction` is presented in Listing 5.37. The types between the square brackets in the function header defines the type of the input value, the type of the aggregated value and the type of the returned value. The aggregated value and the returned value are both of type `SliceMomentumStatistics`, which is the target type of the program. The `AggregateFunction` consists of 4 methods namely `createAccumulator`, `add`, `getResult` and `merge`. The `merge` function is not shown, as it is not needed in this case. The `createAccumulator` functions assigns an initial value to the accumulator. The `add` function defines what happens when a new element arrives. First, the aggregated value is updated by the same functions used before in Section 5.2.1. Finally, the new value is returned and the `accumulator` is updated accordingly. The `getResult` function defines the output of the `AggregateFunction`.

```

1  class IncrementalMomentumStatisticsAggregateFunction extends
    AggregateFunction[Authorisation, SliceMomentumStatistics,
        SliceMomentumStatistics] {
2  override def createAccumulator() = SliceMomentumStatistics("",
        MomentumStatisticsAggregator(0L, 0.0, 0.0, 0.0, 0.0), 0, null)
3
4  override def add(value: Authorisation, accumulator:
        SliceMomentumStatistics) = {
5      //update current momentum statistics
6      val momentumStatistics = addValueToStat(accumulator.aggregator,
        value.amount.toDouble)
7      //update current time if necessary
8      var currentTime = accumulator.currentTime
9      if(value.eventTime > currentTime){
10         currentTime = value.eventTime
11     }
12     // return the new values, the accumulator will be set to these
13     SliceMomentumStatistics("", momentumStatistics, currentTime,
        null)
14 }
15
16 override def getResult(accumulator: SliceMomentumStatistics) = {
17     accumulator
18 }
19 }

```

Listing 5.37: AggregateFunction to Apply Before the ProcessWindowFunction

```

1  class MomentumStatisticsOfASlicePWF extends
    ProcessWindowFunction[SliceMomentumStatistics,
        SliceMomentumStatistics, String, TimeWindow] {
2
3      def process(
4          key: String,
5          context: Context,
6          aggregator: Iterable[SliceMomentumStatistics],
7          out: Collector[SliceMomentumStatistics]
8      ) = {
9          out.collect(SliceMomentumStatistics(
10             key, aggregator.last.aggregator,
11             aggregator.last.currentTime,
12             CustomTimeWindow(context.window.getStart,
                context.window.getEnd)))
13     }
14 }

```

Listing 5.38: ProcessWindowFunction of the AggregateFunction

A `ProcessWindowFunction` is then applied over the returned results of the `AggregateFunction`. As shown in Listing 5.38, this function just adds some context information to the element returned by the `AggregateFunction`. Finally, the result of it is the stream of `SliceMomentumStatistics` elements needed in the second part of the algorithm to aggregate the slices.

Aggregation of Window Slices

In Listing 5.34, a stream of `SliceMomentumStatistics` objects was created, which will be processed in the second part to create a stream of `DistributionStatisticsOverPeriod` elements. The method, which is responsible for the aggregating of the slices, is shown in Listing 5.39. Beside the input stream it needs the `timeOfRelevancyInMs` as input parameter. It defines the duration for which a slice is valid and when it can be dropped. As before, the stream is keyed with the same key as before, which is the attribute `customerId`. After that, the window parameter is set to a global window, since the second part should combine all window slices together. Hence, it needs to include them all in its window. The `CountTrigger` is used as trigger again as the second part has to create new results for each incoming element.

```

1 def combineAggregationsOfSlices(
2     statsPerSlice: DataStream[SliceMomentumStatistics],
3     timeOfRelevancyInMs: Long) = {
4     statsPerSlice
5     .keyBy(_.key)
6     .window(GlobalWindows.create())
7     .trigger(CountTrigger.of(1))
8     .evictor(new OldSlicesEvictor(timeOfRelevancyInMs, false))
9     .process(new
10         StatisticalDistributionOverSlicesPWF(relevantTimeWindowInMs))

```

Listing 5.39: Flink Program to Aggregate the Window Slices

For the present example, also an evictor is needed since the old or duplicated values of a slice have to be removed. The used `OldSlicesEvictor` has two input parameters. The parameter `timeOfRelevancyInMs` defines after which time an element is evicted. The parameter `doEvictAfter` defines if the evictor should be called before processing the elements or afterwards. In this case, the parameter is set to false and thus the evict method is applied before the process function. This was done because old slices have to be removed before the new computations are applied. A simplified version of the `OldSlicesEvictor` is shown in Listing 5.40. In a first step, the new record is gotten from parameter `elements`, which contains all elements in the window. Considering the new element, the current time is updated if necessary. By subtracting the `timeOfRelevancyInMs` from the current time, the evict time is calculated. It defines the time at elements are not relevant anymore. Hence, as a last step window slices with an end time before the evict time are removed.

```

1 private def evict(
2     elements: Iterable[TimestampedValue[SliceMomentumStatistics]],
3     size: Int,
4     ctx: Evictor.EvictorContext): Unit = {
5
6     // get the new record
7     var iterator = elements.iterator
8     var newRecord: elements.getNewRecord
9
10    // update the current time if necessary
11    var currentTime = ctx.getCurrentWatermark
12    if(newRecord.currentTime > currentTime){
13        currentTime = newRecord.currentTime
14    }
15
16    // calculate the time before which slices can be evicted
17    var evictCutoff = 0L
18    if (currentTime > 0) evictCutoff = currentTime -
19        timeOfRelevancyInMs
20
21    // remove all slices with a time smaller than the calculated
22    // time and each duplicated not up to date slice
23    iterator = elements.iterator
24    while (iterator.hasNext) {
25        val record = iterator.next.getValue
26        if(record.window.endTime < evictCutoff){
27            iterator.remove()
28        } else if(record.window.startTime ==
29            newRecord.window.startTime){
30            if(iterator.hasNext) iterator.remove()
31        }
32    }
33 }

```

Listing 5.40: Evict Method of OldSlicesEvictor

After all components are set up, the `ProcessWindowFunction` is defined. It combines all remaining relevant slices. The code of it is shown in Listing 5.41. The types in the square brackets again define the type of the input elements, the output elements, the key and the window used.

As a first step, an aggregator of the type `MomentumStatisticsAggregator` is initialized with 0 assigned to all its values. After that the momentum statistics of each slice is added to the aggregator one by one. Finally the distribution statistics are calculated out of the momentum statistics and the result is emitted.

```

1  class StatisticalDistributionOverSlicesPWF(windowLength: Long)
    extends ProcessWindowFunction[SliceMomentumStatistics,
        DistributionStatisticsOverPeriod, String, GlobalWindow] {
2
3  def process(key: String,
4             context: Context,
5             input: Iterable[SliceMomentumStatistics],
6             out: Collector[DistributionStatisticsOverPeriod]) = {
7
8      var aggregator = MomentumStatisticsAggregator(0,0,0,0,0)
9
10     // add the momentum statistics of each slice one by one
11     for (inFull <- input) {
12         val in = inFull.aggregator
13         if (aggregator.count == 0) {
14             aggregator = in
15         }
16         else {
17             aggregator = combineRawMoments(aggregator, in)
18         }
19     }
20
21     // emit the statistical values
22     val currentTime = input.last.currentTime
23     out.collect(
24         DistributionStatisticsOverPeriod(
25             key,
26             getStatisticValues(aggregator),
27             CustomTimeWindow(currentTime - windowLength, currentTime)
28         )
29     )
30 }
31 }

```

Listing 5.41: ProcessWindowFunction to Combine the Remaining Relevant Slices

5.2.3 Combination of the Generated Test Data and the Aggregations

Now that the methods for the aggregations were introduced, the next section will describe how the functions are applied to the generated test data. Listing 5.42 shows a simplified version of the main method which combines the data generation and the aggregations. This method sets up the environment, generates a test input stream and then runs the aggregations on that stream. At the beginning in line 3-5 the Flink environment is initialized with the default stream execution environment of Flink along with some specific

adaptions. To avoid concurrency problems, the execution environment is set to parallelism 1. This means that every operation runs on a single instance. To optimize the performance, this value can be increased. This way, different keys are handled by different instances. However, at the current state the operations only run on one thread. As discussed in Section 2.8.4, the program has to process the stream in event time. Thus, the time characteristics of the execution environment is set to event time.

```

1  def main(args: Array[String]): Unit = {
2      // setup environment
3      val env = defaultEnvironment
4      env.setParallelism(1)
5      env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
6
7      // create test stream of authorisations
8      val startTime =
9          Instant.parse("2020-01-01T00:00:00.00Z").toEpochMilli
10     implicit val timer = getFullSpeedTimer(startTime)
11     val authStream = TestStreamGenerator.generateModelAuths(env)
12
13     // apply the aggregations over the test stream
14     val timeOfRelevancy = Time.hours(24)
15     val sliceLength = Time.minutes(30)
16     val distributionStatistics =
17         calculateStatisticsFixedSlicingWindows(authStream,
18             timeOfRelevancy, sliceLength)
19
20     // execute the code and print the result
21     distributionStatistics.print
22     env.execute()
23 }
```

Listing 5.42: Structure of the Main Method of the Flink Program

In a next step, the input data stream is created in line 8-10. The parameter `startTime` defines the start date from when on the events should be created. As a start date, first of January 2020 at midnight was picked. The timer on line 9 defines how the events should be generated, either in real time or as fast as possible.

Using the defined environment and timer a stream of events is generated by the function `generateModelAuths`. Listing 5.43 shows the method to create the input stream. It takes the defined environment and timer as input parameter. The function `genStructure` explained in Section 5.1.6 is applied to create a stream of authorisations and transactions in line 2 and 3. The watermark assigner function `PunctuatedWatermarkAssigner` simply sets the watermark equal to the last timestamp if it is bigger than the current watermark. Like this, the time is as advanced and calculations are triggered as fast as possible (*cf.* Section 2.8.4).

As mentioned before, a stream of authorisations and transactions is generated by the **gen-Structure** function. However, in the current implementation only the authorisations are considered. At this point, the created stream of authorisations is ready to be processed by Flink.

```
1  def generateModelAuths(env: StreamExecutionEnvironment)(implicit
    timer: ContextTimer) = {
2      val src =
3          new TemporalEventSourceFunction[Authorisation](genStructure,
                AuthTransEventEffectSourceFunction.transTag)
4      val authStream =
5          env.addSource(src)
6              .assignTimestampsAndWatermarks(new
                PunctuatedWatermarkAssigner)
7      authStream
8  }
```

Listing 5.43: Creating the test input stream of authorisations

In lines 13 - 15 the generated stream is given as input to the aggregation method, which works with fixed window slices. The other method could have been picked here as well. The parameter for the length of a slice can be chosen by the developer. Less slices mean that more values are contained in a slice and thus reduce the performance of the function. In this example, half an hour is picked for the length of the slices.

Finally in line 19 the environment is executed and the results can be used in further pipelines or simply printed out to the command line as done in line 20. These explanations conclude the implementation of the aggregations. Now the evaluation of the project and its artifacts can start.

Chapter 6

Evaluation

This Chapter evaluates the solutions implemented in this thesis and provides statements about the time and space complexity, as well as clear indications about the achieved performance. In a first Section, the test data generator is examined. Also, the tree structure used in it is checked for its promises regarding performance ($O(\log(n))$ for insert, delete, and lookup). After that, the aggregations with Flink are evaluated.

6.1 Test Data Generators

The evaluation of the test data generators can be divided into two different Sections, each of which has an impact on the measured overall performance. One is the data structure used, which should allow to perform search, delete, and insert operations with a complexity of $O(\log(n))$. On the other hand space requirements and memory allocation are also of great importance. For this purpose the tool Prometheus is used to analyze the effective events per second in different system compositions. The memory requirements of the system are investigated with VisualVM, which uses the profiler provided in the Java Development Kit (JDK) 1.8.

6.1.1 Tree Operations

The most promising way to check the performance of the augmented Red Black Tree structure is to test if the properties of the tree are fulfilled or not. Accordingly, the following requirements must be met, which have an impact on performance.

- The height of a tree with n elements must never be greater than $2 \times \log_2(n + 2) - 2$
- Prefix search in $O(\log(n))$
- Key search in $O(\log(n))$
- Deletions in $O(\log(n))$
- Additions in $O(\log(n))$

Maximum Height

First, a test is performed to determine the maximum height of the tree. As mentioned in Chapter 5, a separate attribute called height was integrated into the tree nodes, which recursively represents the height of the tree. If the root node is queried for the height, the maximum height of the tree is obtained automatically. The code snippet which makes this possible is presented in Listing 6.1.

```

1  val height: Int = {
2      if (left == null && right == null) 1
3      else if (left == null) right.height + 1
4      else if (right == null) left.height + 1
5      else math.max(left.height, right.height) + 1
6  }
```

Listing 6.1: Height Attribute in Tree Node

With the following snippet it can be checked automatically for 100 executions (with different number of nodes and values) if the condition $height \leq \frac{2 \times \text{Math.log}_{10}(\text{amount})}{\text{Math.log}_{10}(2)}$ holds. It states that a tree must never grow larger than the logarithmic size required. This statement is important for example for the range queries, additions and the deletions. These functions return new trees and not just nodes like in the prefix and key search. Thus, it is important to check that the results are correct and meet the requirements.

```

1  property("GetLargeTree") = Prop.forAll(getLargeTree(amount)) {
2      (tree: Option[RBT[Long, Long, Double]]) =>
3      val initialAmount = amount
4      amount = amount + 80000
5      tree.get.height <= 2 * Math.log10(amount) / Math.log10(2)
6  }
```

Listing 6.2: Test Case to Check The Maximum Size of a Tree

The function `getLargeTree(amount)` is used frequently in the following tests and is implemented as shown in Listing 6.3. First a start time is defined, which is later used to set the total duration. Then the nodes are created and inserted into the tree. Finally, the time needed to instantiate the tree is saved and the tree is returned.

```

1  def getLargeTree(amount: Long): Option[RBT[Long, Long, Double]] = {
2      val time1 = System.nanoTime()
3      val firstNode = genNode(0).pureApply(parameters, seed)
4      var t1 = RBT(Black, firstNode._1, firstNode._2, firstNode._3)
5
6      var range = 1L.to(amount);
7
8      for (x <- range.toList) {
9          val node = genNode(x).pureApply(parameters, seed)
10         t1 = Some(t1.get.add(node._1, node._2, node._3))
11     }
12     val time2 = System.nanoTime()
13     instantiationTime = time2-time1;
14     t1
15 }

```

Listing 6.3: Generator That Creates a Tree With a Certain Number of Nodes

These test cases run smoothly and since the informational gain of a picture on which is written "100 tests successful" is not very high, it will not be presented here. Therefore, the next Sections will provide the outcome of the tests in a graphical form and statements about how long the operations take with growing trees and where the possible limitations are.

Search by Key

The search for the key is verified by looking at the number of nodes touched and whether this number falls within the complexity of $O(\log(n))$. For this purpose, the maximum tree depth is marked in blue and the viewed nodes in orange. The viewed nodes (orange line) must never cross the total height (blue line). Furthermore the trend line is to be considered, which must take a logarithmic course. Additionally, the time needed to find a key in an ever growing tree and the time needed to create the tree itself were also taken into account. The creation of the tree must be linear due to the growing number of nodes, whereas the time to search should follow a logarithmic curve (as the number of nodes grows very fast in the following examples, the curve will almost look constant). The tree starts at 1 node and increases by 80000 nodes with each test run. Finally, the test tree should end at 8 million nodes. The tree elements are a tuple consisting of an integer as key, an integer as value and a double as weight. As shown in Figure 6.1, the requirements mentioned above are met.

Search by Prefix

Again, as a setup, the test starts with one node and gradually increases up to 8 million. At each step a prefix search is performed to observe the time it takes to perform the operation under test with an increasing tree. As can be seen from the example in Figure 6.2, the

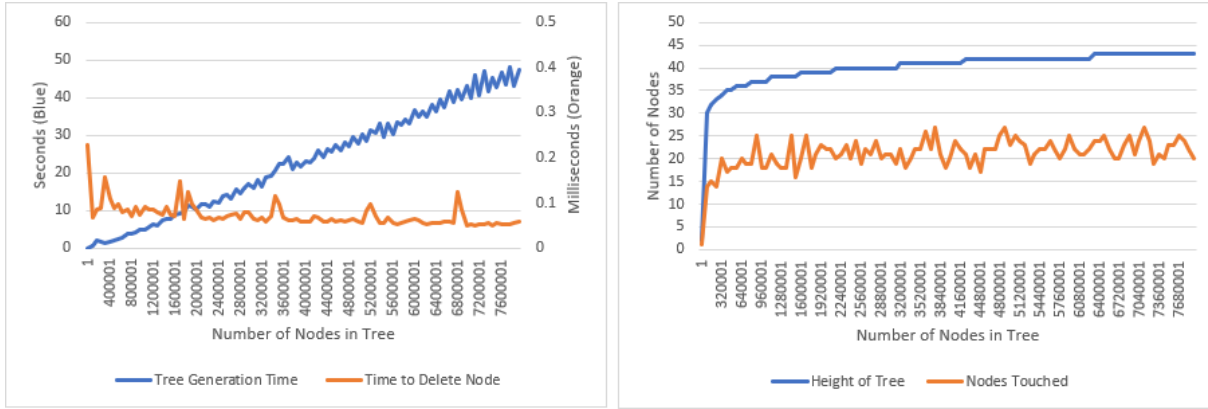


Figure 6.1: Comparison Between Tree Size and Time to Search for a Key

result is again similar to the key search. But what is also noticeable is that it takes more time per action at the beginning than towards the end. This can be explained by the warm-up of the system, which behaves the same even if the number of nodes remains the same (checked with 1, 10 and 100 nodes without increase). In addition, the number of nodes to be touched for a search does not change much afterwards, which gives the impression of a constant curve.

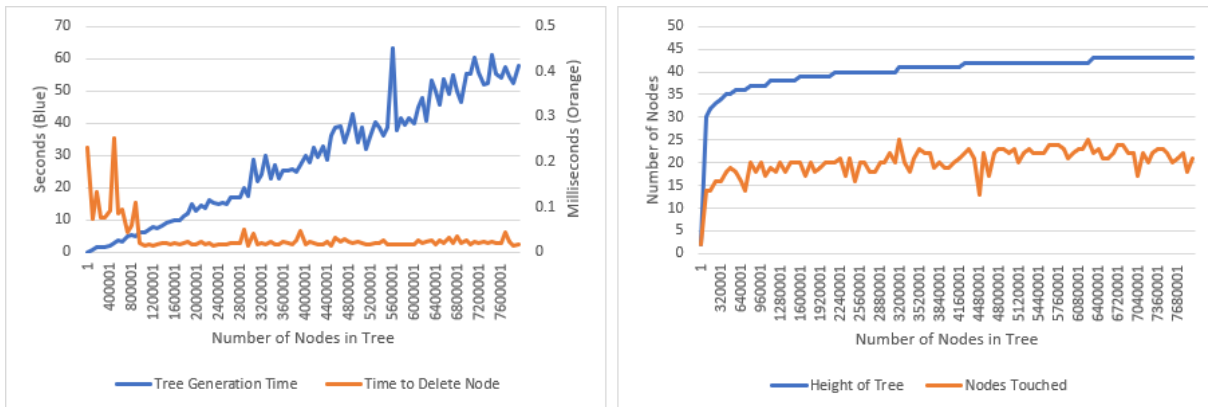


Figure 6.2: Comparison Between Tree Size and Time to Search for a Prefix

Addition of Nodes

When adding the nodes, the same approach as before was used. This means that on the one hand the time for inserting a node on a growing tree was considered and on the other hand the required touching of levels when inserting. The tree was again started at 1 node and should increase by 80000 each time. The results shown in Figure 6.3 clearly state that again the requirements are met.

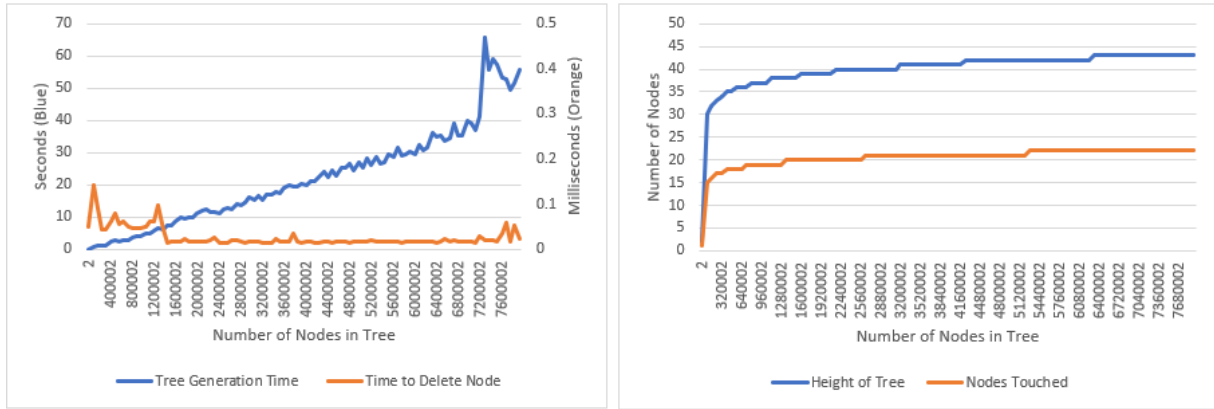


Figure 6.3: Comparison Between Tree Size and Time to Add a Node

Deletion of Nodes

The deletion of nodes is a difficult task. So it is even more interesting to see how the performance behaves. Again, the number of nodes to be touched for a delete operation as well as the time for deleting is shown. The tree starts initially with 8 million nodes and is expected to grow to 16 million. As shown in the Figure 6.4 the tree behaves as desired. The curve on the right side shows the later course of a logarithmic function. Also the time to delete the elements lies in an acceptable range of around 0.1 milliseconds.

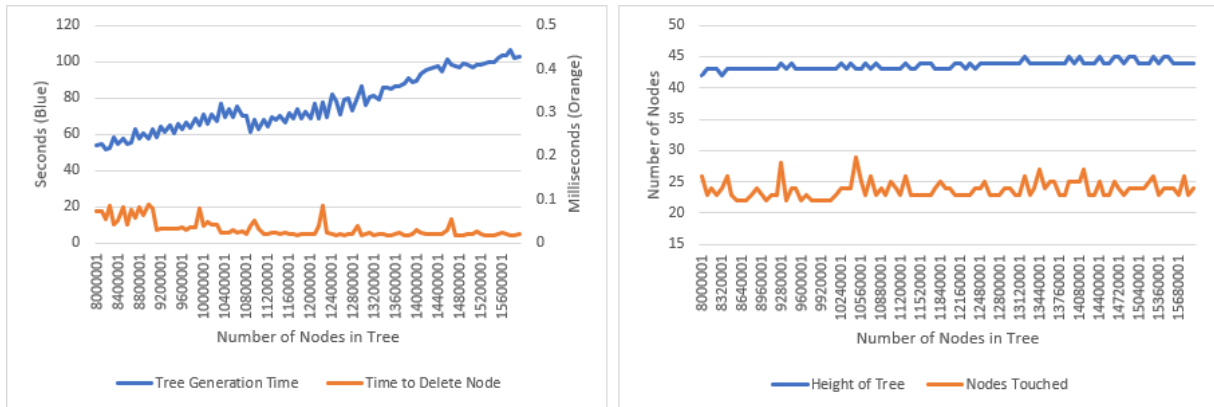


Figure 6.4: Comparison Between Tree Size and Time to Delete a Node

Regarding the complexity of the tree operations in big O notation, it can be said that it has been proven that all of them meet the requirements and that the performance is guaranteed despite the large number of objects. As can be seen in the diagrams, there are always single spikes in the trees. Some bigger and some smaller. These spikes and their effects will be discussed in the next Section, as they have an influence on the performance of the structure.

6.1.2 Resource Requirements

If one looks at the outliers during tree structure generation, the question immediately arises where exactly they come from. Up to now the trees have been tested with tuples of the type (Int,Int,Double), which means a relatively low resource consumption. But what happens when real entities, such as the merchants, are to be created? When generating the effective merchants, the expected linear increase in the generation time of the tree was observed up to a certain number. However, from a certain point in time (approx. 6 million merchants) the growth becomes exponential as it can be seen in Figure 6.5. Also during the addition of more and more nodes to the tree, there seem to appear spikes on a regular basis. As the tree itself performs as expected under normal circumstances, a misconception in the tree can be excluded as a potential source of the issue. So what remains is the garbage collector, which may have an impact on the process. To further investigate this suspicion, the two tools Java Object Layout (JOL) and VirtualVM were used to track exactly how much resources the individual entities require and where there is a need for optimization.

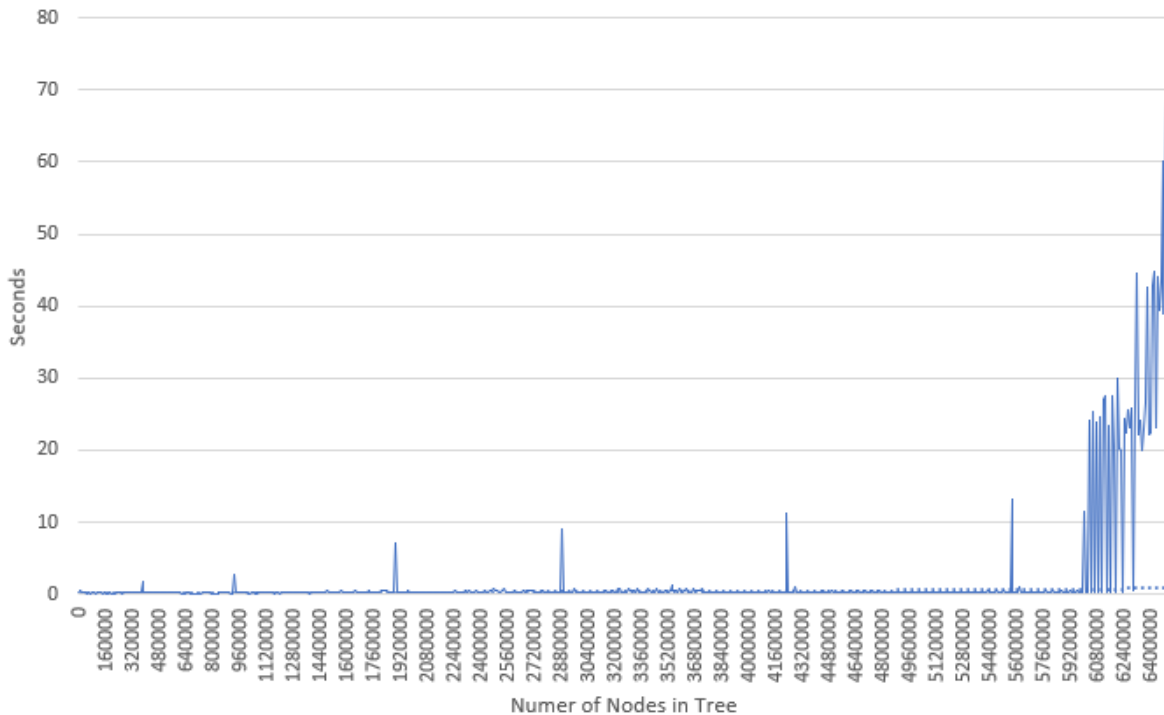


Figure 6.5: Exponential Growth of Time it Takes to Generate the Tree

One problem that came up when looking at the profiler in Figure 6.6 was that with millions of objects it is important what kind of key, weights and values are stored in each RBT node. There seemed to be a lot of Tuple4 and Tuple3 objects that reflect the tree keys and use a lot of space for their string attributes. Nevertheless, the biggest problem seemed to be the usage of Doubles as weights. According to the profiler, there is large number of active *Double* objects and their wrappers that inflated the whole construct. As a potential relief, all Double weights were converted to integer weights in order to save unnecessary load. Next, the total object size of the merchants were considered. For this, the tool JOL

was used, which allows a detailed insight into the object structure (*cf.* Figure 6.7). The size of a single object was with around 3100 bytes almost gigantic and it became more and more clear why the memory was full. With the simple calculation of 3100 bytes * 6 million objects a sum of about 18 gigabytes of memory is needed. Therefore, it was tried to optimize the object structure and to evaluate where the biggest consumers of storage space are to be found. It is important to note at this point that the listing seen in Figure 6.7 can even be shown in a more detailed version, where also the exact place and name of the variable assigned with the memory usage is depicted.

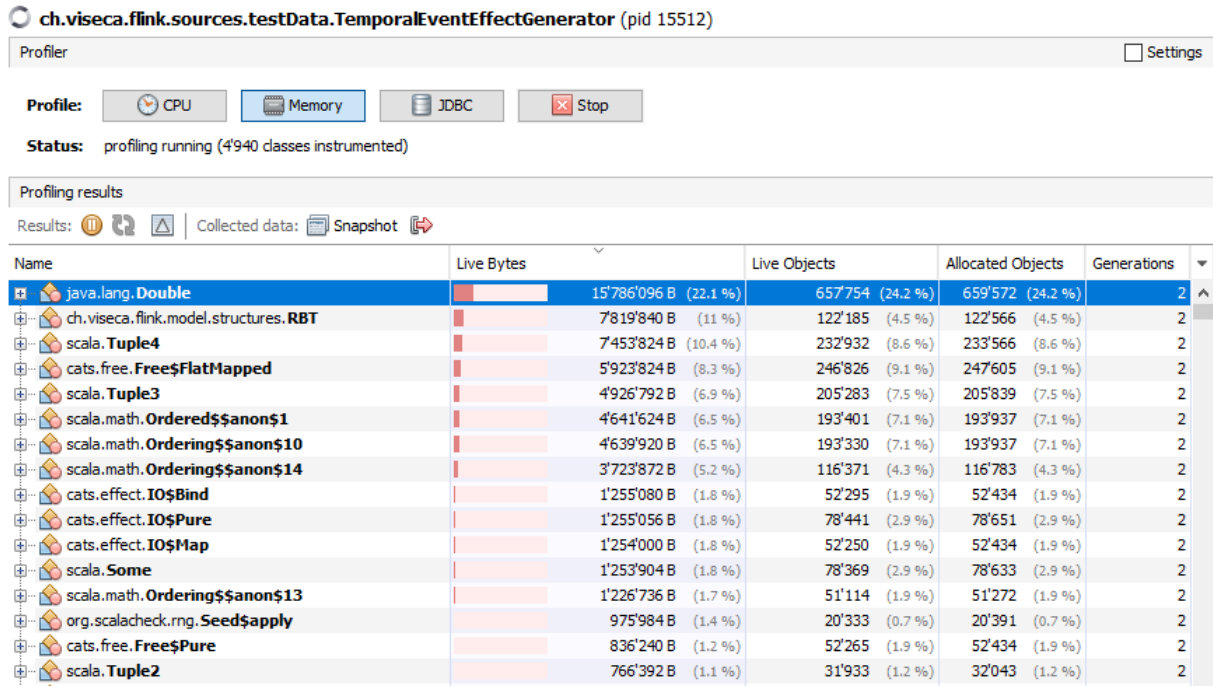


Figure 6.6: Weights of RBT Nodes

According to JOL it also became clear that the other big problem besides the doubles was the storage of the RBT structure (700 bytes for the overall structure alone) for card usage patterns, as well as the storage of actual mcc and region objects on the merchants themselves. As a consequence the merchant category code and region objects were changed and only their key was saved. This means that if the effective object is needed, it must be fetched from the state bag first. Furthermore, the tree structure for the card usage patterns was changed to a Gen to save space. For this purpose a lightweight alternative has to be found at a later time. Possible options for very small data structures include storing them in a normal list, in which the currently valid ranges must be calculated for each add, query or delete. For small collections, however, this is unproblematic. Nevertheless, a trade-off must take place between calculation time and storage space. Additionally, categorical attributes like the mcc group were converted to integers. This also saved some space per entity. With all these adjustments, the object structure could be reduced to an overall size of 600 bytes. This is still rather large, but with these changes it was possible to test with the required number of merchants for issuing the events. As a conclusion on resource utilization, it can be said that the garbage collector reaches its limits after a certain amount of memory usage and thus requires ever greater delays until

Class: Merchant(156,M156_USA_5818,Region(US,USA,840,United States of America,United States,N,2693427.0))

COUNT	AVG	SUM	DESCRIPTION
25	36	912	[C
1	80	80	ch.viseca.flink.model.Mcc
1	48	48	ch.viseca.flink.model.Merchant
1	16	16	ch.viseca.flink.model.Merchant\$\$\$Lambda\$916/1579254274
1	16	16	ch.viseca.flink.model.Merchant\$\$\$Lambda\$917/1922346036
1	16	16	ch.viseca.flink.model.Merchant\$\$\$Lambda\$918/1579340224
1	16	16	ch.viseca.flink.model.Merchant\$\$\$Lambda\$919/1875478998
1	48	48	ch.viseca.flink.model.Region
1	16	16	ch.viseca.flink.model.structures.Black\$
3	64	192	ch.viseca.flink.model.structures.RBT
1	16	16	ch.viseca.flink.model.structures.Red\$
1	16	16	ch.viseca.flink.testData.BaseOperators\$\$\$Lambda\$925/1940238042
1	24	24	ch.viseca.flink.testData.BaseOperators\$\$\$Lambda\$927/1405540655
9	24	216	java.lang.Double
25	24	600	java.lang.String
10	40	400	java.math.BigDecimal
1	24	24	java.math.MathContext
1	24	24	java.math.RoundingMode
1	24	24	org.scalacheck.Gen\$\$\$Lambda\$888/58204990
1	24	24	org.scalacheck.Gen\$\$\$Lambda\$926/1202592832
3	24	72	org.scalacheck.Gen\$\$anon\$3
1	32	32	org.scalacheck.Gen\$Choose\$\$anon\$6\$\$\$Lambda\$889/1218696049
1	16	16	scala.Some
10	24	240	scala.math.BigDecimal
1	16	16	scala.math.Numeric\$DoubleIsFractional\$
103		3104	(total)

Figure 6.7: Space Requirements of a Merchant

the entire system is no longer capable of functioning. This must be prevented under all circumstances. Ideas how this can be achieved are described in Chapter 7.

6.2 Generation of Events

The actual goal of the data generation was to create the possibility to simulate patterns of the productive data to create a test environment for the aggregations. One of the key points, and most likely the biggest source of events, are the authorisations. As no exact number can be stated here, an approximate number of 1 million authorisations per day will be expected to be simulated. In addition, a single-digit million number of customers and merchants must be generated in the system. As test values, a number of 8 million different merchants and 5 million customers were selected. The test data generation was then visualized using Prometheus and Grafana to track the number of events per second. The results of this analysis can be seen in Figure 6.8. These recordings were made using the full-speed environment, which means that the real world delays are only simulated but not effectively executed. This allows to test the maximum load of the system. The number of events emitted varies between 50 and 70 thousand events per second. Extrapolated to

one day, a pessimistic calculation amounts to $50000 * 60 * 60 * 24 = 4'320'000'000$ events, which exceeds the desired requirements by far and offers potential for extensions.

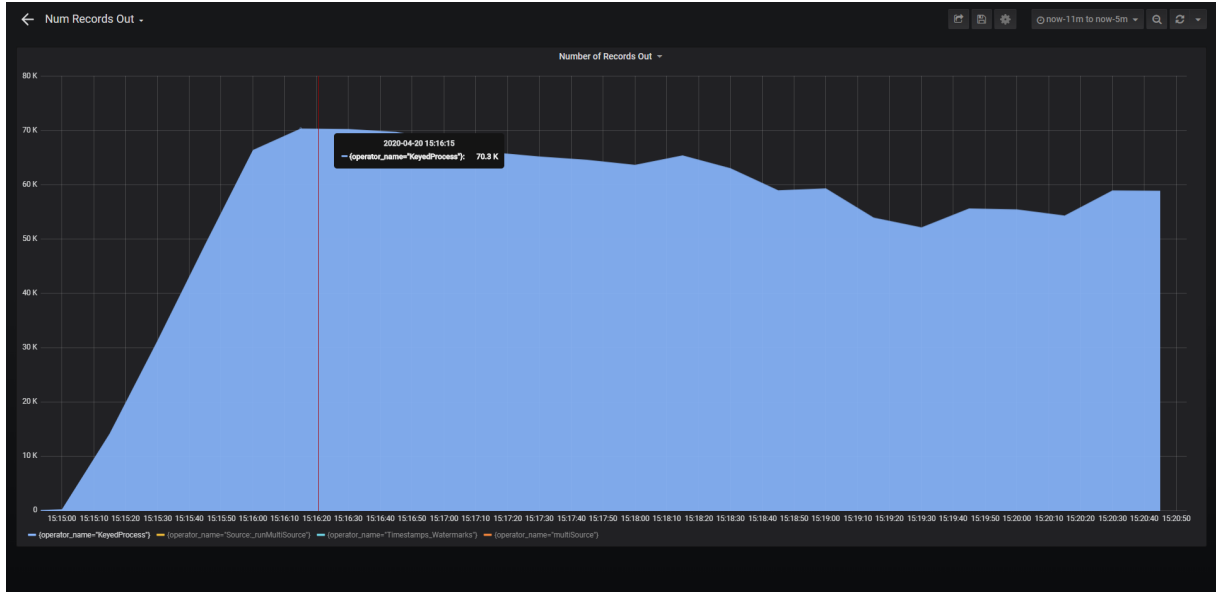


Figure 6.8: Events/s of the Test Data Generator

6.2.1 Determinism

Another important sub point to look at is determinism. For this purpose two independent Avro outputs were generated, which can be read back as source afterwards. Afterwards a summation of an attribute has to be done, because the events are (intentionally) not stored in the same order in the Avro file due to the built-in delay. However, this could be switched off if desired. The files should then be identical. If the final results of the aggregations match, it can be assumed that the same data is available and it is therefore proven that the random generation of the data is deterministic. As a test case, all amounts of the authorisations of the countries should be added together. The time to be considered was set to 1 hour. For the summation of the values a `KeyedProcessFunction` was implemented. The values are then aggregated and printed as in Listing 6.4. The aggregations were carried out twice, once for the data in the first file and once for the data in the second file. Both of which were created independently. As a simplification, only the aggregation for the first file is shown in this code example, however, the aggregation for the second file is identical. It is important to note that due to the filter commands only values up to a certain point in time are considered. Furthermore, the test should only show the values for Switzerland and Germany.

```

1 stream1
2     .assignTimestampsAndWatermarks(new OutOfOrderWatermarkAssigner())
3     .filter(_.eventTime < 1588067000646L)
4     .keyBy(_.country)
5     .process(new SumAmountPerKeyProcessFunction)
6     .filter(x => (x._1 == "CHE" || x._2 == "DEU"))
7     .print("Auth_File_1")
8 }

```

Listing 6.4: Aggregation of Generated Test Data to Prove Determinism

As a result, the following two lines were displayed, which confirm the functioning of determinism. The last results displayed all the same amount for each country on both streams.

```

Auth_File_1> (CHE,CHE --> 79529.72287453721051467)
Auth_File_2> (CHE,CHE --> 79529.72287453721051467)
Auth_File_1> (DEU,DEU --> 11492.4477224064387348)
Auth_File_2> (DEU,DEU --> 11492.4477224064387348)

```

The evaluation showed that the chosen approach to generate deterministic random data based on probability distributions works and meets the requirements. However, there are still some problems left. These are namely the object size with increasing number of elements in the data structures, which puts a heavy load on the garbage collector. Possible solutions to this problem are discussed in Chapter 7. That concludes the evaluation of test data generation, the remainder of this Chapter is now dedicated to evaluating the aggregation of data using Flink.

6.3 Evaluating Correctness of Window Aggregations

Before the performance of the window aggregations can be measured, the correctness of the different implemented methods has to be evaluated. Thus, different test cases were defined for the window aggregations. The test cases consist of fixed events such that the result will be deterministic and can be analyzed. Finally, the results are compared to an expected value to check if they match. The elaborated unit tests evaluate if the event count correctly increases when an element is added. Furthermore, checks are executed to see if the count decreases again when the element lost its relevance. In more advanced tests the slicing will be evaluated. In order to test the functionality of the general method, it suffices to check the count and average parameter. This way it can be made sure that the correct element has been added or removed.

To check the validity of the aggregation functions the resulting statistics were compared with results from a calculus application, where well tested formulas for the parameters are used. These values are then taken as expected values and compared to the test results. Both implementations described in Section 5.2 passed the executed tests and thus, deliver the correct results.

6.4 Measuring Performance of Window Aggregations

To individually measure the performance of the window aggregations, the used test data needs to be separated from the data generation. To do so, the generated authorisations are stored in an Avro file. This Avro file can then be read as a stream and used as input to the aggregations. Like this the aggregations are not dependent on the test data generation anymore as the data was already created individually before. Thus, the performance can be measured separately. To analyze the performance the monitoring system Prometheus was used along with Grafana to display the graphs. As performance is key for the stream processing method the throughput of processed elements was measured. In the following paragraphs the two methods described in Section 5.2 are evaluated one by one. Before a method is evaluated its complexity is deduced.

6.4.1 No Window Assigner

First, the performance of the method with no window assigner is evaluated. As discussed in the comparison of the methods in Section 4.3.4 this method can add and delete values incrementally. Yet, to get the value that has to be removed it needs to store each event. Hence, the more elements are currently stored, the slower the removing of an event gets. As the values are stored in a `HashMap`, retrieving such an element has $O(1)$ complexity. The following list summarizes the complexity of its main operations.

- Adding and Removing an Element in $O(1)$

In total the function is assumed to have $O(1)$ complexity to process one element, hence to process n elements it has a complexity of $O(n)$. Hence the computations are not exponentially dependent on the number of elements, which is very important in the fraud detection case. However, the main influence on the performance is the total number of elements that has to be stored in storage, as this effects the general performance. To confirm this, different configurations for the number of events emitted by one customer and the number of customers in total are used. First an experiment is made where only 100 customers are created and each customer emits an event on average every 6 hours. Hence in total there are approximately 400 events per day, which should not matter considering the storage. Figure 6.9 shows the number of events emitted per second by the process function when applying these configurations. Since each input element is processed twice, one time when it arrives and one time when it is removed, the shown value needs to be divided by two, to find out how many input events where processed. In average around 90'000 output events are emitted and hence around 45'000 input events are processed.

However, in this example only a small number of events per day were processed. Hence, as a first step the time between card usages is decreased to 1 minute. This means that one customer can generate up to 1440 events per day. Multiplied by the number of customers in total there are 144'000 events per day. Figure 6.10 shows the number of elements evicted using these configurations. The function still processes around 35'000 events.

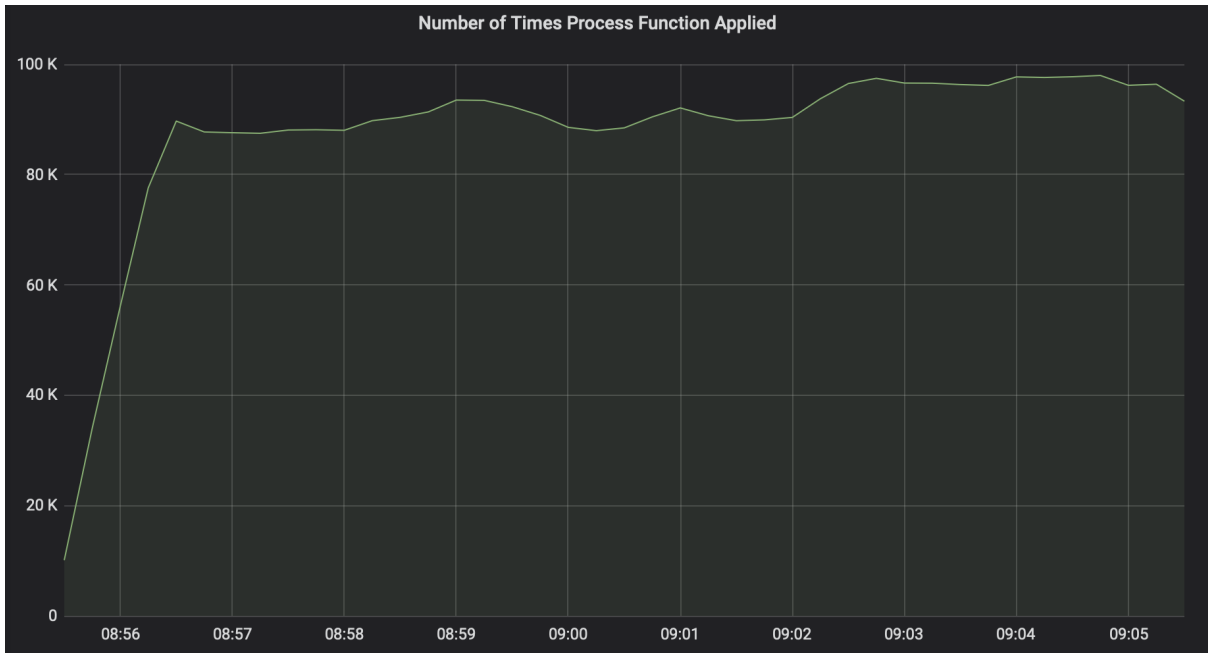


Figure 6.9: Events/s No Window Assigner / 100 Customers / AVG 6 Hours

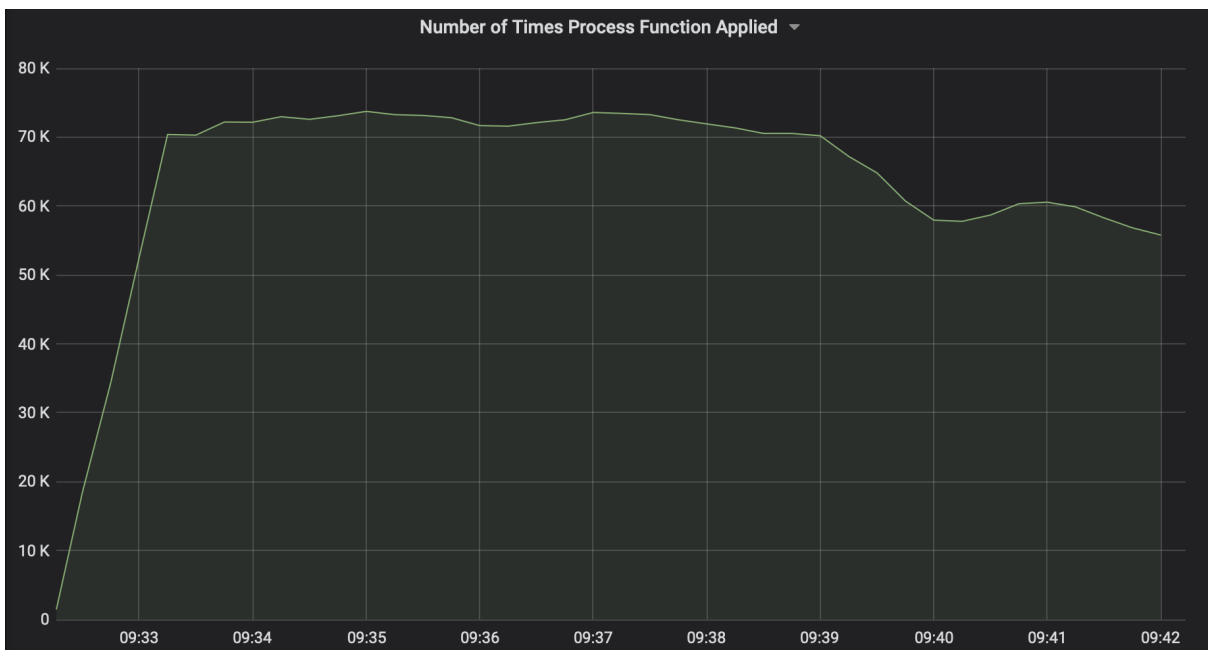


Figure 6.10: Events/s No Window Assigner / 100 Customers / AVG 1 Minute

It seems that the program can handle the amount of authorisations and the performance did not suffer a lot by increasing the number of events. Again the number of emitted events per day are increased by increasing the number of customer to 1'000. This leads to around 1'440'000 events per day, which all have to be stored at once. Figure 6.11 shows the number of emitted events. Now the amount of processed elements decreases by a considerable factor. At a point the processing rate falls down from around 22'000

element to 2'500 elements and then stays at this level. Thus, only around 1'250 input events could be processed per second afterwards. This again is due to the already known memory issues.

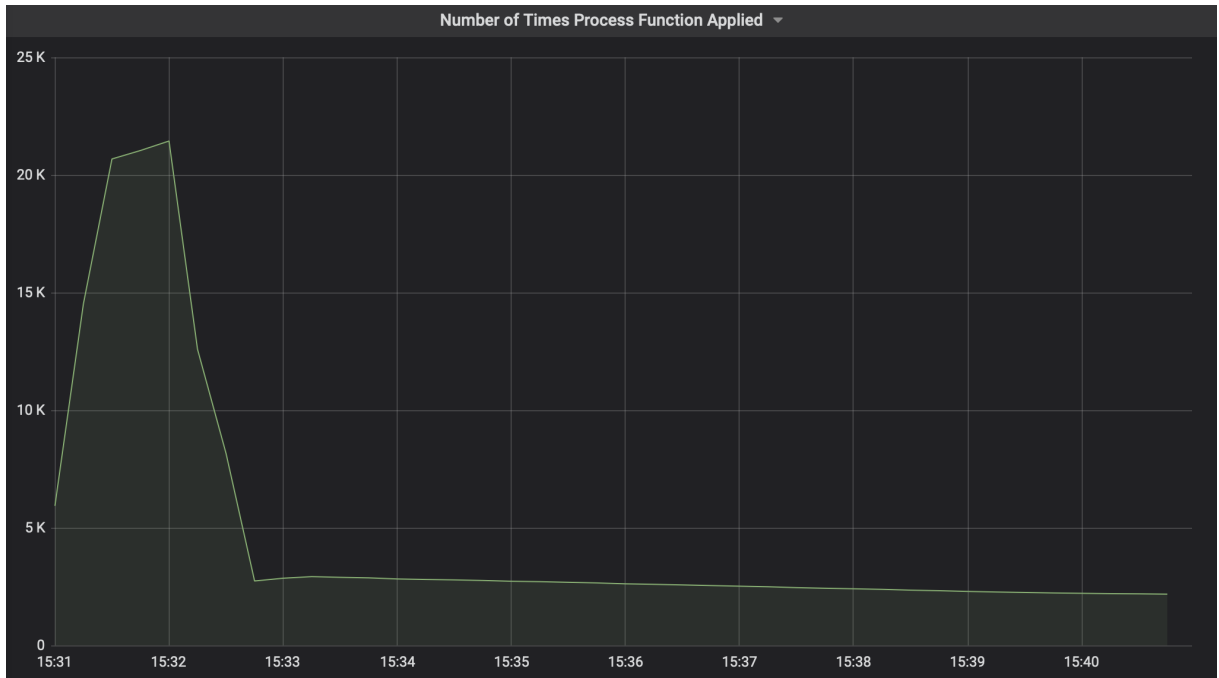


Figure 6.11: Events/s No Window Assigner / 1000 Customers / AVG 1 Minute

An approach to partially solve this problem is that each keyed stream is processed by a different process on a different machine as mentioned in Section 2.8.1. This would lead to a smaller memory usage per machine. However, a better way would be to actually not store each event as done in this method.

6.4.2 Window Slices Evicting Whole Slices

In this subsection the performance of the method with window slices is evaluated. As mentioned in the comparison of the methods in Section 4.3.4 this method does not exactly comprise the needed time window, nevertheless it offers a high performance. Adding an element can be done incrementally in the slice and as previous to combine the slices each accumulator of the window slices has to be considered. Removing of a single element is not necessary here. The following list summarizes the complexity of its main operations.

- Adding an Element to a Slice in $O(1)$
- Combining the Slices in $O(n)$

In total the function has $O(n)$ complexity to process one element, with n being the number of slices. Thus, this method is mainly dependent on the number of slices created. As before, tests with different configurations for the number of elements emitted by a customer

and the number of customers in total are used. For the first tests a fixed slice length of 30 minutes is chosen, which means that 49 slices are used per key. For this method it is assumed that the number of slices is much smaller than the number of elements. In order to be able to compare this method to the one before, the same configurations as in the first experiment of Section 6.4.1 are used. Hence, there are 100 customers emitting events on average every 6 hours. Figure 6.12 shows the amount of emitted output events. This time an event is only processed once and does not need to be evicted individually. For the mentioned configuration this method has a slightly worse performance than the first method and around 43'000 elements could be processed per second. This is because the number of elements needed to store in the first method was very small and therefore had nearly no impact on the performance. In this method, the added intermediate step of creating slices is just an unnecessary added complexity. However, the performance of this method is still considered feasible.

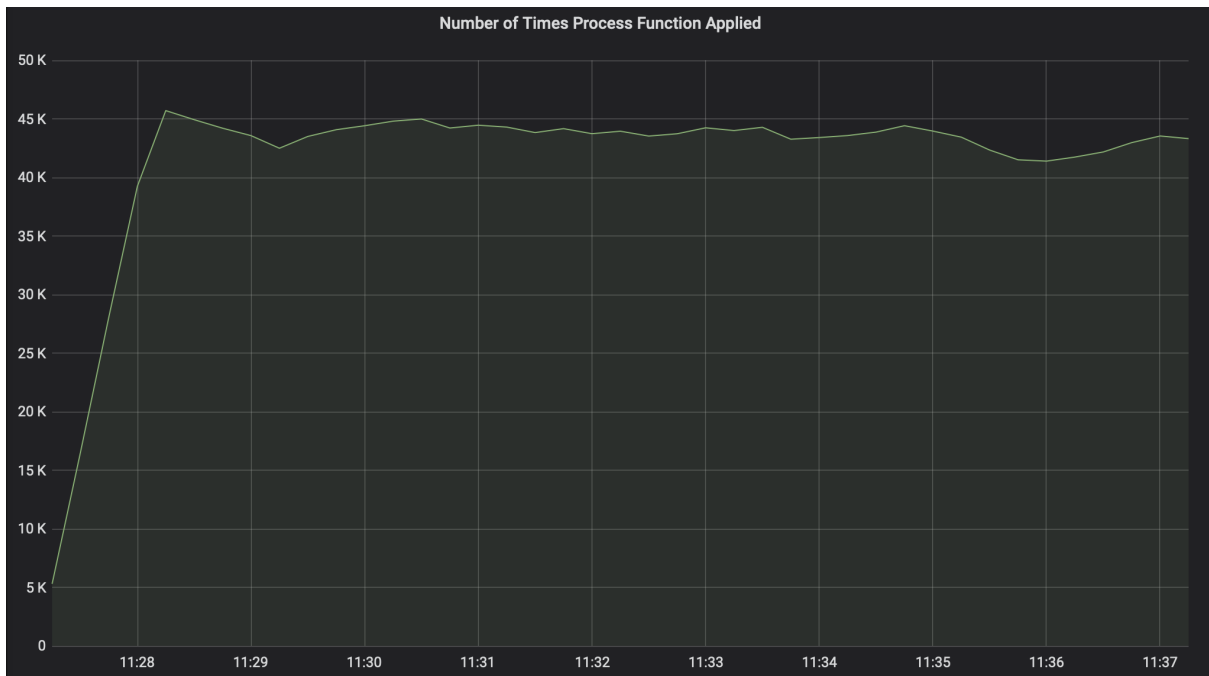


Figure 6.12: Events/s Using Slices / 100 Customers / AVG 6 Hours

To get more significant result the time between emissions is decreased to 1 minute as before. The achieved results are shown in 6.13. With 32'000 emitted events per second compared to the 35'000 before, this method is still a bit worse. The reason is the same as before. As long as the first method can hold all slices in its heap memory it processes very fast, even faster than this method.

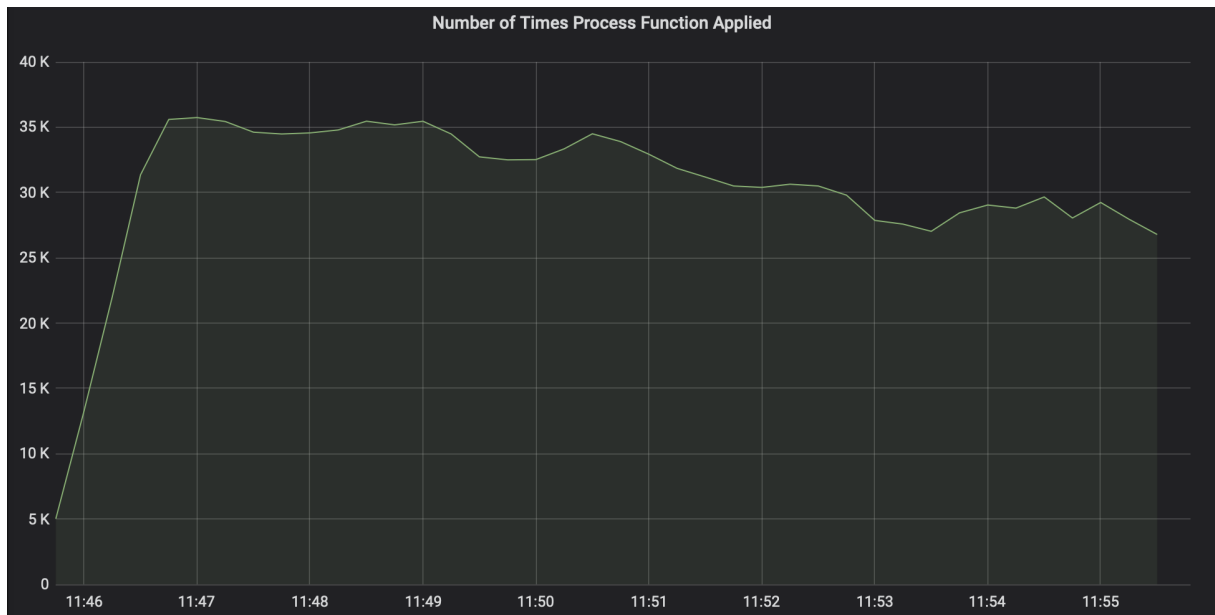


Figure 6.13: Events/s Using Slices / 100 Customers / AVG 1 Minute

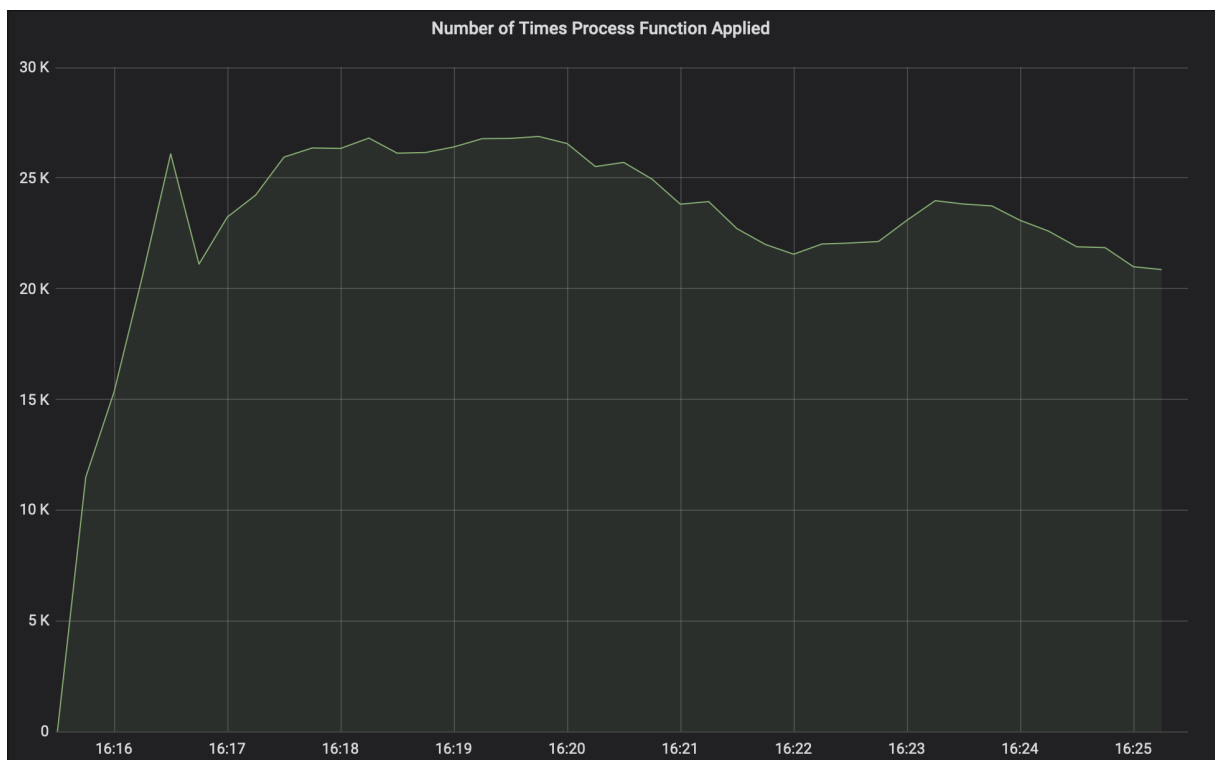


Figure 6.14: Events/s Using Slices / 1000 Customers / AVG 1 Minute

The configurations used in the first method, that lead to a slow down due to memory issues, will be used now to check if the second method still works faster. Figure 6.14 shows the processed events. Compared to the method without a window assigner the performance of this method stays constant considering the amount of processed events.

This is due to the fact that the method did not have to store all the events. However, the performance seems to decrease as well. Considering this, the problems from the previous method is only partially solved but in cost of some precision.

The evaluation of the aggregations showed that the methods work in principal. Nevertheless there are still open issues with scaling, that clearly need refinement.

Chapter 7

Summary and Conclusions

The overall task of this thesis was to implement a data preparation system for Fraud Scoring with Apache Flink. As a sub task, test data had to be generated which then in turn would be used for the aggregations made with Flink. This part was essential, as it was not possible to work with productive credit card data. Yet, a similar system load had to be generated to reliably test statements about whether the desired performance could be achieved or not. The test data should be generated according to a given statistical distribution, provided by Aduno Group [2], and show certain patterns. This was important because in Flink aggregations have to be made on certain keys (*e.g.*, card number, merchant, and region) and these aggregations may give distorted results without correlations between them. Furthermore, the system may behave unpredictable and possibly even inefficient on the productive environment due to bad partitioning or similar. An important focus was put on the simplicity and composability of the system, since the design of a correct model is later to be taken over by data scientists to refine the pattern. As a second part, a pipeline for the data preparation of authorizations for a fraud scoring machine learning model had to be developed. The model has been newly developed by the Aduno Group [2] in order to check the probability of credit card use as to whether it is fraud or not. The problem with the existing system is that it takes about an hour to process the data. For a system that operates in the millisecond range, this is naturally too much time. Accordingly, an optimization is to be carried out here with Flink. The goal is to aggregate the attributes required for scoring via tumbling and sliding windows and then make them available to the algorithm. As a third part, a reclassification of authorizations into other windows had to be made possible. This is necessary because authorizations are usually first marked as non fraud. Eventually after some time, there is the possibility that it is noticed that the card usage was actually fraud and therefore the data set has to be removed from all non fraud windows and integrated into the correct ones.

For the first part it can be stated that it was implemented according to the requirements and allows a statistically distributed data generation. In test scenarios, 8 million merchants and 5 million customers were generated, which finally resulted in a frequency of 50-70 thousand emitted events per second. A problem was found to be the memory consumption of individual objects which overload the garbage collector when a large number of entities are generated. Two optimizations can be applied for this. On the one hand,

the RBT data structure seems to produce a large overhead (700 bytes) when stored on the individual tree nodes. The planned solution for this looks as follows. A simple array structure could be used, which allows to store a data value and a corresponding weight. The structure should also store the total value of the weights. Now a random value between 0 and the maximum weight could be chosen. Afterwards, each element of the collection is it can be returned. This approach is similar to the stick variant for probabilities mentioned in Figure 4.2. With such a structure, most of the 700 bytes could be saved, but at the expense of computational operations. On the other hand, the size of the objects would still be quite large despite all the optimizations, such as the extraction of categorical attributes (*i.e.*, `mcc group` is currently a string, but could be mapped to an integer representation) to mappings. Therefore, serialization with the Flink serializer could be considered. Thus, a reduction of the effective object size could be achieved. This reduction of the object size in memory would make it possible to work with even more entities in the millions without any problems. This is also considered necessary, as it is not expected that the data scientists who will use this tool later on will perform low level optimizations to remain efficient. This is at least the practical experience that has been made. Nevertheless, to the best of the author's knowledge and belief, this is the first academic paper to address the deterministic random generation of statistically distributed events in a streaming environment.

The focus of the second part laid in finding a method to process the generated test data from the first part as fast and memory preserving as possible. Finally, two efficient methods were implemented that can process up to 45'000 events per second. Thus, the required performance can be fulfilled in principal. Nevertheless, the performance of the first method suffers a lot when the number of events in the input stream gets too big, as it needs to store all events in memory. Despite these problems, it can still process the input elements. To avoid the storage of each event and the resulting reduction of performance, the second method uses window slices, which will be evicted as a whole. Due that, less storage is needed and the performance is theoretically less reduced as the number of events per time window increases. As a final conclusion the elaborated methods are functioning, but they have to be improved in a way they scale better. Due to the focus on the first two parts and their given complexity, the third part of the project could unfortunately not be integrated into this work. It will therefore be dealt with in a follow-up project at Aduno, using the findings of this work.

7.1 Future Work

Future work will arise on the one hand in data generation. It must be possible to create and use collections of statistically distributed entities of any size. As a possible solution for this, the serialization of the individual objects was mentioned in the previous Section. It is also possible to replace the RBTs in the individual entities with lighter ones to reduce the overhead per object. Another point that can be considered as future work is a formal proof of the determinism given by the use of the Temporal Event monad. On the other hand, the aggregation of the data certainly needs to be improved as it has weaknesses in regards to performance and space complexity. Finally, as there was not enough time for the reclassifications, this has to be addressed in a future project.

Bibliography

- [1] R. R. Popat and J. Chaudhary, “A Survey on Credit Card Fraud Detection Using Machine Learning,” in *International Conference on Trends in Electronics and Informatics (ICOEI 2018)*, (Tirunelveli, India), pp. 1120–1125, May 2018.
- [2] Aduno Group, “Aduno Group - The Smart Way to Pay,” 2020. <https://www.aduno-gruppe.ch/en/>, Last visit April 21, 2020.
- [3] R. Nilsson, “ScalaCheck - Random Generators,” 2019. <https://github.com/typelevel/scalacheck/blob/master/doc/UserGuide.md> Last visit February 4, 2020.
- [4] J. Traub, P. M. Grulich, A. R. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl, “Efficient Window Aggregation with General Stream Slicing,” in *International Conference on Extending Database Technology (EDBT 2019)*, (Lisbon, Portugal), pp. 99–108, March 2019.
- [5] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier, “Hyperloglog: The Analysis of a Near-optimal Cardinality Estimation Algorithm,” in *International Conference on Analysis of Algorithms (AOFA 07)*, (Juan des Pins, France), pp. 127–146, June 2007.
- [6] Jessica Allen, “Ravelin data reveals one in five payments are lost through 3D Secure,” 2019. <https://www.ravelin.com/blog/one-fifth-of-payments-sent-to-3d-secure-are-lost> Last visit April 20, 2020.
- [7] L. J. Guibas and R. Sedgewick, “A Dichromatic Framework for Balanced Trees,” in *19th Annual Symposium on Foundations of Computer Science (SFCS 1978)*, (Ann Arbor, MI, USA, USA), pp. 8–21, October 1978.
- [8] J. Gall, *The Systems Bible: The Beginner’s Guide to Systems Large and Small*. General Systemantics Press, 2002.
- [9] D. J. Simms, “Roger godement, topologie algébrique et théorie des faisceaux, vol. i, actualités scientifiques et industrielles 1252, publications de l’Institut de mathématique de l’université de strasbourg xiii (hermann, paris), 283 pp., 3600 francs,” *Topologie Algébrique et Théorie des Faisceaux*, vol. 11, no. 3, p. 189, 1959.
- [10] James Iry, “A Brief, Incomplete, and Mostly Wrong History of Programming Languages,” 2009. <http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html> Last visit April 20, 2020.

- [11] T. Petricek, “What we talk about when we talk about monads,” *The Art, Science, and Engineering of Programming*, 2018. <https://arxiv.org/ftp/arxiv/papers/1803/1803.10195.pdf> Last visit April 20, 2020.
- [12] P. Wadler, “Monads for functional programming,” in *Program Design Calculi* (M. Broy, ed.), (Berlin, Heidelberg), pp. 233–264, Springer Berlin Heidelberg, 1993.
- [13] TypeLevel, “Typelevel.org - Cats,” 2019. <https://typelevel.org/cats/> Last visit April 20, 2020.
- [14] Y. Yang, “Using agent-based modeling to study multiple risk factors and multiple health outcomes at multiple levels: 3m study and agent-based modeling,” *Annals of the New York Academy of Sciences*, vol. 1408, pp. 7–14, 11 2017. <https://doi-org.ezproxy.uzh.ch/10.1111/nyas.13558> Last visited April 20, 2020.
- [15] J. Lagorse, D. Paire, and A. Miraoui, “A multi-agent system for energy management of distributed power sources,” *Renewable Energy*, vol. 35, no. 1, pp. 174 – 182, 2010.
- [16] Ververica, “What is stream processing?,” 2020. <https://www.ververica.com/what-is-stream-processing> Last visit April 27, 2020.
- [17] The Apache Software Foundation, “Apache Flink - Stateful Computations over Data Streams,” 2019. <https://flink.apache.org/> Last visit February 4, 2020.
- [18] T. A. S. Foundation, “Windows,” 2020. <https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/stream/operators/windows.html> Last visit April 27, 2020.
- [19] T. A. S. Foundation, “Working with state,” 2020. <https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/state/state.html> Last visit April 27, 2020.
- [20] T. A. S. Foundation, “Event time,” 2020. https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/event_time.html Last visit April 27, 2020.
- [21] S. Ewen, “Advanced Streaming Analytics with Apache Flink and Apache Kafka,” 2016. <https://www.slideshare.net/ConfluentInc/advanced-streaming-analytics-with-apache-flink-and-apache-kafka-stephan-ewen>, Last visit April 21, 2020.
- [22] S. Gupta and S. Pujari, “A multi-agent system (mas) based scheme for health care and medical diagnosis system,” in *2009 International Conference on Intelligent Agent Multi-Agent Systems*, pp. 1–3, 2009.
- [23] W. Lee, S. Cho, P. Chu, H. Vu, S. Helal, W. Song, Y.-S. Jeong, and K. Cho, “Automatic agent generation for iot-based smart house simulator,” *Neurocomputing*, vol. 209, 06 2016.
- [24] P. Grulich, J. Traub, S. Bress, A. Katsifodimos, V. Markl, and T. Rabl, “Generating reproducible out-of-order data streams,” in *Generating Reproducible Out-of-Order Data Streams*, pp. 256 – 257, 06 2019.

- [25] Y. Yue, Y. Li, K. Yi, and Z. Wu, “Synthetic data approach for classification and regression,” in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 1–8, 2018.
- [26] J. Traub, P. Grulich, A. R. Cuéllar, 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)*, 2019. <https://www.user.tu-berlin.de//powibol/assets/publications/traub-efficient-window-aggregation-with-general-stream-slicing-edbt-2019.pdf> Last visit April 20, 2020.
- [27] StackOverflow Community, “Random number with Probabilities,” 2015. <https://stackoverflow.com/questions/20327958/random-number-with-probabilities> Last visit February 4, 2020.
- [28] StackOverflow Community, “Random number with Probabilities.” <https://stackoverflow.com/questions/24869304/scala-how-can-i-generate-numbers-according-to-an-expected-distribution> Last visit February 4, 2020.
- [29] P. Pebay, T. Terriberry, H. Kolla, and J. Bennett, “Formulas for the computation of higher-order central moments,” in *Formulas for the Computation of Higher-Order Central Moments*, 2016.
- [30] www.geeksforgeeks.org - ChiragAcharya, Preet Dabre, “Red-Black Tree | Set 2 (Insert),” 2019. <https://www.geeksforgeeks.org/red-black-tree-set-2-insert/?ref=1bp> Last visit April 20, 2020.
- [31] VirtusLab, “Red-Black Tree | Deletion,” 2018. <https://github.com/VirtusLab/scalastrap/blob/master/corpus/scala-library/src/scala/collection/immutable/RedBlackTree.scala> Last visit April 20, 2020.
- [32] S. Kahrs, “Red-black trees with types,” *J. Funct. Program.*, vol. 11, no. 4, pp. 425–432, 2001.
- [33] C. Okasaki, “Red-black trees in a functional setting,” *Red-black trees in a functional setting*, vol. 9, no. 4, pp. 471–477, 1999. <https://wiki.rice.edu/confluence/download/attachments/2761212/Okasaki-Red-Black.pdf> Last visited April 20, 2020.
- [34] Scala, “Scala redblack,” 2011. <https://github.com/scala/scala/blob/v2.9.3/src/library/scala/collection/immutable/RedBlack.scala> Last visit April 27, 2020.
- [35] Scalacheck, “ScalaCheck: Property-based testing for Scala,” 2019. <https://www.scalacheck.org/> Last visit April 20, 2020.
- [36] Typelevel, “Io monad,” 2019. <https://typelevel.org/cats-effect/datatypes/io.html> Last visit April 27, 2020.

- [37] Typelevel, “Trampoline,” 2019. <https://github.com/typelevel/cats-effect/blob/master/core/shared/src/main/scala/cats/effect/internals/Trampoline.scala> Last visit April 27, 2020.
- [38] Typelevel, “Freemonad,” 2019. <https://typelevel.org/cats/datatypes/freemonad.html> Last visit April 27, 2020.
- [39] Will Koehrsen, “The Poisson Distribution and Poisson Process Explained,” 2019. <https://towardsdatascience.com/the-poisson-distribution-and-poisson-process-explained-4e2cb17d459> Last visit April 20, 2020.
- [40] Wikipedia, “Poisson point process,” 2019. https://en.wikipedia.org/wiki/Poisson_point_process Last visit April 27, 2020.

Abbreviations

MCC	Merchant Category Code
Auth	Authorisation
Trx	Transaction
CardId	Card Identification Number
ABM	Agent-Based Modelling
MAS	Multi-Agent System
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

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 can be reserved.

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

Flink Stream Processing Framework used for this project.

Red Black Tree Self-balancing subtype 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.

Gen Monad that handles the generation of distributed variables

TE Temporal Event Monad that handles modelling and progression of time, sampling of distribution variables, the emission of streaming events, concurrency and synchronization as well as global state and the handling of side effects.

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

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.

List of Figures

2.1	Skewed Binary Search Tree	8
2.2	Same Values as in 2.1 but With Red Black Tree Algorithm	8
2.3	One Input Stream [17]	18
2.4	Keyed Stream by the User Id as shown by [18]	20
2.5	Tumbling Windows shown by [18]	21
2.6	Sliding Windows [18]	22
2.7	Watermark for a Stream in Order as Shown by [21]	24
2.8	Watermark for a Stream Out of Order [21]	24
3.1	Aggregation of Tuples Into Slices as Proposed by [26]	28
4.1	Dataflow Diagram of the Data Preparation for Fraud Scoring	30
4.2	Metaphorical Stick With Intervals	32
4.3	Metaphorical Stick With Summed Probabilities	32
4.4	Example of a Scala TreeMap With Weights as Keys	33
4.5	Example of a Tree With an Aggregated Weight of 60 in the Root Node . .	35
4.6	Diagram of a Customer and its Lifetime Cycle	37
4.7	Sliding Window With a Window Size of 5 Seconds and one Slide per Second	41
4.8	Only an Evictor, Which Deletes non Relevant Transactions	42
4.9	Global Window Split in Moving Slices	44
4.10	Global Window Split in Fixed Slices	45
5.1	Visualization of a Left Uncle Red Case	51

5.2	Visualization of the Problem When Using Range Queries	54
5.3	Visualization of a CDF of an Exponential Distribution	68
6.1	Comparison Between Tree Size and Time to Search for a Key	88
6.2	Comparison Between Tree Size and Time to Search for a Prefix	88
6.3	Comparison Between Tree Size and Time to Add a Node	89
6.4	Comparison Between Tree Size and Time to Delete a Node	89
6.5	Exponential Growth of Time it Takes to Generate the Tree	90
6.6	Weights of RBT Nodes	91
6.7	Space Requirements of a Merchant	92
6.8	Events/s of the Test Data Generator	93
6.9	Events/s No Window Assigner / 100 Customers / AVG 6 Hours	96
6.10	Events/s No Window Assigner / 100 Customers / AVG 1 Minute	96
6.11	Events/s No Window Assigner / 1000 Customers / AVG 1 Minute	97
6.12	Events/s Using Slices / 100 Customers / AVG 6 Hours	98
6.13	Events/s Using Slices / 100 Customers / AVG 1 Minute	99
6.14	Events/s Using Slices / 1000 Customers / AVG 1 Minute	99

Listings

2.1	Map Example	10
2.2	Difference Map and FlatMap	10
2.3	Left Identity Example	12
2.4	Right Identity Example	12
2.5	Associativity Example	12
2.6	Output of a Map Operation	13
2.7	Definition of the ADT	14
2.8	Definition of the ADT Type	14
2.9	Lifting of Functions Into the Monad	14
2.10	KVStore Programm	15
2.11	FreeMonads Compiler	15
2.12	General Structure of a Flink Program	19
4.1	Probabilities With If Else Structure	31
4.2	Prefix Algorithm	34
5.1	Basic Structure of a RBT Node	50
5.2	Addition of a Node	51
5.3	Left Balancing of an Uncle Case Node	52
5.4	Deletion From an RBT	53
5.5	Core Method of the Range Queries	54
5.6	Rebalancing of Range Queries	55
5.7	Search For Key Algorithm	56
5.8	Search For Prefix Algorithm	57
5.9	Gen Monad Examples	59
5.10	Sleep Model Time	60
5.11	Enqueueing Continuations With Sleep	60
5.12	Implementation of Pure	61
5.13	Implementation of Now	61
5.14	Implementation of Delay	61
5.15	Implementation of Gen	61
5.16	Implementation of SetSeet	62
5.17	Implementation of Emit (Simplified)	62
5.18	Implementation of PutToStateBag	63
5.19	Implementation of GetFromStateBag	63
5.20	Implementation of Effect	63
5.21	Implementation of Concurrent	64
5.22	Loading a Tree From a CSV	65
5.23	Creation of a Merchant Tree	66

5.24	Creation of a Merchant	67
5.25	Called Functions When Using Sample	69
5.26	PureApply That Allows For Determinism	69
5.27	Generation of Datastructures and Customers	70
5.28	Start of a Customer Lifetime and Card Usage	71
5.29	Card Usage Pattern, Simple Case	72
5.30	DistributionStatisticsOverPeriod	73
5.31	DistributionStatistics	73
5.32	Flink Program With no Window Assigner	74
5.33	GetStatisticsOverTimeProcessFunction	75
5.34	CalculateStatisticsFixedSlicingWindows	77
5.35	SliceMomentumStatistics	77
5.36	Flink Program to Create the Window Slices	78
5.37	AggregateFunction to Apply Before the ProcessWindowFunction	79
5.38	ProcessWindowFunction of the AggregateFunction	79
5.39	Flink Program to Aggregate the Window Slices	80
5.40	Evict Method of OldSlicesEvictor	81
5.41	ProcessWindowFunction to Combine the Remaining Relevant Slices	82
5.42	Structure of the Main Method of the Flink Program	83
5.43	Creating the test input stream of authorisations	84
6.1	Height Attribute in Tree Node	86
6.2	Test Case to Check The Maximum Size of a Tree	86
6.3	Generator That Creates a Tree With a Certain Number of Nodes	87
6.4	Aggregation of Generated Test Data to Prove Determinism	94

Appendix A

Contents of the CD

The attached CD contains the following files and directories:

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