Go with the Flow

Exponential Decaying Reservoir Sampling of Evolving Data Streams

Master Thesis

First Examiner: Prof. Dr. Stefan Decker
Second Examiner: Prof. Dr. Bernhard Rumpe
Supervisor: Dr. Michael Cochez

Georg Groß

Computer Science 5 - Information Systems
RWTH Aachen University

Aachen, 4. März 2018

Abstract

This thesis will present an analysis of a previously proposed method for biased reservoir sampling of evolving data streams. As biased sampling methods will allow to follow the recent developments of the observed data stream more closely, a wide range of applications exists. The method involves random influenced item weights and is built upon an exponential decay for element inclusion. Furthermore, the method will enable event time based sampling and can perform sampling in a distributed environment. This is an important property because more and more scenarios rise up that share such a requirement. For comparison, a theoretical analysis and a experimental evaluation are performed. The thesis will close with an application of the maintained sample on the stream processing framework Apache Flink. Herein, the biased sampling method will be used in order to implement a stream processing frequent pattern mining algorithm in order to retrieve the frequent patterns from a real-world StackOverflow data set.

Zusammenfassung

Diese Arbeit stellt eine Analyse, einer kürzlich vorgestellten Methode zur kontinuierlichen Extraktion einer Stichprobe eines Datenstroms dar. Die erhobene Stichprobe hat dabei die Eigenschaft, dass sie kürzlichen ”Events” des Datenstromes eine höhere Zugehörigkeit zur Stichprobe zuschreibt als älteren. Durch diese zusätzliche Eigenschaft eignet sich die gewonnene Stichprobe vor allem für Analysen des aktuellen Verhaltens des Datenstroms.
Um diese Eigenschaft zu garantieren, wird jedem Element innerhalb der Stichprobe ein Gewicht zugeschrieben, welches einem exponentiellen Verfall folgt. Desweiteren ermöglicht diese neue Methode eine verteilte Erhebung der Stichprobe im Netzwerk sowie die Erhebung einer Stichprobe eines ”event time” basierenden Datenstroms. Um die gewonnen theoretischen Resultate genauer evaluieren zu können, wurde ein Benchmark durchgeführt und die Resultate mit bestehenden Verfahren verglichen. Die Arbeit schließt mit einer Anwendung der gewonnen beeinflussten Stichprobe auf ein Data Mining Szenario innerhalb des Datenstrom Frameworks Apache Flink.

Contents

1 Introduction

I got this strange idea that maybe I could study the Bible
the way a scientist would do it, by using random sampling.
The rule I decided on was we were going to study
Chapter 3, Verse 16 of every book of the Bible.
This idea of sampling turned out to be
a good time-efficient way to get into a complicated subject.
— Donald Knuth (2008)

Due to the increasing amount of data generated on different sources, efficient data processing and analysis techniques become more and more important. Especially in the context of Big Data, efficient algorithms are required. Nevertheless, in some cases even efficient algorithms are performing too slow, because of the sheer amount of data. Therefore, the size of the input has to be reduced in order to gain results in a fair amount of time. Hereby, *sampling* is a well known technique for reducing the input size without changing the probabilistic occurrence of each individual data point [**aggarwal2007data**]. It is very important for such sampling methods that all elements within the sample are a 'representative' of the elements within the whole data set [**gama2010knowledge**]. This property needs to be preserved, if not any analysis on top of the taken sample would be faulty and will not provide any clear insight to the original data set.

Given a fixed-sized data set with length N, finding such a sample, in which the inclusion probability for every item is uniform, is a trivial task. In fact, the inclusion probability for an element in the data set to be present in the obtained sample would be equal to n/N, resulting in a sample of size n.

However, it is not possible to derive this inclusion probability directly, if this size of the data set N is not known beforehand or even continuously increasing. Such a restriction has to be considered if one wants to obtain a sample of a data stream. As more and more data is available, or will become available, in such a form of a continuous stream of data, stream processing methods become an important research area. Especially if one considers new applications, like in the Internet of Things domain with its small sensor nodes. These nodes can be deployed distributively and will deliver fast forwarding measurements of their observed surrounding. A practical example for such an application can be named in field of modern power grid monitoring. Modern power grids are continuously monitored for current load and

demand in order to provide fast response and to prevent blackouts. Analysis of such occurring blackouts showed, that one major cause of these event were lack of real time situation awareness [**hazra2011stream**]. In such a scenario, data stream processing methods can provide fast insights and encourage quick response to the observed events. Because of the special characteristics of a data stream, processing methods that require random access or a fixed sized input can not be applied directly. Either specialized methods have to be conducted or a process of sampling can be performed in order to provide a snapshot, which follows the observed data stream. Furthermore, also the specialized approaches often perform some kind of sampling internally, in order to cope with the characteristic of a data stream. This shows the importance of sampling methods for data streams.
For this case, different methods have been proposed: The most dominant approaches are *Windowing* and *Reservoir Sampling*. These methods keep a distant history of data points in the data stream in an unbiased way, on which further analysis can be performed.
However, for some occasions such a uniform sampling behavior is not suitable. As data streams often show the characteristics of some time-dependency, this property might also to be considered during the process of sampling. The resulting sample representation of the data stream one may want to obtain should therefore provide recent observations a higher attribution to be present in the sample as older ones and hence a bias has to be introduced to the sampling procedure. A popular example for such a requirement is performing trending topics analysis on a data stream. As recent topics of the observed stream are more important to derive such trending events, the sampling process should adapt to this requirement. In order to consider such a prioritization of recent elements, further methods for data stream sampling have been proposed that introduce such a bias to the sampling procedure. However the bias, which is introduced to the sampling scenario has to be well understood in order to obtain the requested behavior of the sample in the end.
In general the bias is introduced using a decaying probability of element inclusion within the sample over stream progression. With the help of such a biased representation of the sample further scenarios can be conducted, which will not be able to perform using a uniform sample representation of the data stream alone. Especially if one considers analysis on recent developments within the data stream.

Therefore, data stream sampling methods (uniform or non-uniform) are an important part for data stream processing and assist to perform further analysis of stream development, in a reasonable amount of time.

Course of this thesis

This thesis will demonstrate the well known methods of data stream sampling and also introduce its biased adaptations in the beginning. After this introduction to the background of data stream sampling a comprehensive analysis of a recently developed method called

Exponential Decaying Sampling will be performed. This method will generate a biased sample of a data stream, following an exponential decay of inclusion for elements within the sample over stream progression. As part of this analysis, the question of how the method behaves during the process of sampling and how it will compare to the existing approaches will get answered. In order to provide such a comparison, the properties of the maintained sample need to be evaluated and an analysis of the involved computational complexity for the process of sampling has to be performed as well. Additionally to the theoretical analysis, a benchmark of the proposed sampling method will be performed in order to confirm those results in a practical implementation. The benchmark is followed by a discussion of more advanced data stream sampling scenarios: Considering the process of sampling in an event time based data stream, respecting out-of-order arrivals. Furthermore, an application within a distributed environment will also be possible using the new approach.
As an application, in which the new method can be applied, a distributed decaying frequent pattern mining algorithm will be implemented within the Apache Flink stream processing system. The derived patterns will include the bias, towards more recent elements, and will also follow the stream progression. This approach was applied to a real world data set consisting of posts of the StackOverflow discussion board.

2 Sampling of Data streams

This section will provide an introduction to the important concepts of data streams in general and sampling of those in particular. The basic characteristics of a data stream and its special properties need to be understood in order to recognize the importance of sampling in such a scenario. This chapter will also introduce the most dominant sampling methods of data streams, including algorithms which respects the underlying time dependency in any data stream scenario.

2.1 The Data Stream Model

Different from traditional database system processing, streams of data reveal some additional challenges. Those induced problems of the model are not existing for a sampling scenario exclusively, moreover they will be present in any data stream processing scenario. In a data stream elements arrive in a continuous flow and are typically not available for random access. Therefore, four main differences and resulting problems in contrast to the traditional database processing can be concluded, that have to be considered for data stream processing [**babcock2002models**]:

1. Elements in the stream arrive in an online setting
2. Elements can arrive in a non-ordered sequence, furthermore does the processing system have control over the ordering
3. Elements can arrive infinitely, resulting in an unbounded sized data stream
4. Elements have to be processed directly after arrival - if they are not stored explicitly, no further access to the element can be made afterwards

For this reason, stream processing methods and algorithm have to include special properties in order to be applicable in such a scenario. As consequence of the potential unbounded size of the stream, maintaining the stream as a whole and performing processing on the gathered data set will not be feasible. Therefore, algorithm maintain a "sketch" of the underlying data stream and update this model continuously with stream progression. For such a model A, three generic types are existing in literature [**garofalakis2016data**]:

Time-Series Model:
In a Time-Series model, for any new arriving data point a_i the model is updated in increasing order of i, resulting in a new representation of $A[i]$. This naturally models the time dependency of the data stream, and is therefore applicable for scenarios like temperature measurements or network traffic volume at certain time periods. It has to be noted, that such a model prohibits the update from past (lower value of current i) entries.

Cash-Register Model:
In contrast to the Time-Series model, Cash-Register models allow also changes to values, which arrived before the current noticed i-th element. For every update, the only restriction of the model is that at an arbitrary position only incrementations can be performed. Therefore, this model is suitable for performing accumulation over time such as in web server access monitoring for specific IP addresses for example.

Turnstile Model:
In the Turnstile Model, no restrictions are made regarding the update of the underlying model. Therefore, it is the most general case to symbolize a fully dynamic streaming situation. For example, monitoring current connections to a web service for a given IP range can be performed with the help of such a model, as connection can be initiated and dropped dynamically.

As the Turnstile model is the most general one, any method that embeds this model will also be able to be applicable to any other scenario. Rather than maintaining such a model of a data stream internally in the algorithm, sampling of the data stream can also be performed in order to generate a smaller extract of the transmitted entries. Such a generalized method enables any further processing of the data, because the maintained sample is limited in size and is available for random access. Also, the process of keeping a sample helps in order to estimate many different statistics and additional help to form answers to non-aggregate queries [**muthukrishnan2005data**]. However, the key task of the sampling method of the data stream is to follow the stream continuously, while forming a smaller sized representative for the original transmitted entries. Therefore, their distribution should not be manipulated and possible changes in the stream should be followed. However, the discussed differences to a traditional data storage also apply to the task of sampling the data stream. Especially the unbounded nature of a data stream can render performing an uniform sampling process a complicated task.

For this reason the next section will introduce methods for generating such a sample of a continuously evolving data stream and proof that their underlying distribution will not be

affected.

2.2 General Sampling methods

Reservoir Sampling

Reservoir algorithms are randomized algorithms, which key task is to generate a list (with at least length n) from which a sample of length n can be extracted. According to Vitter [**vitter1985random**] the following has to hold for any reservoir algorithm:

> An algorithm is a reservoir algorithm if it maintains the invariant that after each record is processed a true random sample of size n can be extracted from the current state of the reservoir.

As reservoir sampling is a one-pass process and the length of the input can be unknown, it is well suited for data stream sampling. Additionally, it assures us, through its invariant, that the generated sample is a 'representative' for its original source. The way the algorithm achieves this property is by using a continuously decreasing input probability.
It is best to see this on an example:

Assuming, we have a reservoir of size one and want a uniform sample of the incoming data stream. Therefore, a reservoir sampling approach would be:

- Fill up the reservoir with our first item
- For the next item i:
  - Replace the item in the reservoir with the new item with probability $1/i$
  - Keep the existing item and ignore the new one with probability $1 - 1/i$

This approach is trivial to implement and would generate a sample of size one, but it remains to show that the important invariant is respected. By induction this will follow as:

Algorithm 1: Algorithm R: reservoir sampling [**vitter1985random**]

```
Data: stream S[1...n]
Result: reservoir R[1...k]

// fill up reservoir
for i = 1 to k do
    R[i] := S[i]
end
// sample elements with decreasing probability
for i = k +1 to n do
    j := random(1, i)
    if j ≤ k then
        R[j] := S[i]
    end
end
```

Proof:

- If there is only one element within the stream, the probability that this item will be in the stream is one
- If the data stream consists of two elements, the probability that each item will be selected for the sample is $1/2$
- Assuming, this holds for a stream of size n and for any given element x, the probability that x will be in the sample is $\frac{1}{n}$ by induction.
- Now suppose a new element y is appearing:
  - This $(n+1)$-th element will be kept with probability $\frac{1}{n+1}$
  - Any element x before y will be kept with probability $\frac{1}{n} * (1 - \frac{1}{n+1}) = \frac{1}{n} * \frac{n}{n+1} = \frac{1}{n+1}$

Therefore, this approach will create a uniform sample from the data stream, as each element in the stream has equal probability of inclusion.

Generating a sample from the data stream using this approach is efficient, but one may maintain a larger sample as with a size of one. For this reason [**vitter1985random**] proposed an algorithm, using the above principle, for any reservoir size $k \in \mathbb{Z}_{>0}$.

First of all, the algorithm will populate the reservoir until all slots are used and then starts to begin replacing elements within the reservoir with continuously decreasing probability in order to achieve an uniform sample R with size of k. This decreasing probability is implemented using a random variable j, within the range 1 to the current number of the new data point i. With more and more incoming data points, this range will increase and therefore the

probability that $j \leq k$ is fulfilled will decrease. Hence, the probability that a new element i will be added to the reservoir is $\frac{k}{i}$ and $\frac{1}{k} * \frac{k}{i} \Leftrightarrow \frac{1}{i}$ for an existing item in the reservoir that this element will be replaced, in order to include the new element.
One can show that each item within the reservoir will have equal probability (k/n) of being selected for the reservoir using a similar approach like the one for the one-size reservoir.
Because of our assumption all $i-1$ elements are in the reservoir with probability $\frac{k}{(i-1)}$. As the probability for a replacement of an existing element in the reservoir is $\frac{1}{i}$, its probability of remaining there is $1 - \frac{1}{i} \Leftrightarrow \frac{(i-1)}{i}$. Therefore, multiplying both probabilities would result in the total probability for any of the $(i-1)$ elements to be in the reservoir after sampling the i-th element: $\frac{k}{(i-1)} * \frac{(i-1)}{i} = \frac{k}{i}$. As those results hold for any value of i, the proof follows by induction. For a stream of length n, all elements will therefore be in the reservoir with fixed probability $\frac{k}{n}$ at the end.

The Algorithm R reservoir sampling procedure will generate a random uniform sample from a given data stream at runtime cost of $\mathcal{O}(n)$ with constant cost for sampling any new occurring data point of the stream. Therefore this approach is quite efficient, while being also implementable intuitively. However, there are optimized adaptions from this Algorithm existing, namely Algorithm X, Algorithm Y and Algorithm Z. These methods remove the need to perform an inclusion check for every new occurring element but rather generate a randomly distributed skip variable. Using those skips, not every new occurring elements has to be considered for inclusion and therefore the complexity for sampling can be improved towards $\mathcal{O}(s + log(\frac{n}{s}))$ with s equal to the size of the maintained reservoir for Algorithm Z [**vitter1985random**].

Bernoulli Sampling

A sampling scheme, which can also be applied in context of a data stream is the so called Bernoulli sampling method. In a Bernoulli sampling procedure, each new occurring element is appended to the reservoir with a fixed probability $q \in (0,1)$, called sampling rate. This fixed probability will enforce that each element will have the same probability of inclusion, and results therefore in a uniform sample. Such an approach is easy implemented and even optimized approaches exist, that replace the expensive computation of a random number at every sampling step with a initially randomly determined sampling gap [**haas2016data**]. As a consequence of the sampling scenario the resulting sample will not be of any fixed size. Nevertheless, because of the underlying binomial distribution of a probability that a sample will contain exactly k elements can be given with its probability mass function: $\binom{n}{k} q^k (1-q)^{n-k}$ and the expected size of the sample will be nq.
However, for an unbounded data stream, the resulting sample of the Bernoulli sampling method will also be unbounded in size and therefore additional measures need to be under-

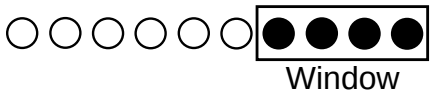

Figure 1: Schematic drawing of a sliding window

taken to restrict the sample size. An example for such a restriction is an additionally performed windowing over the Bernoulli sample. Nevertheless, because of its known properties and intuitively implementation Bernoulli sampling has its reason for existing, especially for bounded data streams. Bernoulli sampling can also be computed distributed and in parallel without large efforts.

2.3 Incorporating Recency in the Data Stream

Windowing Techniques

Additional to Reservoir sampling and Bernoulli sampling approach, sampling a potentially infinite data stream using a window technique is also often performed. Similar to the Reservoir sampling method, such a window technique will often generate a fixed sized sample of the data stream, depending on the used implementation. This window will evolve similar to the data stream with stream progression.
In general one can differentiate between several types of windows for data stream sampling:

Sliding Windows

Sliding Windows are the most dominant implementation for a windowing of a data stream. Within the Sliding Window approach the sample is constructed using a fixed sized (n) window, which continuously 'slides' over the evolving data stream. Every new data point on the stream will be added to this window and will at least pop out after n additional time steps. Figure 1 shows a schematic drawing of a sliding window of size 4. The implementation of such a window is simple, but choosing the right window size n can be hard. Small values of n will follow the stream directly but will also be very effected to noise. Larger window sizes, however, will reduce this noise influence but will also deliver a wider range of historic values. For this reason additional to the *fixed sliding windows* also *adaptive windows* are existing [**bifet2007learning**].

Landmark Windows

In contrast to the Sliding Windows, in which the oldest entry of the window will move forwards with stream progression Landmark Windows define a fixed starting point of the window. From now on all points, that occur after such a *landmark* will be included in the window. For this

reason Landmark Windows are potentially not suitable for an infinite sized data stream sampling scenario. Nevertheless, applications exist, that require a definition a fixed starting point of sampling.

Damped Windows

Damped Windows introduce an additional weight for each element it contains. During the sampling process, the Damped Window will prioritize recent values in the stream before older ones. Therefore, the weights symbolize element arrival that will decay over time.

Without respecting the special cased Damped Window approach, windowing will provide an efficient but potentially restricted and aggressive approach to data stream sampling, as windowing rely on a hard boundary of element inclusion. Hence, its primary focus lies on keeping a recent extract of the stream and can not be applied if some further history or developments of the streams needs to be retained within the sample. For this reason, window based sampling is not applicable to all use case scenarios.

Biased Reservoir Sampling

With the previously presented window based sampling approaches a method was demonstrated, which respects the time dependency of the data stream within the sample by only taking recent data points of the stream into account.
This behavior is sometimes very important for further analysis on top of the generated sample. The demonstrated reservoir sampling method from Section 2.1 does not respect such time information, however an alternate approach exists to generate such a biased sample.

The bias one wants to achieve will have to give recent data points a higher probability of being represented in the sample at some time of sampling t. In the approach of [**aggarwal2006biased**] this is accomplished by introducing an exponential bias function $f(r,t)$. The bias function is proportional to the probability of the r-th data point belonging to the sample at time of arrival of point t. If the bias function is modelled in such a way, that it is monotonically decreasing with t (for any fixed r) and monotonically increasing with r (for any fixed t) the desired behavior of a bias towards more recent element in the reservoir will follow. It has to be noted, that it is still an open question if reservoir maintenance can be achieved in one-pass using arbitrary bias functions. Nevertheless, for the proposed exponential bias function it is possible to perform an one-pass sampling.
Specifically, the proposed exponential bias function looks like the following:

$$f(r,t) = e^{-\lambda(t-r)}$$

with a given bias rate $\lambda \in [0,1]$

The proposed exponential bias function belongs to the class of *memory-less* functions, which symbolizes those functions, where the future probability of retaining a current point in the reservoir is independent of its past history or arrival time.

With inclusion of a bias function to the sampling process another interesting property can be observed. The bias function enforces a maximum requirement on the required sample size, without any additional enforcement.

Theorem 1. *[**aggarwal2006biased**] The maximum reservoir requirement R(t) for a random sample from a stream of length t which satisfies the bias function f(r,t) is given by:*

$$R(t) \leq \sum_{i=1}^{t} \frac{f(i,t)}{f(t,t)}$$

It should be noted, that as $f(r,t)$ is monotonically increasing and $i \leq t$ the fraction is at most one, and therefore the maximum reservoir requirement is often significantly less than the number of points in the stream.
Applying this to the the exponential bias function, the maximum reservoir requirement for a stream of length t is bounded by the constant $1/(1-e^{-\lambda})$, which can further be approximated by $1/\lambda$.

Algorithm 2 will implement a biased reservoir sampling procedure, using the proposed exponential bias function. Given a reservoir of size $n = 1/\lambda$ and an additional function $F(t) \in [0,1]$, providing the fraction how much the reservoir has been filled at the time of arriving of data point t. It is noticeable that this algorithm is efficient to implement and has no greater complexity as the basic reservoir sampling algorithm. Additionally, one can recognize that the reservoir will fill up fast in the beginning but, with increasing value of $F(t)$, this behavior will decay until finally only replacements will be performed and the reservoir size will not increase any further.

In order to see that the proposed algorithm will implement an exponential bias sampling the following proof has to be provided:

Algorithm 2: Aggarwal Bias Reservoir sampling [**aggarwal2006biased**]

Data: data point i
Result: sample R[1...k]

```
j := random(0 ,1)
if j ≤ F(t) then
    // replace element in reservoir
    p := random(0, k)
    R[p] = i
end
else
    // append to reservoir
    n := size(R)
    R[n] = i
end
```

*Proof [**aggarwal2006biased**]:*

- The algorithm basically splits into two parts, either we are replacing an element within the reservoir or we simply append it:
  - Appending behaves in a deterministic way and will not be from further interest once the reservoir is full.
  - For the replacement part we flip a coin with success probability of $F(t)$, if we are successful we replace any of the existing elements in our reservoir with size $n * F(t)$. As the probability that an element will be selected in that case is uniform it is given by $\frac{1}{n*F(t)}$. Therefore, the probability for a replacement at the time point of arrival of data point t is equal to $F(t) * \frac{1}{n*F(t)} = \frac{1}{n}$
  - For any point in the reservoir r it will stay until some point t if it will not be replaced in any of the $t - r$ steps. As the probability of a replacement in one step was $\frac{1}{n}$, the probability that the data point r will stay is given by: $(1 - \frac{1}{n})^{t-r}$, which is equal to $((1 - \frac{1}{n})^n)^{(t-r)/n}$. With the help of an approximation for large values of n the term $(1 - \frac{1}{n})^n$ is equal to $1/e$. Therefore, one can substitute the term to $e^{-(t-r)/n}$ and as $n = 1/\lambda$ the exponential bias function $e^{-\lambda(t-r)}$ will follow.

Sampling with higher space constraints

The property of retaining a maximum requirement on the sample of the proposed approach might not be sufficient for some situations as the required sample size would be still too large to maintain. For this reason, a modified algorithm exists, which imposes an additional

probability p_{in} that an item will be considered to be added to the reservoir. Therefore, the maximum reservoir requirement reduces to $n = p_{in}/\lambda$ and the algorithm will behave exactly as the standard reservoir algorithm, while inducing the same exponential bias. However this additional uniform insertion probability, will induce one important difference to the original procedure:

The time steps needed until our reservoir will be filled up completely with elements will be by far longer, depending on the setting of the variable p_{in}. One possible countermeasure against this behavior could be to initial set the insert probability in the beginning to 1 and slowly decay this until the requested probability is reached. With this procedure the reservoir will still take more steps until the requested size is reached, compared to the original biased reservoir sampling procedure. However, it will also introduce the requested additional restriction to the reservoir size.

This method will be referred as *variable reservoir sampling* and the intuitive one with the fixed input probability p_{in} as *fixed reservoir sampling* in future reading.

3 Exponentially Decaying Sampling (EDS)

As a new method for biased reservoir sampling the so called *Exponentially Decaying Sampling (EDS)* is proposed [1]. This approach combines the previously discussed Reservoir Sampling method with the concept of Sliding Windows. Within EDS, the probability whether or not an item is included in the final sample will decay similar over time like in the Aggarwal Biased Reservoir Sampling method, but achieved following a different concept. The new method relies on a random influenced priority weight, which will be assigned to each new element in the reservoir. This weight value will determine for how many steps an element will remain, as its value will decay continuously over time. Similar to the Sliding Window sampling procedure, EDS is also built upon a decreasing probability that an item will remain in the Reservoir over time. Therefore there will be a minimum weight, which is necessary to remain in the Reservoir provided as input of the algorithm, symbolizing the end of the window in the Sliding Window approach.

All in all EDS will create a biased sample, comparable to bias reservoir sampling, but with some clear advantages. The EDS algorithm will support sampling of multiple elements at the same time, with the help of empty time steps. Additional, out-of-order arrivals of data points can also be sampled without large effort as the weight function behaves deterministically and the sampling procedure can be performed distributed as well.

3.1 The Algorithm

The basic concept of the *Exponentially Decaying Sampling (EDS)* can be drawn from a simple algorithm. Within EDS every data, which is received in the data stream, will be sampled and therefore contained in the reservoir at first. In order to achieve the bias of attributing recent sampled elements, a higher inclusion probability against older ones an additional weight value needs to be stored, which will influence the dropout time of the specific ele-

[1] The original algorithm for both the basic EDS algorithm and its optimized version were devised by Michael Cochez, the advisor of the thesis

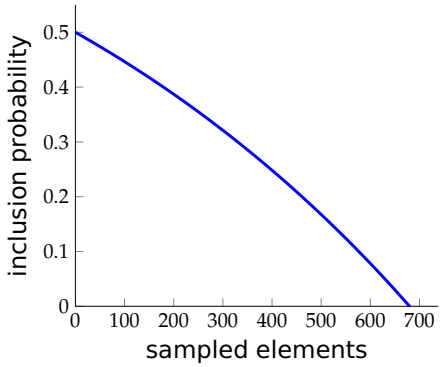

Figure 2: Decaying element inclusion probability in the context of stream progression

ment. This weight consists of a rational number and is defined randomly during the inclusion ($R \in (0, 1]$) of the element to the reservoir. The item weights will decay with some predefined factor α, every time a new element will be sampled in order to measure stream progression. Additionally to the decay a filtering will be performed in order to remove all elements from the reservoir, with a corresponding lower item weight as a defined lower weight bound X. Therefore, the algorithm depends on two parameters. At first the decay ratio α, which is a rational factor with a value less than one and used in order to decay the priority weights for every item in the reservoir at every iteration or timing step. Secondly the parameter of the lower bound X has to be defined, which will provide the bound for the priority weights above which they will remain in the reservoir. In the step of sampling a new arriving element, the inclusion probability to the reservoir will be therefore only depend on the lower bound at first, as the randomly assigned priority value needs to be larger than X. After this the inclusion probability of an existing item in the reservoir will decay with the provided factor of α and therefore the amount of steps an element will remain at the reservoir will be influenced by α further on. Figure 2 depicts this decay for an item in the reservoir and a specific scenario of $X = 0.5$ and $\alpha = 0.999$.

The proposed algorithm will maintain a non fixed-size reservoir with higher inclusion probability for recent elements than older ones. Nevertheless, through the influencing random variable also older elements have a probability to be present within the sample in contrast to a Sliding Window approach. Despite this simple approach, one can not recommend this algorithm for performing biased reservoir sampling in practice, because of its high computational costs:

Runtime complexity

In order to sample a new data point different steps have to be performed:

1. Add the new element to the reservoir
2. Loop through all elements in reservoir:
    a) Decrease item weight with factor α
    b) Check corresponding item weight against lower bound X and remove if the weight has become lower

Appending an element to the internal reservoir comes at nearly no costs, depending on the internal implementation. Typically, this will result in a constant cost of $\mathcal{O}(1)$. However the main cost for sampling a new data point will occur during the procedure of filtering out elements with a lower weight than the value of the lower bound X and the procedure of individual weight decrease. As the resulting loop will iterate the whole reservoir, this will result in a cost of $\mathcal{O}(n)$ with n equal to the current size of the reservoir. Therefore, the overall costs consists of the summation of both individual costs: $\mathcal{O}(1) + \mathcal{O}(n)$. However, as the linear term is dominating one can state the computational complexity of sampling an individual data point will be in $\mathcal{O}(n)$. For maintaining a large reservoir on a data stream, which moves forward with a high frequency this can get infeasible to perform. As a result data points could be missed because of the linear computational costs. Therefore, a more optimized approach is required, which is presented in the next section.

3.2 An optimized Algorithm

As a consequence of the result of the complexity analysis of the previous algorithm, a more efficient approach is requested. In the previous algorithm the main costs occurred during filtering the reservoir, for elements to be removed and to decrease the individual priority weight for each item. Therefore, a more sophisticated and optimized approach, that solves these two problems, would result in a better overall computational complexity of the algorithm.

First of all one can bypass the need for decreasing the individual weight for every item within the reservoir after each sampling step with a minor adjustment to the algorithm. Instead of decreasing the weight for each individual item after insertion and maintaining a fixed lower bound X one can provide each new sampled element with a continuously increasing weight value and leaving the weight of already sampled elements in the reservoir untouched.
With this approach recently sampled elements will gain a higher priority weight value assigned as older elements and therefore gain a higher probability for remaining in the reser-

Algorithm 3: EDS sampling algorithm

Data: X: threshold priority, P: increasing priority, α: decaying factor
Input: data point e

```
priority = P * random.Float64()
reservoir.Insert(e, priority)

headPriority = reservoir.Head()
while headPriority < X do
    reservoir.Remove()
    headPriority = reservoir.Head()
end

X = X / α
P = P / α
```

voir. Hence, no weights of already existing elements of the reservoir have to updated anymore. As shown before, new elements in the reservoir receive a random item weight value - this behavior needs to be adapted to the optimized approach as well. Therefore, the increasing item priority weights has also to be influenced by a random number in order to guarantee the same sampling properties as in the previous approach.
Nevertheless, this approach will eliminate the need of the iterative weight updates but also induce new problems. As the item weights will now continuously increase a fixed lower weight bound can not be preserved anymore. Instead, the lower bound has to increase with the same amount as the item weights in order to achieve the behavior of restricting the size of the reservoir by providing a dropout strategy. Additionally, in implementation, one has to consider the case of limit number ranges and has to avoid overflows as the item weights can not scale to infinity, which is assumed for the theoretical analysis. A possible solution for this problem will also be discussed in the following sections. But as first optimization, this approach will solve the problem of the iterative weight updates of the previous approach resulting in less complexity for item insertions into the reservoir.

The required filtering of elements in the reservoir, which holds a lower priority weight value as the current lower bound X, results in the second cost intensive operation of the sampling algorithm. Regarding the previous improvements, the static lower bound X modifies to an increasing lower weight value for each iteration X_i, but the problem of filtering out outdated elements still remains. However, as of the common nature of this problem several methods exists to address this. As in the algorithm only elements with the lowest weight in the reservoir are of interest, an ordering can be maintained internally. This will lead to fast lookup towards those elements of interest. A well known structure, that provides such an ordering and assists with dropping out elements with lowest or highest weight is the so called *Priority Queue*. Such queues provide low computational costs for accessing the element, which is

	Binary Heap	**Binomial Heap**	**Fibonacci Heap**	**Brodal Heap**
insert	$\mathcal{O}(log(n))$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
get min	$\mathcal{O}(1)$	$\mathcal{O}(log(n))$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
delete min	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$

Table 1: Runtime complexity of different heap operations [**cormen2009introduction**]

on the head of the structure. This can either be the minimum or the maximum element in the data structure. However, insertions and deletions will introduce additional costs as the requested ordering has to be maintained and possible elements have to be rearranged internally. Nevertheless, because of the high filtering costs in the previous approach, using such a data structure will eliminate the linear filtering and therefore improve the overall computational complexity. A more precise reflection of the computational costs will be provided in the next section. The improved version of the algorithm in pseudo code can be found in Algorithm 3.
For each new arriving data point e its random influenced item priority weight is determined beforehand. After that the element is inserted to the reservoir with its assigned priority. This procedure is followed by the discussed filtering approach using Priority Queues and is performed as long as no element with a lower bound than the current bound of X can be derived from the reservoir. As final step the current lower bound bound threshold X and the increasing priority value P have to be recalculated, using the decay factor of α in order to perform a timing step of the algorithm.

3.2.1 Runtime complexity

Due to the frequency and density of data, which are typically present in real world data stream scenarios, runtime costs play a key role of stream sampling algorithms as they have to be able to process information nearly online. Therefore, the runtime complexity of the new proposed methods needs to be evaluated more closely. In order the get a better estimate as in a typical worst-case scenario an analysis of the amortized complexity will be undertaken.

As the algorithm mainly consists of different operations on top of the Priority Queue, the overall complexity will be heavily influenced by the used implementation of such a queue. Typically, such queues internally rely on a heap data structure, hence the runtime complexity is given by the used heap. Table 1 shows different forms of heaps and operations necessary for an implementation as a Priority Queue. In order to get a good fit for the sampling algorithm, a heap which induces a low complexity on insertions and minimum extraction would be suitable. Therefore, a *Fibonacci Heap* or *Brodal Heap* can be applied. For the following complexity estimation their corresponding costs will be used.
If one applies the costs of the different heap operations to Algorithm 3 sampling an individual

data point will lead to the following costs:

$$\underset{\text{(insert)}}{\mathcal{O}(1)} \; + \underset{\text{(get min)}}{\mathcal{O}(1)} \quad + k * [\underbrace{\mathcal{O}(log(n))}_{\text{(delete)}} + \underbrace{\mathcal{O}(1)}_{\text{(get min)}}]$$

where $k \in \mathbb{Z}_{\geq 0}$ is a variable indicating the number of loop operations necessary. This number is typically unknown, however it can be estimated using an amortized analysis over a larger input size N. During the process of sampling those data points, k will either be zero or a number greater than zero depending on the current priority lower bound (X) and the minimum priority values in the queue ($headPriority$), which are individually influenced by a random number. Since the operations within the loop are only performed if there are elements within the queue with lower associated priority than the current lower bound, and only once for such a finding over the whole input, one can state that for the sum of k_i at step i holds: $\sum_{i=1}^{N} k_i = N$.
Therefore, the costs of sampling a larger input of size N will result in:

$$2N * \mathcal{O}(1) + N * [\mathcal{O}(log(n)) + \mathcal{O}(1)]$$

If one scales this result back to the sampling cost of an individual data point, the amortized costs will follow:

$$\frac{2N * \mathcal{O}(1) + N * [\mathcal{O}(log(n)) + \mathcal{O}(1)]}{N}$$
$$\Leftrightarrow 3 * \mathcal{O}(1) + \mathcal{O}(log(n)) \approx \mathcal{O}(log(n))$$

As a result the EDS sampling algorithm has an amortized runtime complexity of $\mathcal{O}(log(n))$, depending on the size n of its internal reservoir.

3.2.2 Numerical challenges

As a consequence of the performed optimization to the algorithm, the priority values for each new sampled element will not be bounded anymore. Also, the lower bound will increase in every performed sampling operation. Because of the possibility of an infinite sized data stream and the limit on representable numbers of a computer system, this will lead to problems in the implementation of the optimized sampling procedure. Therefore, a method is required, which overcomes this problem.
One intuitive solution for this problem, could be to perform a rescaling of the priorities and bound at some point in time. As the reservoir is bounded in size, which is typically far lower than the possible amount of numbers available, the rescale operation can be performed at

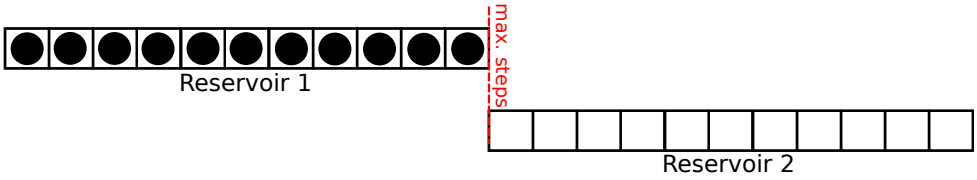

Figure 3: Handover scenario for sampling long running data streams

every time the maximum bound is exceeded. However, this approach will induce additional complexity as a complete pass through the reservoir and rescaling of every item priority will be required. A better approach will be to delay such a rescaling until the priority values will be able to exceed the maximum representable number available. The amount of possible sampling steps before such an event will occur can be even determined beforehand and is given with $|log_\alpha(\frac{f_{max}}{P_0})|$ for a decay factor of α, the initial priority weight of P_0 and the maximum value of a floating point number on a computer system f_{max}. The lower bound value X_0 has no relevance for this number because of the fact that the priority values will exceed the maximum possible representation before the lower bound. With the help of this estimate the rescaling can be reduced and will only be performed as soon as they are necessary.
Nevertheless, the rescaling will solve the problem of a possible overflow in the priority values but will still require a complete pass through the reservoir. As this was the main objective for the optimized algorithm an approach, which overcomes this problem without any additional pass through the reservoir might be more suitable.

Therefore, the method of a reservoir "handover" can be applied. Because of the fact that the point in time in which the overflow will occur is known beforehand. An additional "handover" reservoir can be prepared. This additional reservoir is implemented as an additional Priority Queue internally and will only be used as soon as an element will be able to produce an overflow. At this point in time all occurring insertions will be handled by the second Priority Queue and the current hold priority weight value of P_i can be reset to the initial value of P_0 preventing the possible overflow situation. This handover scenario is also depicted in Figure 3. However, for the dropout strategy two lower bound have to be concerned during the handover phase. Both, the lower bound value $X_i^{(1)}$ of the original reservoir, and the lower bound value of the second reservoir $X_i^{(2)}$ will need to be considered in order to perform a search for possible dropouts in the reservoirs. This behavior has to be maintained as long as values in the first reservoir are present. After all elements in the first reservoir have been evicted, the reservoir can be removed or held been forward for a following handover situation.
Beside the bookkeeping information and the additional lower bound for the second reservoir, no additional values need to be stored as the sampled data points are only split between the two reservoirs. Therefore, the additional storage requirement for the handover procedure is negligible. However, the computational costs will increase for such a scenario, as the search for dropouts has to be performed in more than one reservoir. Comparing the two presented

procedures, the handover procedure therefore will produce higher computational costs in total as the rescaling procedure. The rescaling will induce additional costs of $\mathcal{O}(n)$, with n equal the size of the first reservoir. However, the handover procedure has to consider two queues during the process of sampling. As the cost of performing the sampling operation on the second queue will show the same costs as a normal sampling procedure, they will not induce additional costs for the handover scenario. Nevertheless, the heap operations that have to be performed on the first queue, during such a scenario, will induce additional costs and therefore the costs of such a handover will follow from those operations. During the handover, lookups for every iteration and possible delete operations have to be performed on the first queue in order to maintain the requested sample. There will be a total of n delete operations necessary in order to complete the handover phase, resulting in the normal sampling scenario. As no modification to the remaining time of an element in the reservoir is introduced with this procedure, older elements will still be removed with a higher probability from the reservoir as more recent ones. Hence, the first queue will see more delete operations as the second one, which will contain the recently sampled elements from the stream. Furthermore, as no modifications to the random influence are introduced in this procedure, the total amount of delete operations that will occur in both maintained queues in will be equal to the normal sampling procedure with one queue. With this characteristic, the handover procedure will only induce additional costs for determining on which queue the deletion will be performed. This additional minimum extraction with cost of $\mathcal{O}(1)$ on the first queue has to be performed for every sampling operation until the handover phase completes. This information of the amount of necessary lookups involves the randomly influenced remaining time for any element in the reservoir and therefore no clear factor can be provided, but the following section will provide some estimates. As the size of the reservoir n is also influenced by this characteristic, the handover method will provide comparable costs as the rescaling procedure in the end. Nevertheless, the procedure of a handover will provide some fundamental benefit as during the procedure of rescaling the sampling has to be paused, in contrast to the handover scenario. Therefore, the handover method will give an essential benefit in an online data stream sampling scenario.

3.3 Characteristics of the Reservoir

Beside the runtime complexity of the sampling procedure, are the additional and potentially even more interesting characteristics of the obtained reservoir: Insights into that topic will also help to evaluate the runtime complexity more closely as they are depending on the reservoir size. Different from many other sampling techniques, EDS will not maintain a constant reservoir size during the process, as the reservoir size will fluctuate because of the random influence in the priority weights. Therefore, four different statistical features of the

reservoir need to be evaluated more closely, specifically:

1. The Max-Bound of the reservoir size
2. Expected size
3. Variance of the size
4. Skewness of the probability distribution

Insights to those different features will help to provide a good shape of the obtained reservoir with respect to the selected parameters beforehand. Additionally to the dependency of the run time, knowledge of the obtained reservoir size will also help to select the appropriate parameters for a specific scenario.

3.3.1 Max-Bound

As the current reservoir has to be maintained in memory, a maximum bound of the required storage volume will be an important information.
In order to obtain a maximum sized reservoir for a specific parameter scenario, it is assumed that each element will remain in the reservoir for the maximum amount of performed sampling steps. To achieve such a high remaining time for each element in the reservoir, the random influencing variable for the priority weight has to be always maximum. Because of its given interval of $(0, 1]$ the maximum value will be one. Therefore, the random variable will be always fixed to one, hence it can be neglected in the formula. Now this will follow:

Proof:

- The item priority at step n is given with: $P_n = P_0 * \alpha^{-n}$
- The lower priority threshold at step n is given with: $X_n = X_0 * \alpha^{-n}$
- An element will stay within the reservoir as long its priority P_i is more than X_n
  - Therefore it will be removed at step n if P_i is equal X_n
- Item P_i will stay in reservoir until $P_i = X_0 * \alpha^{-n}$, solve this for n:
  - $\frac{X_0}{P_i} = \alpha^n \Rightarrow n = log_\alpha(\frac{X_0}{P_i})$
  - resolve P_i: $n = log_\alpha(\frac{X_0}{P_0 * \alpha^{-i}}) \Rightarrow n = i + log_\alpha(\frac{X_0}{P_0})$
- For $i = 0$ the maximum size will follow as both series P_i and X_i are increasing monotonically with the same constant factor α and therefore the size of the reservoir will follow at the time step in which the first element will be removed. As $i = 0$ depicts the element, which was included using the initial item priority P_0, the first element in the reservoir can be derived using this number

This result shows that the maximum bound of the maintained reservoir is influenced by the specific values of α and the initial value of the priority values of X_0 and P_0. Specifically,

the maximum size of the reservoir is given by:

$$n_{max} = log_{\alpha}(\frac{X_0}{P_0})$$

3.3.2 A probability distribution as a model for EDS

For the maximum bound of the sample, the influence of the random variable within the algorithm could be neglected, as it was fixed to a constant factor of one. However, in order to gain more realistic insights of the maintained sample this simplification needs to be removed. Therefore, the maintained sample is now more randomly distributed and in order to provide a estimate for the expected outcome or its imposed variance the influence of the random variable and the overall probability distribution of element inclusion will need more discussion.

Typically, sampling is a procedure, in which the sample is extracted from a larger input set by making a yes/no decision on every element of the input. If this boolean valued decision returns a yes/true the element will be included in the sample and will be ignored otherwise. Therefore, each sampling step is independent from the others.
For this reason, the *binomial distribution* is a perfect model of such a sampling procedure in probability theory. The binomial distribution models the number of successes (yes decisions) of n independent events, each with the same success probability p. Multiple k coin flips asking for the probability of achieving k heads, are a famous example of the binomial distribution among many others. As this distribution is common in many scenarios, its properties are well researched and known but unfortunately do no apply directly to the EDS sampling procedure. As EDS has some bias by design in order to support recent elements within the sample, its inclusion probability p is unique for every arriving item and therefore the binomial distribution will not model the inclusion probability correctly. One can fix the inclusion probability to a certain level to gain a minimum or a maximum bound of the distribution but will not be able to get a precise result just with the binomial distribution alone.
However, there exist a special case of the binomial distribution called *Poisson binomial distribution* or also known as *Generalized Binomial Distribution*, in which each of the independent trials can have its own success probability p_i. Therefore, the Poisson binomial distribution can also be described as a sum of independent *Bernoulli trials* (Z_i) with different probabilities assigned, with $S_Z = Z_1 + Z_2 + \cdots + Z_N$ forming the Poisson binomial distribution. The probability mass function of this distribution, rendering the probability of k successes out of n is given with [**wang1993number**]:

$$\mathbf{Pr}[K = k] = \sum_{A \in F_k} \prod_{i \in A} p_i \prod_{j \in A^c} (1 - p_j)$$

where F_k is the set of all k-sized subsets in n, that can be selected and A^c forming the complement of such a set. As an example for $n = 3$ and $k = 2$, the set will be $F_2 = \{\{1,2\},\{1,3\},\{2,3\}\}$. It can be observed, that this set will grow quickly and will contain $n!/((n-k)!k!)$ elements for any value of n and k. For a fixed value of $n = 100$ and $k = 2$ the set will contain 4.950 elements and for $k = 3$ this value will rise to 161.700 elements, which have to be evaluated. For this reason evaluating the probability mass function directly is not feasible, especially in a data stream sampling scenario in which n will symbolize the number of elements in the stream and therefore can become extensively large over time. However, alternate approaches exist, in order to be able to provide this probability with less computational effort. There are either approximate solution that use a Poisson distribution in order to express this probability or other approaches like a discrete Fourier transformation or recursive formulations [**chen1997statistical**].
However, evaluating the probability mass function directly is not always necessary, as the expected outcome and its variance are known, since they inherit as a sum of the individual Bernoulli trials with its assigned success probability p_i. Since that the expected value and the variance of the Poisson binomial distribution are:

$$\mathbf{E}[n] = \sum_{i=1}^{n} p_i \qquad\qquad \mathbf{Var} = \sum_{i=1}^{n} (1 - p_i) p_i$$

In order to apply these formulas, expressing the success or inclusion probability for every data point i is necessary. Within EDS, elements will also drop out of the sample over time, therefore expressing the inclusion probability has to include a second variable symbolizing the current time of sampling. This probability can be expressed as follows:

Proof:

- As extension to the Max-Bound, an item is inserted with its item priority multiplied with some random variable $R \in (0,1]$
- Same as in the Max-Bound, an item is inserted to the reservoir, if its random influenced priority is higher than the current lower bound X_i
- Therefore, the success probability at time of arrival of item i is: $\mathbf{Pr}[P_i * R_i > X_i] = \mathbf{Pr}[R_i > \frac{X_i}{P_i}]$:
  - For any uniform random variable c of range $[a,b]$ follows: $\mathbf{Pr}[R_i > c] = \frac{b-c}{b-a}$
  - $\mathbf{Pr}[R_i > \frac{X_i}{P_i}] \approx 1 - \frac{X_i}{P_i}$ with uniform random variable $R_i \in (0,1]$ [a]
- If time moves forward, the inclusion probability of item i at time of arrival of item k with $i \leq k$ is:
  - $\mathbf{Pr}[R_i * P_i > X_k] = \mathbf{Pr}[R_i > \frac{X_k}{P_i}]$
  - $\mathbf{Pr}[R_i > \frac{X_0 * \alpha^{-k}}{P_0 * \alpha^{-i}}] = \mathbf{Pr}[R_i > \frac{X_0}{P_0} * \alpha^{i-k}] = 1 - \frac{X_0}{P_0} * \alpha^{i-k}$

[a] As the difference between the half-closed and the closed interval is marginal (value of zero in contrast to an infinite closed value to zero) this fact will be neglected in further reading

Therefore, the inclusion probability of an element i to the sample at time of arrival of element k with $i \leq k$ is:

$$\mathbf{Pr}_k[i] = 1 - \frac{X_0}{P_0} * \alpha^{i-k}$$

Expected-Size of the Reservoir

The proposed Max-Bound is a good measure for a worst-case storage requirement scenario, but a more realistic measure would be the expected size of the maintained reservoir. Using the obtained item inclusion probability and the expected size of the Poisson binomial distribution one can combine these results to the expected size of the sample.
Fixing the time of arrival k to n_{max} and iterating over all items from the beginning of the stream ($i = 0$) till the maximum size of the sample (n_{max}) will result in the expected size of the reservoir:

$$\mathbf{E}[n] = \sum_{i=0}^{n_{max}} 1 - \frac{X_0}{P_0} * \alpha^{i-n_{max}}$$

By fixing the variable k to n_{max} and iterating the probabilities till n_{max} one will get a bounded and an easy to compute formula in order to express the expected size of the reservoir. This restriction can be made, because every element after the induced maximum bound of ele-

ments in the reservoir n_{max}, can not become part of the reservoir. Therefore, its inclusion probability is less than zero and hence does not need to be considered for the expected size or any further statistical property.

Variance

Similar to the expected size, the variance as a measure of the deviation from the mean, can also be expressed by using the item inclusion probability and the Poisson binomial distribution. Therefore, the variance of the sample size is given with:

$$\mathbf{Var} = \sum_{i=0}^{n_{max}} (\frac{X_0}{P_0} * \alpha^{i-n_{max}}) * (1 - \frac{X_0}{P_0} * \alpha^{i-n_{max}})$$

Skewness

Skewness is a measure for how symmetric or asymmetric a probability distribution looks in shape. A probability distribution is symmetric, if its shape looks the same on each side around its mean value. The value of skewness can either be positive, negative or undefined (zero). For example the Normal distribution is a candidate for a completely symmetric distribution and therefore has a skewness value of zero. If the left tail of the distribution is longer the skewness will be negative and positive otherwise.
The skewness of the Poisson binomial distribution is known to be [**HONG201341**]:

$$\gamma = \frac{1}{\sigma^3} \sum_{i=1}^{n} (1 - 2p_i)(1 - p_i)p_i$$

Therefore, the skewness within the sample of EDS can be expressed with:

$$\gamma = \frac{1}{\sigma^3} \sum_{i=0}^{n_{max}} \underbrace{(-1 + \frac{2X_0}{P_0} * \alpha^{i-n_{max}})}_{(1)} \underbrace{(\frac{X_0}{P_0} * \alpha^{i-n_{max}})}_{(2)} \underbrace{(1 - \frac{X_0}{P_0} * \alpha^{i-n_{max}})}_{(3)}$$

By analysing this formula one can show that the skewness of the probability distribution of EDS will always be positive if $\frac{X_0}{P_0} \geq \frac{1}{2}$ [2]. This will be the case for a wide range of typical sampling scenario in which P_0 will be equal to one. For a positive skewness in the reservoir one can conclude, that its size will be lower than the expected outcome for a lot of outcomes and therefore can serve as a maximum bound for most cases.

[2] Proof follows on next page

Proof:

- With $i \in [0, n_{max}]$ and $0 < \frac{X_0}{P_0} < 1$, show bounds of sub-terms:
- In *(2)*:
  - for $i = 0$:
    $\frac{X_0}{P_0}\alpha^{0-n_{max}}$, with formula of n_{max}: $\frac{X_0}{P_0}\alpha^{-log_\alpha(\frac{X_0}{P_0})} \Leftrightarrow 1$
  - for $i = n_{max}$:
    $\frac{X_0}{P_0} * \alpha^{n_{max}-n_{max}} \Leftrightarrow \frac{X_0}{P_0}\alpha^0 \Leftrightarrow \frac{X_0}{P_0}$
  - therefore, *(2)* $\in [\frac{X_0}{P_0}, 1] \Rightarrow$ *(2)* > 0
- In *(3)*:
  - As *(2)* $\in [\frac{X_0}{P_0}, 1]$, therefore *(3)*: $1 -$ *(2)* $\in [0, 1 - \frac{X_0}{P_0}] \Rightarrow$ *(3)* ≥ 0
- In *(1)*:
  - As *(1)* $= -1 + 2 *$ *(2)* $\Leftrightarrow -\frac{1}{2} +$ *(2)*, with *(2)* $\in [\frac{X_0}{P_0}, 1]$
  - *(1)* ≥ 0 for $\frac{X_0}{P_0} \geq \frac{1}{2}$
- Assume $\frac{X_0}{P_0} \geq \frac{1}{2}$:
  - As product of positive terms (1), (2), (3), the result will also be positive
  - Furthermore, the sum of this product in the range $i \in [0, n_{max}]$ will always involve values > 0, as the only terms which can become zero is *(3)* for $i = 0$ and *(1)* for $i = n_{max}$.
  - Therefore, $\gamma > 0$ for $\frac{X_0}{P_0} \geq \frac{1}{2} \Rightarrow$ positive skewness

Numerical Stability

As all of the proposed measures rely on top of the inclusion probability, computing these for several scenarios and iterations will be unavoidable in order to express something about the properties of the maintained reservoir. However, because of the properties of the formula and the limit amount of precision, that floating point arithmetic can deliver the result can get unreliable and will not preserve the original model of the sampling algorithm. In order to see how affected the computation with limited precision floating point arithmetic is, the numerical properties of the inclusion probability needs to be further analyzed.

One aspect of the numerical properties of a problem is the so called *Condition Number*. The Condition Number of a problem gives some insights of how affected a given problem is to an error in the input. For some function $f(x + \delta x)$, in which x is the requested value to be evaluated, δx would be the induced error, which is not completely avoidable when relying on computation with a fixed precision. For example by using a representation of the number as a floating point value with limit precision. Therefore, the condition number expresses by how much the output value will change for a small change in the input. A large condition

number marks the problem as *ill-conditioned* representing a a high change of the output for a small change in the input. The opposite case is called a *good-conditioned* problem, which characterize with a small change of the output. The condition number $\kappa(x)$ of a problem $f(x)$ can be given with:

$$\kappa(x) = \frac{|f'(x)|}{|f(x)|/|x|}$$

By using a similar function to the inclusion probability $f(k) = 1 - \alpha^{-k}$ with its derivative $f'(k) = k * \alpha^{-k-1}$ the specific condition number of the problem can be derived like in the following:

$$\kappa(k) = \left| \frac{k * \alpha^{-k-1}}{1 - \alpha^{-k}/k} \right| \Leftrightarrow \left| \frac{k^2}{\alpha(\alpha^k - 1)} \right|$$

for any fixed $\alpha < 1$ and $\lim_{k \to \infty} \kappa(k)$ the function converges to infinity, which characterize it as *ill-conditioned* for large values of k.

As one can substitute $k = n_{max} - i$, this directly applies to the formula of the inclusion probability. Therefore, the expected size computation is also affected by the ill-conditioned formula. However, by using a data type with fair enough precision this problem can be neglected at least for small values of k. For example, with a value of $k = 1000$ and $\alpha = 0.99$ the condition number κ would be equal to 10^6. By using double precision computation the machine precision would be in a range of 10^{16} and therefore the result would still be reliable. Nevertheless, for a larger value of $k = 10^6$ and a value of α, which is closer to one $\alpha = 0.9999$ this property will quickly fade away. In this example the value of κ would be in the area of 10^{12}, which can not be seen as reliable anymore as the problem is now more affected of error than the precision can overcome.

Despite this fact computing the expected size or the different statistical properties like variance or skewness will still not be impossible with this formula, as they are not computed for a single large value of k alone. In fact, because of the summation within the formulas, multiple inclusion probabilities are summed up together in order to gain the proposed measures. Therefore, every computation also consists of a sum of multiple inclusion probability, that share a small value of k. Even more, the elements in the sum, which should have a higher inclusion probability, would have a smaller value of k than elements with a lower inclusion probability. This property results from the fact, that the iterative component i will get closer to the bound of n_{max} and therefore the value of k will get lower in every iteration closer to n_{max}. Therefore, elements with a low value k will get a higher influence to the final outcome than elements with a high valued k, that are more numerical unstable. As a result, this form of computation will minimize the error, which is often unavoidable in floating point

computations and can therefore predict the overall inclusion probability with a minimum of the induced error. The following chapter 4 will provide a further analysis on how precise the different stochastic properties can be estimated for some synthetic examples.

3.3.3 Equivalence of the algorithms

The previously analyzed mathematical properties of the reservoir respected the optimized approach exclusively. Nevertheless, the results directly apply also for the original algorithm as they behave equally. In order to conclude this equality, it has to be shown that they will maintain the same sample. For this reason the inclusion of new elements for the reservoir and the dropout procedure has to deliver equal results, respecting the inclusion probability in the reservoir.
As both algorithms will perform a similar inclusion strategy, the first property will follow directly. In both presented approaches an element will be sampled accompanied with either a random number $R \in (0,1]$ or a random influenced weight $P_i * R$ value. Either way the element will be included into the reservoir at first and therefore the inclusion probability at the point of sampling is equal for both algorithms. As it has been shown, that both algorithms will sample the same elements on occurring, it remains to show that the same elements will be removed from the reservoir at same point of time. For the original approach the amount of steps k a sampled element will remain in the reservoir can be derived as in the following:

$$\begin{aligned} R * \alpha^k = X \Leftrightarrow \alpha^k = \frac{X}{R} \\ \Leftrightarrow k = log_\alpha(\frac{X}{R}) \end{aligned} \qquad (\textit{original})$$

for some random variable $R \in (0,1]$, decay factor α and threshold bound X. For the optimized approach the point of a dropout t for a previously sampled element i can be given with:

$$\begin{aligned} P_i * R = X_k \Leftrightarrow P_0 * \alpha^{-i} * R = X_0 * \alpha^{-t} \Leftrightarrow \alpha^{t-i} = \frac{X_0}{P_0 * R} \\ \Leftrightarrow t - i = log_\alpha(\frac{X_0}{P_0 * R}) \end{aligned} \qquad (\textit{optimized})$$

for an additional priority weight factor P_0 and the initial threshold value of X_0.
As the result of the subtraction $t - i$ is exactly the amount of steps any sampled element will remain in the reservoir, both equations can be compared directly. Hence, for any scenario in which $X = X_0$ and $P_0 = 1$, both algorithms will perform the same dropout strategy.
For this reason and the property that both algorithms will sample the same elements, the

Deviation k	Minimum population
5	96%
6	97.2%
7	98%
8	98.4%
9	98.8%
10	99%
15	99.6%
20	99.8%

Table 2: Achieved population within deviation from the expected value

proposed approaches are equal for the presented scenario configuration.

3.4 Sampling with higher space constraints

As a special case, EDS can also adapt to a scenario with higher space constraints. Resulting from the internal probabilistic model, the maximum bound of the reservoir size has to be kept available during the whole sampling process in order to assure, that the whole sample will fit into the memory at all time. This space will needed to be assigned to the sampling process alone whether it will be actually used or not. With the known expected size and variance of the sample one can state that using the complete maximum bound in storage can get very unlikely.
As a measure of how unlikely this scenario will get, the *Chebyshev's inequality* can be used. It will give a probability measurement of how much of the population is covered within a given deviation from the mean. Hereby, the deviation k is expressed as a factor of the standard deviation (σ) [**kvanli2005concise**]:

$$\mathbf{Pr}[|X - \mu| \geq k\sigma] \leq \frac{1}{k}$$

The Table 2 provides an overview about how much of the actual population are contained within k times the standard deviation from the expected value.

These values represent a minimum lower bound and as they do not concern any special probability distribution are universal valid for all distributions, that have a defined variance. However, by using a bound, which is closer to the Poisson Binomial distribution it is possible to get significantly better result as with the Chebyshev's inequality. For example the *Chernoff bound* will provide a much sharper bound and as consequence will result in a lower required deviation parameter k. A Chernoff bound can be applied if the distribution relies upon sums of independent random variables, a condition which is not necessary for the Chebyshev inequality. As the used Poisson Binomial distribution suits this requirements,

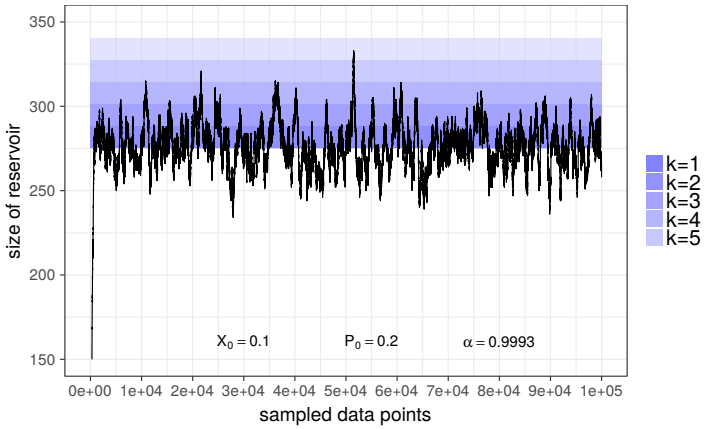

Figure 4: Size of Reservoir with k deviation from the mean

the Chernoff bound can be applied. Therefore, a sharper bound for the minimum population is given with [**Chung2006**]:

$$\textit{(Upper tail)} \quad \mathbf{Pr}[X \geq \mathbf{E}(X) + \lambda] \leq e^{-\frac{\lambda}{2(\mathbf{E}[X]+\lambda/3)}} \quad \text{with } \lambda = \sigma * k$$

As the bound depends on the standard deviation and the expected value of the distribution no general table can be provided. In order to provide an impression by how much closer this bound is a EDS sampling procedure with a symbolic setting ($P_0 = 0.2, X_0 = 0.1, \alpha = 0.9993$) will reach 99.99% of its population within a deviation of $k = 6$. Compared to the Chebyshev's bound with a population covered of 99.8% for $k = 20$ or of 97.2% for k equal to 6, this is a significant improvement.

By using those bounds one can restrict the required amount of space for sampling to the number: $\mathbf{E}[n] + k\sigma$ while achieving the specific population cover to the parameter k. In the example from above, such a restriction would decrease the storage requirement to 35.93% of the original maximum size. Figure 4 shows some example of a sampling procedure with different deviations of k with a possible maximum value of 990.
As a consequence, the storage requirement for EDS is much less but the maximum possible reservoir size will not fit into the reserved reservoir storage anymore. For this scenario the EDS algorithm has to implement a procedure in order to cope reached storage capacity during sampling. Hereby, multiple scenarios can be considered:

1. New data points can just be dropped until space is available again
2. Delete a randomly selected data point in order to make space
3. The data point with the lowest priority will be deleted before sampling a new one

All methods will induce a certain error to the sampling procedure as either the point which

Algorithm 4: EDS sampling algorithm with higher space constraints

Data: X: threshold priority, P: increasing probability, α: decaying factor
Input: data point i

```
if collection.Full() then
    collection.Pop()
end

priority = P * random.Float64()
collection.Insert(i, priority)
headPriority = collection.Head()
while headPriority < X do
    collection.Pop()
    headPriority = collection.Head()
end

X = X / α
P = P / α
```

should have been within the sample will be rejected or existing elements are deleted. As an error is inevitable the method, which induces the lowest possible error should been chosen. The first method, with simply rejecting new elements until space is available again, will be the simplest one to implement but also the method, which will induce the highest error. Certainly, any new element, which arrives for sampling, would have a high inclusion probability and therefore would stay within the reservoir for a longer period of time. Deleting such elements will result in a high induced error as this element should have originally been within the reservoir but is missing in the considered method. Any method, which will select an element that is more neglectable will reduce the induced error. For this reason the second method can possible provide a better result but the error of the third method would be minimal of the conducted scenarios. The element with the lowest priority, as it is selected by the third method, is the next element of the ones in the reservoir to be dropped out at a later point anyway. So the induced error only consists by the amount of time this will happen beforehand. Algorithm 4 implements the discussed behavior. Depending on the choice of parameter k, this will happen less or more frequently.

4 Computational Results

In order to evaluate the sampling procedure in practice, the optimized algorithm was implemented and proofed against a synthetic example, consisting of a stream with one byte sized elements [1]. During this process the actual gained measures are compared to the expected ones. Therefore, it was necessary to test multiple configurations of the EDS algorithm. An automatic testing procedure was designed, which iterates the algorithm with different parameters and observes several key aspects like reservoir size or timing information for insertions. Each scenario, consisting of the different observations, was recorded into a single data file and the resulting list of files served as input for the following analysis.
The implementation for this experiments is made on the *Golang* programming language and run on a standard desktop computer (Intel Core i5-6200U, 8GB RAM) multiple times under different work loads. This benchmark will provide good indices and empirical results of whether or not the theoretic measures and bounds also apply in practice and demonstrate the practical benefits for a wide range of sampling scenarios. As key points of the analysis are the computational complexity of maintaining the desired reservoir and the properties of the maintained reservoir considered.

4.1 Complexity

Empirical measuring of the complexity of a program or algorithm is not a trivial undertaking because many side effects will influence the measurement and result in a data set full of noise. However, such a measurement can provide helpful information about program behavior under different scenarios if one considers the possible high noise.

In order to measure the computational complexity and reduce the noise, the implementation has to be adjusted. For this reason memory allocation was performed beforehand and the internal garbage collection was disabled during the time of measurement. The measurement and analysis process followed a similar approach like [**goldsmith2007measuring**] and recorded the time for sampling a specific range of data points with different size of the reservoir involved. This process was performed for multiple configurations of EDS, resulting

[1] The source code for this benchmark is accessible at `https://git.rwth-aachen.de/georg.gross/eds-benchmark`

in a large data set of timing information per reservoir size. Furthermore, the whole process was performed multiple times in order to reduce the inference of side effects like different machine loads or interrupts during the measurement. As consequence of this, multiple time durations were available for each individual scenario. Therefore, the data set was aggregated to use always the lowest timing information, which was available. The minimum duration was seen as the value which is achievable with a minimum of inference given and reduces the noise significantly. Due to the fact of limited precision in time measurement on the CPU, time durations could not been recorded for sampling an individual data point but had to be extended to a whole set of sampling points. For the conducted measurement four different sets were considered. At first, the duration to sample a set consisting of 50.000 elements was performed. After that, sets consisting of 100.000, 200.000 and 400.000 elements followed.

Evaluation through Linear Regression

In order to evaluate those empirical measurements and map them to possible theoretical runtime complexities multiple *Linear Regressions* were performed. The method of Linear Regression tries to construct a linear model out of the input or training data set with minimum deviation. The sum of least-squares is usually used as a measurement for the deviation. Therefore, the Linear Regression tries to minimize the following error function [**han2011data**]:

$$\sum_{i=1}^{n}(y_i - (a + bx_i))^2$$

by selecting the best fit for the free variables a and b, in order to get a minimum amount of error.

In order to apply Linear Regression to the task of describing computational complexity out of empirical data the non-free variables x_i and y_i had to be mapped to the measurement:

- The y values represent the time duration/cost for sampling a specific set of elements
- The x values represent the current size of the maintained reservoir

By transforming x with the popular complexity classes a relation to the theoretically defined complexity can be drawn. Hereby, the selected values of the free variables a and b can be neglected for the complexity as they do not scale with the size of the input n and are just used in order to define the best linear model by Linear Regression. Figure 5 shows the development of the time durations for sampling given different reservoir sizes (points) together with the trained models as graphs. Illustrations for different sampling sets beside 100.000 can be found within the Appendix.

Beside the quadratic complexity class (or also any other n to the power of $k \in \mathbb{Z}_{>2}$) all remaining models provide an adequate fit to the measurements. In order to measure how

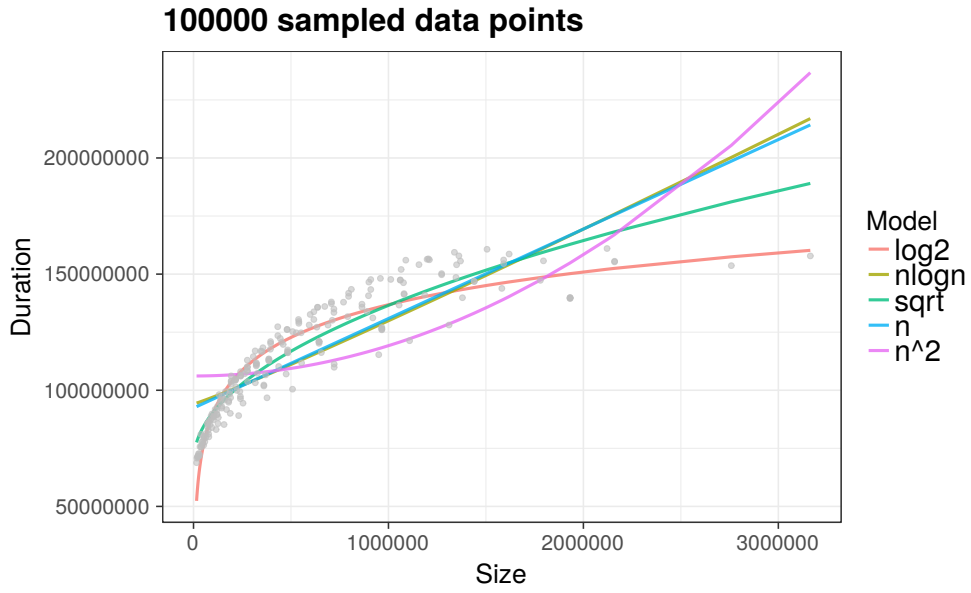

Figure 5: Time costs for sampling with different reservoir size together with complexity models

well each individual model fits the original input data and which one fits best the method of *Relative squared error* (R^2) can be applied. The method of R defines a measure of how accurate the model predicts the original training set. It is built upon a squared error loss function $(y_i' - \overline{y})$, in which y_i' represents the i-th prediction from the model and $\overline{y}$ the mean value of y from the training set. Beside the squared error there are also other loss function possible, as the squared error exaggerates the presence of outliers. For example using a loss function, which is built around the absolute deviation. Nevertheless, within R, the squared error is applied.

The given formula $(y_i' - \overline{y})$ symbolizes the squared amount of deviation of the model to the mean value of the training set and in order to be meaningful this value is normalized by the total sum of squares $(y_i - \overline{y})^2$, which is proportional to the variance. If one sums up those fractions over the whole model R^2 will follow:

$$R^2 = \frac{\sum_{i=1}^{n}(y_i' - \overline{y})^2}{\sum_{i=1}^{n}(y_i - \overline{y})^2}$$

The value of R^2 is bounded in the range $0 \leq R^2 \leq 1$ depending on the amount of variance, which can be described by the model. A value of one describes a scenario, in which the model follows the training data perfectly and can describe every point of variation. With a value of zero the trained model will not be able to describe any variation. Therefore, R^2 can be seen as a percentage value by how much the model will follow any variations within the training set. As a consequence of the fact, that the R^2 value will rise given more independent variables in the model albeit any better linear model, typically the *adjusted* R^2 is

	50k	100k	200k	400k	∅
$log(n)$	0.9022572	0.8975363	0.8917485	0.8563828	0.887
$nlog(n)$	0.7096134	0.703257	0.6966744	0.6888633	0.6888633
$\sqrt{n}$	0.8859925	0.8805475	0.8750745	0.8567127	0.8567127
n	0.7344956	0.7283206	0.7220057	0.7134474	0.7245673
n^2	0.4063633	0.3959854	0.3847625	0.3799948	0.3917765

Table 3: (adj.) R^2 values of different Linear Regression models

used. Therein, R^2 is additionally scaled with some constant depending upon the number of independent variables and observations to erase this behavior.

Table 3 shows an overview of the (adjusted) R^2 values from the conducted measurements. Each model was independently analyzed through the different sample sets, resulting in an independent R^2 value. It can be observed that the $log(n)$ followed the measurement at best directly followed by the $\sqrt{n}$ model. The quadratic model provided the worst fit as it was expected from the graph. Since the theoretical complexity analysis revealed an amortized complexity of $\mathcal{O}(log(n))$ the measurements can now support this outcome. However, $\sqrt{n}$ also yield an acceptable result and was only 3% less accurate as the $log(n)$ model on average. But as the $\sqrt{n}$ function and the $log(n)$ function share a similar shape in the beginning, this result was not unexpected. For larger values of n the $\sqrt{n}$ function will show a much steeper increase then $log(n)$, which was also noticeable on the data set of the measurement. The log function predicted the smaller amount of large values of n far better as the square root-based function. For additional larger data points this behavior will continue and $\sqrt{n}$ would gain a worse fit as $log(n)$. Therefore, the $log(n)$ model can be considered as the model, which predicted the measurements at best, resulting in a validation of the theoretically proposed complexity of $\mathcal{O}(log(n))$ with $n =$ Size of Reservoir.

4.2 Reservoir characteristics

Additionally to the processing time, which is required to maintain the reservoir, the actual storage requirements are also a critical point of the analysis. This section will try to support the previous discussed measurements and bounds. Therefore, the *Max-Bound*, *Expected size* and the *Variance* are evaluated in a synthetic example. Multiple input configuration, resulting in different reservoir scenarios, are applied similar to the complexity benchmark before. In total *7000 different scenarios* were tested, which cover a wide range from small reservoir sizes to larger ones.

Maximum-Bound

In all tested scenarios the proposed Max-Bound of section 3.3.1 was never exceeded. In fact, it was never reached either. This observation shows the unlikeness of getting a reservoir with maximum possible size. The theoretical maximum bound relies upon the fact that each new inserted sample point will be always added with the full item weight. Over a reasonable amount of sampled data points, this behavior will be getting highly unlikely. In order to express the validity of the theoretical bound, even for this unrealistic side case, an additional measurement was performed. Within this experiment the random influencing variable was fixed to one, resulting in an item insertion which includes always the highest possible weight. As the theoretical bound explicitly relies upon this assumption, the maximum bound was reached, but not exceeded in every scenario. Even more, the measured maximum was always equal to the rounded down (to the nearest integer) value of the computed theoretical bound in all of the 2.000 different scenarios measured. Therefore, the empirical results shows the expected characteristics and supports the theoretical bound.

In fact, even more interesting than the exceed of the maximum, is how close the actual reservoir size was to the maximum bound during the process. Figure 6 shows by what fraction the observed size was to the computed maximum bound during sampling, for different values of α and a fixed setting of X_0 and P_0 [2]. It can be observed, that for every increase of α the difference between the maximum bound and the measured one will increase and therefore the percentage will decline. Such a behavior is not unexpected as for a larger reservoir the probability of reaching the maximum bound will decay, because of the larger amounts of full item weight insertions that are necessary to reach the bound. During all measurements the minimum percentage value of the observed maximum size against the theoretical one was around 8.8%. The maximum percentage was around 59.3% with a overall median of 29.5%. These values show that in practice, reaching the maximum size will be highly unrealistic and therefore applying the technique of section 3.4 for storage reduction can provide a drastically lower space requirement. As a result follows, that for all measurements the proposed theoretical bound holds but the probability of reaching the bound decreases with the sample size and therefore provide a tighter bound for small reservoirs compared to larger ones.

The Expected Size and Variance

As a practically more relevant measure, the expected size and the variance of the sample can be conducted. Different from the Maximum-Bound these measures can be estimates using the proposed formulas with a sum over the different inclusion probabilities. This directly results in a more iterative formulation with different numerical properties. Regarding this

[2] $X_0 = 0.2$, $P_0 = 0.7$

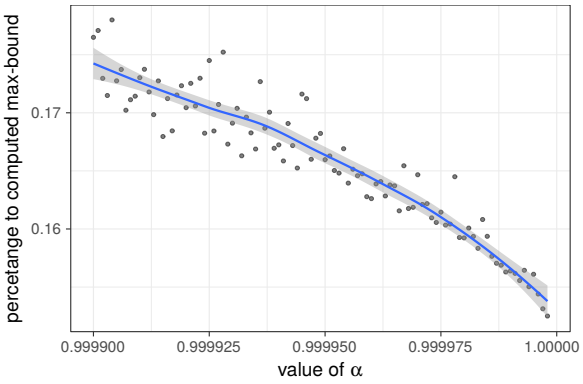

Figure 6: Percentage reached of maximum bound with different values of α

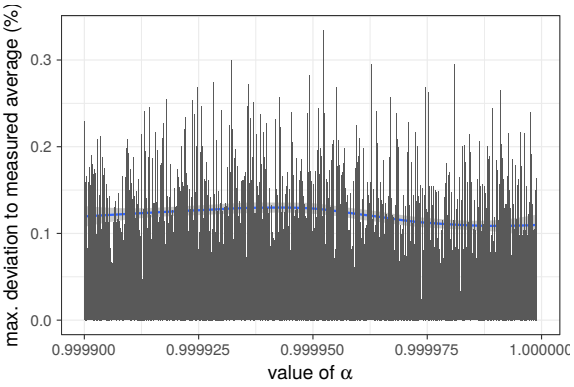

Figure 7: Maximum deviation to measured average size in units of std. dev.

fact, section 3.3.2 showed that the inclusion probability formula is classified as numerical ill-conditioned for data point that are sampled a longer time ago. For data points, that occurred more recently this property will extinguish and the resulting value is more reliable. In order to see how reliable the proposed formula of the expected-size and the variance can predict the actual maintained ones within the reservoir and how affected the computation is in practice, measurements in the synthetic example are performed. Therefore, analyzing the deviation of the proposed bounds of the reservoir against the measured ones is a key element of this section. Figure 7 depicts the overall maximum deviation of the expected size within the conducted experiments graphically, given an increasing value of α. The data was aggregated to show the maximum deviation for each value of α, in order to provide a worst-case scenario. Within the plot the deviation is given in units of standard deviations of the used scenario. This will provide a better estimation of how reliable the actual bound is estimated.

Within this plot it can be recognized, that the deviation never reached a full unit of standard deviation, even a half unit was never exceeded. The maximum deviation in the expected size, which was measured, was at a value of 0.334 in units of standard deviation. The minimum had a very low rate of 0.011 units of standard deviation. This results show that the theoretical expected size bound can explain the maintained bound in the experiment with a good precision. For further analysis a regression line was computed, which is also visible in Figure 7. The regression showed a nearly constant shape supporting an even distribution of the different deviations measured. Therefore no specific scenario range with a high deviation of the proposed bound could be identified. This result shows that the fraction of possible numerical ill-conditioned computations have no big influence for the whole formula and the expected size can be estimated with reasonable accuracy in practice.

For the variance, a similar behavior was observed, which was expected as the theoretical bound also relies upon the same item inclusion probability formula. Therein, the fractional deviation ranged from a minimum value of 0.017 to a value of 0.36 with a mean of 0.15. Therefore, the estimated variance is in a similar accuracy scale as the expected size and can

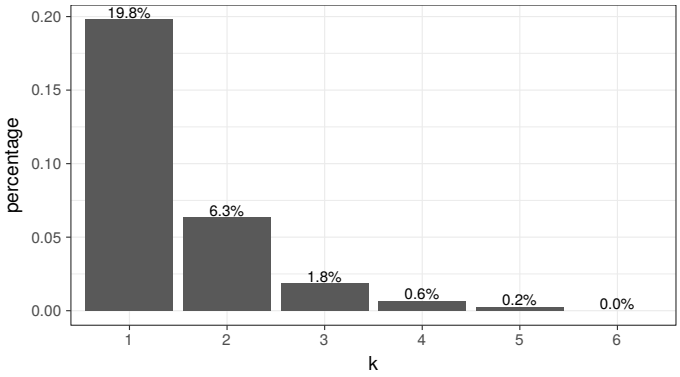

Figure 8: Different Confidence levels of k, with maximum time they were required during the experiments

provide good estimation about the maintained reservoir over the time of sampling.

Confidence Intervals

The performed maximum-bound experiments demonstrated that reaching the maximum size of the reservoir is highly unlikely, during the process of sampling. This property can be used in order to reduce the storage requirement for sampling drastically, as it was also discussed in Section 3.4. Hereby, the storage requirement was limited with help of a factor k, the standard deviation and the mean of the reservoir size. The new defined maximum reservoir size resulted from the mean value plus the factor k times the standard deviation. By using the discussed Chernoff bound a probability for the population which is covered can be derived from the given setting of parameters. For higher values of k this probability will rise but the storage requirement that is necessary to store those values would also increase.
In Figure 8 the results from the computational experiments with different confidence intervals k are demonstrated. Each bar represents an increasing value of k with an associated percentage value. The percentage value symbolizes the amount of steps a sampling scenario remains in the given level. In order to get an impression for the whole data set, those values are aggregated to show only the maximum percentages measured. Therefore, one can see, that for some scenarios the confidence level of $k = 1$ was required in 19.8% of steps during sampling and no scenario reached the $k = 6$ level at any time. The percentage values are decreasing for larger values of k like it was expected and demonstrates that reaching a size of the reservoir near the maximum bound is unrealistic. As for the measured rather low values of k a large population of reservoir sizes have been covered, it confirms the expectations from the maximum-bound even further. By relying on this findings, one can fix a lower maximum reservoir size and it is safe to say that this size will not be exceeded at any time. Hence, the proposed method of storage reduction will result in drastically lower storage requirement. As for example by fixing the value of k to 5 resulted in a storage requirement of

only 14.5% to the maximum size on average, within the conducted experiments. Therefore, a value of $k = 5$ provides a reasonable storage reduction with a high confidence value that this size will not be exceeded and can bee seen as an empirically good value for a large number of scenarios.

In order to assure a specific level of certainty the proposed bound is important and it was measured in all tested scenarios that the discussed Chernoff bounds hold its provided estimations. Even more, the experiments showed that there might be also a sharper bound, as the reservoir remained much longer within some confidence level as the Chernoff bound is predicting it. Thus, it was observed that during all measured scenarios, the reservoir remained with a higher probability in the interval of $k = 1$ as the Chernoff Bound value for $k = 2$. Only for $k = 3$ the Chernoff Bound slightly started to gain a higher probability as the measured value for $k = 1$.
Nevertheless, the bound is still sharp enough to provide some practical benefits in sampling scenario with higher space constraints.

Startup phase of the Reservoir

As an additional crucial part of real world data stream sampling scenarios is the amount of steps required in order to achieve a certain level of data points during the startup phase of the reservoir. As sampling on data streams always have to rely on new incoming elements and can not sample an existing pool of data directly, it is favorable that most of the new incoming elements should directly go into the reservoir for the initial reservoir constructing phase. Therefore, the biased reservoir sampling algorithm introduced an specialized variant, which supports initial fast reservoir construction. A fast startup phase of any sampling algorithm would decrease the amount of time that has to be waited before any analysis can be conducted with the whole reservoir in size. For any analysis, which is conducted before only a small subset is available and may therefore be not representative. For data streams that have a high frequency or sampling methods with a high inclusion probability, the time which is necessary in order to reach a full reservoir will be small but for streams with a slow pace of incoming new data points or sampling scenarios with a low inclusion probability this can get a critical component.

As EDS does not maintain any fixed sized reservoir, there is no fixed point to reach directly. Nevertheless, the expected size can be considered as such a bound, which is to be reached and will be available further one with a high probability.
The original EDS sampling algorithm is not optimized for such a requirement as its primary target is to hold the statistical assumption about the inclusion probability for an item at any

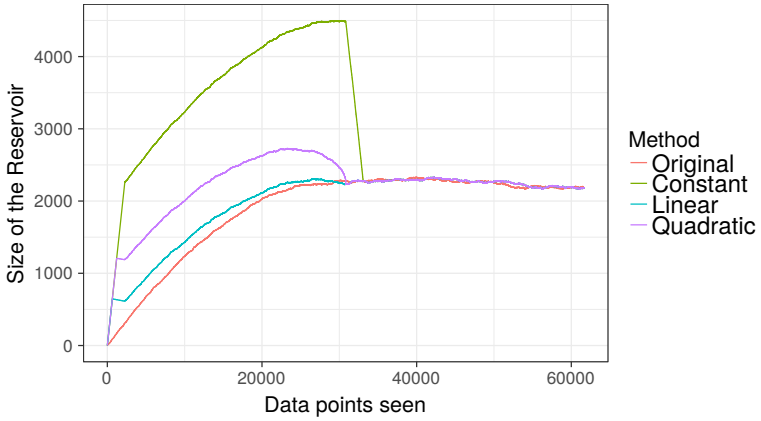

Figure 9: Different methods for fast reservoir startup

time. Introducing such a fast startup for the reservoir in the beginning would violate this assumption in the beginning. Therefore, EDS will sample any item of the data stream with the same probability in the beginning of the sampling process as for the whole sampling process in general. As a consequence a possibly large amount of steps are measured in the benchmark scenario in order to reach the expected sample size. On average of all measurements about 60% of the maximum size of elements in the sample had to been seen in order to reach the expected sample size (minimum: 23%, maximum: 88%). For a scenario, in which a large sample should be maintained and data occurs only once in a while reaching the expected size will take some time during that only a partial sample is available for processing.

Therefore, some additional bias towards first elements in the stream might be acceptable in order to guarantee a fast startup phase of the reservoir. An intuitive method for such an additional bias would be to pick every element into the reservoir until a certain threshold of points seen is reached. After that, the original sampling procedure is applied. As discussed before, a good fit for such a threshold value will be the expected size of the sample. This method would definitely decrease the startup phase to the minimum but will also generate a sample, which is untypically large in size at the beginning. Due to this added bias, the sample will increase closer to the maximum bound n_{max} at the beginning and will quickly decrease towards the expected size as soon as the first elements from the sample will fade out. This will introduce some extraordinary elevation in the sample, which can also be noticed in Figure 9. In order to minimize this introduced elevation a variable bias can be considered, which will start at a high value in the beginning and will decrease with stream progression. Using such an approach, the handover between the fast startup procedure and the original algorithm will be much smoother in shape. Hereby, either a linear variation and a quadratic are taken into account. Both variations will start at value of one and will decay towards the value of 0.5. Therefore, they can replace the random influencing in the original algorithm during the startup phase, while guaranteeing to have a higher inclusion probability as the

original algorithm, without any further change to the method. The different shapes of the obtained startup phase is also depicted in the Figure 9. Hereby, the different biases follow the following functions:

Constant	**Linear**	**Quadratic**
$f_t(i) = 1$	$f_t(i) = \frac{1}{2} + \frac{t-i}{2t}$	$f_t(i) = -\frac{1}{2t^2} * i^2 + 1$

for some defined threshold value t and the current number of element i in the data stream.
The constant method refers to the intuitive method, in which every item will be included in the reservoir until the certain threshold is reached.
During the benchmark the different fast startup scenarios achieved the following results:

	Time to Average	Elevation
Original	59.55%	-
Constant	10.55%	98.47%
Linear	42.57%	4.17%
Quadratic	21.70%	27.3%

Table 4: Results of different fast startup scenarios measured

The values of *Time To Average* state the amount of steps on average, which required to reach the expected value. The percentage was computed using the maximum size of the reservoir as base value. The *Elevation* measures the absolute difference between the maximum size of the reservoir during the startup phase and the gained expected size afterwards in percentage to the expected size.
It can be observed that the constant method achieved the fastest startup of the reservoir. This result was not unexpected because this method describes the fastest possible way to fill up the reservoir. Therefore, the result of the constant method can serve as lower optimal bound for comparison with the other methods. However, as consequence of the fast startup also a high additional elevation in reservoir size is introduced. The elevation of the constant method can also be seen as the maximum possible elevation and therefore can be used as the worst maximum bound that can be achieved for such a method in the data of the measurements. Hence, it can be concluded that the linear method has clearly its focus towards decreasing the elevation and achieving only a slightly better startup than without any special fast startup scenario.
The most balanced results have been achieved by the quadratic decay of the inclusion probability in the beginning. This method can therefore be seen as the best fit for a fast startup scenario, while also generating a low additional elevation in the reservoir size.

5 Advanced features

Different from the normal biased reservoir sampling, which was discussed in the previous chapter some more enhanced features of sampling can also be performed with the EDS algorithm. This chapter will provide some introduction to common problem in data stream sampling and presents some solutions.

5.1 Asynchronous element arrival

A common problem of data streams in general is delayed or asynchronous arrival of elements due to transmission errors or just delayed submission. If the data consists of elements, that enforce a specific order such an afterwards arrivals can render the received data set unusable if the order can not be reconstructed or this fact is simply ignored. Therefore, several techniques in data transmission exists in order to prevent these problems and make some real world scenarios like video conferencing even possible.

In the context of data stream sampling, this problem is described as *out-of-order* arrivals of elements and also needs to be considered specifically. It is not necessary to pay attention to this fact in every data streaming scenario, but as it is often the case that a data stream induces some time dependency in the information it is transmitting this information should be preserved. In general, one can differ between two timing measurements in the context of data streams. On the one hand, there is the *Processing time*, which induces the time a new element was perceived from a receiving instance measured by a wall clock. On the other hand, the data in the stream itself can embed a time value. This information is referred as *Event time*. Therein, the new element describes the exact time on which the described event occurred. Because of the discussed possible late arrival of element in the data stream at the receivers side, the event time can differ from the measured processing time.
For this reason, in such an event time-based scenario out-of-order arrivals are not negligible and need to be handled specifically. This is especially important for a biased sampling scenario. For a bias towards more recent elements in the sample out-of-order arriving elements have to be considered. If such a sampling scenario neglects this fact the maintained sample may not preserve the bias in a correct manner, because element arrivals can contain

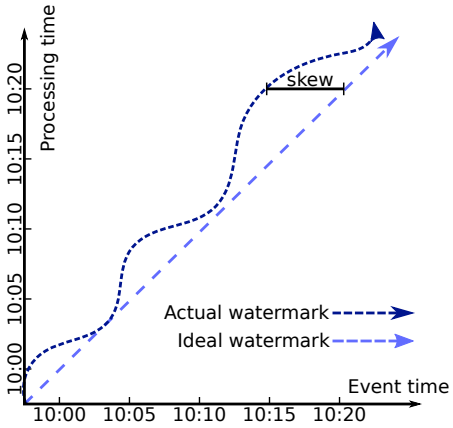

Figure 10: Possible skew between processing and event with involved Watermark

a highly different event time compared to the time of reception. Therefore, the attributed priority value should adapt to this behavior.

This characteristic of the data stream induces several new problems. Most important: The data stream can now be seen as unbounded in both directions. Of course new elements can arrive in the positive direction of time but the possibility of out-of-order elements arrival make even arrivals in negative time direction possible at any point of time. For this reason the stream will get unbounded in both directions. As a consequence, no clear progress on the event time of the stream can be given, which will lead to some processing issues in the implementation of any further analysis or sampling method. For example a Sliding Window approach on top of a asynchronous stream with a fixed time duration would miss elements that arrive late in the stream after the window of this time area has already processed. For the system, which maintains this window, there is no way to be assured that no further elements will occur in the stream which should be included in a window of a fixed time duration.

Therefore, a concept called *Watermarks* exists in order to provide any stream processing system with the information of current event state in the data stream. Hereby, Watermarks show similar to a water level the current level on event time progression. This information can be given by the source of the stream or can be estimated by the receiver with the help of a heuristic. In both ways, the watermark can either be obtained using perfect knowledge on the submitted data and one can generate a *perfect watermark* or a *heuristic watermark* is estimated. For many scenarios generating a perfect watermark will not be possible[1], however heuristics usually can provide a good estimate on the progression of event time [**akidau2015dataflow**]. Figure 10 depicts the skew between a obtained watermark and the ideal state with perfect alignment between event and processing time graphically. As Watermarks occurrence and length can change over stream progression, they also pro-

[1] Log files are typically a good example for maintaining a perfect watermark, as they follow a clear time forwarding structure.

vide some insights about the validity of the stream itself. A stream with a watermark, which is very close to the processing time, can gain a higher validity as a stream with the opposite. As this value can also change over time this characteristics can also be observed for the same stream under different situations. For example during the night time and daytime. Therefore, watermarks are an important factor for stream processing.

Out-of-order arrivals with EDS

The proposed algorithm of the previous section does not respect event time or out-of-order arrival and therefore only supports sampling with the processing time characteristic. However, with small modifications event time based sampling with out-of-order arrivals can be performed. This section will provide an intuitive simple version at first and more optimized adaptations for special circumstances afterwards.

In order to support event time based sampling the associated event time of each item has to be respected during sampling. For this reason, the procedure of making a time step needs to be changed. In the original algorithm a timing step was made for every item which occurred in the stream. However, this is in general not necessary and will be changed for the event time based sampling procedure. Hereby, a step forward in time is only performed for every element with a higher event time as the most recent one, which was perceived during the progress of sampling. Therefore, this process adopts directly to the stream progression. It has to be noted, that because of the possible asynchronous behavior of event time based data streams, more than one step forward in time could be necessary. In every increase of time the current maintained lower bound X and the priority variable P has to be increased and therefore recalculated. In contrast to the original algorithm with its processing time based design, for the event based algorithm it can not be guaranteed that after a currently maintained event time, the successor will directly follow on the stream. For this reason, the currently maintained event time has to be stored in order to have knowledge about the current state of stream progression. As an additional consequence follows that there might be jumps between the currently maintained bound X and its successor bound, which can be way larger than one. In order to handle this behavior, the approach of the simple event time based sampling algorithm relies on recomputing those values from scratch in every iteration. However, for a direct successor element this procedure can be avoided by relying on the currently maintained values. Depending on the behavior of the data stream more direct successor steps or complete re-computations have to be performed. For any out-of-order element the exponential function $P_t = P_0 * \alpha^{-t}$ has to be recomputed in order to get the item priority values at those time step t. With the help of this priority values the original method of EDS can be applied, without any further changes. This procedure will work and fully implement a sampling method with out-of-order arriving elements and describes the intuitive simple version.

Algorithm 5: Event time based sampling with tree lookup

Data: currently maintained maximum event time t_{sample}, user defined treshold
Input: data point e, event time of point $t_{element}$

// new element time is higher than currently maintained one in sample $\rightarrow$ time step
if $t_{sample} < t_{element}$ **then**
    timestep(e, $t_{element}$)
    return
end

// new element is out-of-order arrival, either direct compute or perform lookup
if $t_{sample} - t_{element} <$ *treshold* **then**
    $P_{element}$ = compute($t_{element}$)
    store($t_{element}$, $P_{element}$)
end
else
    $P_{element}$ = lookup($t_{element}$)
    store($t_{element}$, $P_{element}$)
end

[proceed with normal sampling step without time increase]

Tree-based lookup

However, computing the priority values for any out-of-order element from scratch might not be the most efficient procedure as it is very likely that during sampling, a similar priority had already been computed before. Therefore, a concept is proposed, which stores such similar values in a balanced binary tree structure for fast lookup and relies on the archived values as starting point of the computation. This will reduce the exponent in the priority formula and therefore lower the costs for computation, but however will also introduce additional costs for maintaining the tree structure and retrieving the closest element.
In order to compare both methods, the computational costs for both has to be known. As the methods only differ in determining the different element priority values P_t at a point of time t, only those costs need to be considered. In the simple approach these consists of evaluating the formula and especially by performing the exponentiation. There are several concepts that implement fast exponentiation but the most dominant one is called *exponentiation by squaring* [**cohen2005handbook**]. By using such a method the computational costs for evaluating the formula $P_t = P_O * \alpha^{-t}$ will be $\mathcal{O}(M(\cdot) * log_2(t) + M(\cdot))$, with $M(k)$ inducing the computational complexity for multiplying two k digits values.

Algorithm 5 implements the event time based sampling procedure with support of tree lookup

for some evaluations. For every new arriving element e it decides, whether it should perform a "normal" sampling step with time increase (first case) or if the element is an out-of-order arrival and therefore only determining the element priority value at the specific time point $t_{element}$ is required. In the "normal" case of element arrival, the new element has a higher event time $t_{element}$ as the currently maintained one t_{sample} in the sample. Hereby, the standard procedures follows, with storing the element in the reservoir and performing a time step towards the element event time. The element event time now, replaces the current value of t_{sample} and the sampling procedure will follow event time progression. If the event time of the new element is less then the current event time of the sample the second procedure will be undertaken. Hereby, either a direct computation of the element priority will follow or a tree lookup will be performed. The strategy will be decided by a user defined parameter. This method is chosen, because for small differences between the currently maintained event time of the sample and the of the new element, direct computations will be always faster, by choosing the current priority of the reservoir P as starting point. Therefore, this threshold would be typically small. In the other case a tree lookup will be performed. The lookup procedure will search for a node containing the requested $t_{element}$ value as key and will maintain a closest match to the requested key as backup if no perfect match can be retrieved. In this case, additional calculations have to be performed in order to reach the priority value for the requested time with the determined closest match as starting point. In this case, the element is currently not known to the tree and will be added afterwards. Also the direct computation will enrich the tree with its calculated value. For the side case, that the tree can not provide any closest element (empty tree) also a direct computation of the priority will be performed. In order to provide a good performance during lookup and insertion a balanced tree data structure will be necessary. An appropriate balanced tree data structure would be an *AVL Tree* or a *Red-Black Tree*. Both provide reasonable efficient lookup and insertions methods with costs of $\mathcal{O}(log(n))$, because of their balancing criteria. All operations performed on the tree data structure heavily depend of the height of the maintained tree structure. With the help of the balancing nearly optimal possible height during the process can be guaranteed. As an additional measure the height of the tree can be minimized with the help of the received watermarks during stream progression. For each received watermark possible multiple elements currently maintained in the tree data structure can be removed as they are not interesting anymore. The watermark removal procedure will traverse the tree and will mark all elements with lower referenced time for deletion. For any scenario, in which the root node and its right child is lower than the watermark, the whole subtree, which is induced by the root can be deleted completely and further traversal is not required anymore for this node. Furthermore, subtree deletions can be performed very efficiently on AVL or Red-Black trees with computational complexity of $\mathcal{O}(log(n))$ [**tarjanDataStructures**]. Therefore, the received watermarks assists to maintaining a compact tree structure for the current required priority information.

Despite all optimizations, benchmarks of the different approaches showed, that the simple approach performed faster in most of the cases. Only in rare cases the tree-based approach can provide a better performance, on average the simple method was more than twice as fast (see Table 5 for more detailed information). The reason will follow from the complexity analysis of the tree based method.

Complexity analysis

For any data point, which will be sampled with the tree based method, either a similar procedure like in the simple approach is performed or a tree lookup is conducted. The lookup will result in costs of $\mathcal{O}(log_2(n))$ for the search in the tree and additional costs of $\mathcal{O}(M(\cdot) + M(\cdot) * log_2(d)$ for the following calculation of the exact priority value. Hereby, d represents the distance between the closest match in the tree towards the required actual value. After the requested value has been found, the algorithm stores the result in the tree for following data points, which also induces additional insertion costs of $\mathcal{O}(log_2(n))$. Therefore, one iteration of lookup results in costs of: $\mathcal{O}(2log_2(n) + M(\cdot) * log_2(d) + M(\cdot))$.
For a direct comparison of these costs to the simple method with $\mathcal{O}(M(\cdot) * log_2(t) + M(\cdot))$ some simplifications can be undertaken. As both methods had to perform an exponentiation step, the required costs for the multiplication can be neglected. The same can be conducted for the logarithm. Therefore, the point in time, in which both methods induce the same costs is given with:

$$t = d + n, \text{ with } d \ll n$$

If one substitutes the number of elements in the tree n with the maximum size of the reservoir n_{max}, more insights can be derived. As storing more values, as in the bound of n_{max} inside the tree will not provide any larger benefit, this constraint will not influence the results. If one further states, that the out-of-order arriving elements follow a uniform distribution, additional characteristics of the distance value d can be provided. The value d measures the nearest match, which have been found during tree traversal and therefore are crucial for the exact calculation of the priority value afterwards. The closer this distance is to the actual requested value, the less calculations will be necessary. With the assumed uniform distribution in out-of-order arrivals, the event time values within the tree are also uniformly distributed. As the distance $Y = |X_1 - X_2|$ of two equally uniform distributed random variables X_1, X_2 in the bound of $[0, L]$ is again a random variable, the expected value of the distance can be given with [**philip2007probability**]:

$$\mathbf{E}[Y] = \frac{L}{3}$$

Because of the constraint that n is equal to n_{max} the distance of both random variables will be in the range of $[0, n_{max}]$ and therefore follows that the expected distance is equal to $\frac{n_{max}}{3}$. It has to be noted, that d symbolizes the minimum distance and therefore $\frac{n_{max}}{3}$ can only serve as an upper bound of d. With those assumptions the point in time, in which both methods induce the same costs is bounded by:

$$t = \frac{n_{max}}{3} + (n_{max})^2$$

For any value of the exponent t, which is larger than the results of this formula the tree-based method will provide a better performance. However, for an scenario with n_{max} equal to 1000 the exponent must have a value of 10^6 in order to achieve better results with the tree-based method.
As an optimization of the tree-based method the frequency of storing the computed results within the tree can be decreased to a certain level. For example by performing the insertion operation only for 10% of all sample operations the costs will decrease towards:

$$t = \frac{10 * n_{max}}{3} + (n_{max})^{1.1}$$

Hereby, the expected distance value will increase but the lower amount of insertions in the tree will decrease the more dominant exponent of the exponential term given in the cost formula. For such a scenario of n_{max} equal to 1000 at an exponent level of around 5330 the tree-based method will provide a better performance.

It has to be noted, that this result relies on the assumption of uniform distributed out-of-order arrivals and an upper bound for the minimum distance, which can be determined by search inside the tree. However, this analysis shows that the simple method is preferable for an general out-of-order sampling scenario and the tree-based approach will only provide better performance in long running scenarios resulting in high exponents and long distance out-of-order arrivals.

Fast throughput with Lookup Tables

If a high throughput of sample elements is required and the simple method can not deliver results for the out-of-order arrivals fast enough, all necessary priority values can be calculated in the beginning and hold accessible afterwards with the help of a lookup table. Because of the limit on maximum sample time and the handover to a consecutive iteration, the values which have to be initially calculated is bounded by a specific number. Therefore, also the lookup table is bounded in size and typically can range between a size of several kilobytes to multiple hundreds of megabytes, depending on the scenario parameters of the sampling

	Average Time	Performance
Lookup	0.49s	70%
Lookup (even)	0.50s	72%
Simple	0.69s	100%
Tree (10%)	0.78s	113%
Tree (complete)	1.59s	229%

Table 5: Results of different methods for out-of-order priority calculation on average in measurements

procedure. During the sampling procedure the requested priority values will be accessible in constant time afterwards, resulting in a drastic decrease of computational complexity but at the cost of an increase in storage requirements. The amount of priority values, which will be required to held accessible all the time is given with $|log_{\alpha}(\frac{fp_{max}}{P_0})|$ and influenced by the scenario parameters P_0, α and the maximum possible floating point number fp_{max} of the system. However, this number can easily be decreased to store only a certain fraction and start iterative calculation on the closest match, similar to the procedure in the tree structure. For example, instead of calculating all priority values for each point of time, only the priorities for an even time value are pre-computed and stored. For the even approach searching for a perfect or closest match come at basically no costs. If the requested time holds an even value a perfect match exists within the lookup table, if not its closest match is the direct successor of the requested element. Therefore, this approach will decrease the required values to store by up to a half, while keeping the property of a fast access.

Table 5 provides an overview of the performances of the different discussed methods, that showed up in the measurements. For each method the average time to process one complete scenario and its relative performance to the simple approach is depicted. The time to construct the different lookup tables were not included in these values, because they are not part of the general sampling procedure.
The results show, that the simple approach provides best performance if no pre-computation can be performed or no space is available to store the lookup tables. For any other scenario the lookup tables can provide a reasonable increase in performance.

5.2 Sampling Distributed Data Streams

In many state of the art scenarios, sampling is performed in a distributed manner. For example, if one imagines the current *Internet of Things* area with smart sensor nodes individual sampling can be performed on each node. However, as in general a combined sample is required in order perform analysis on the gained data as a whole, a merging strategy for

the different sub-samples is necessary. Furthermore, the resulting combined sample should contain the same statistical properties as each individual sample. Therefore, in the context of bias sampling this condition should not been violated. This section will introduce a method for EDS, which will support distributed biased reservoir sampling. Hereby, several samples maintained by different sampling nodes are combined to an overall sample through a coordinator instance. Figure 11 depicts such an scenario with three distributed sampling nodes s_1, s_2, s_3 and a coordinator instance s_c. For such a scenario, multiple possible solutions exist in order to support distributed sampling with EDS. The conducted method depends on the desired characteristics of the combined sample. Therefore, multiple solutions are proposed.

Exact Bias Sampling with full Reservoir Transmission

For a scenario, in which the induced bias by each individual sampling node should also be maintained in the combined sample, an additional uniform sampling of the different sub-samples can be performed. Because of the additional uniform sampling procedure the induced bias, towards maintaining more recent elements in the sample, will still be present in the combined sample. In order to generate such a sample the coordinator node will receive the different generated sub-samples with a fixed frequency completely and will perform the merging procedure after their arrival. The procedure will iterate over each element within the sub-samples and pick each element with a probability of $1/s$, with s being the number of different sub-samples. With the help of this approach a combined sample is generated, which follows the bias of the individual sub-samples if they are configured similar. The size of the combined sample can be much larger than the size of an individual sub-sample. In the worst case, the size of the combined sample would be equal to the size of all sub-samples summed up. However, this outcome will be highly unrealistic and the expected size of the combined sample will be equal to the size of the sub-samples, if they share the same size. As EDS does not maintain a static reservoir in size this characteristic can not be guaranteed, even for a scenario using the same parameters. Therefore, the expected size will get an additional increase and will be given with:

$$\mathbf{E}[n] = \sum_{k=1}^{s_{max}} \frac{I(k)}{s}$$

with s_{max} depicting the maximum size in the set of the sub-samples and a function $I(k)$, which symbolizes the amount of sub-samples that have an element at position of k.
Therefore, this approach will generate a combined sample from different sub-samples, which should always follow the expected size of the sub-samples while maintaining their induced bias.

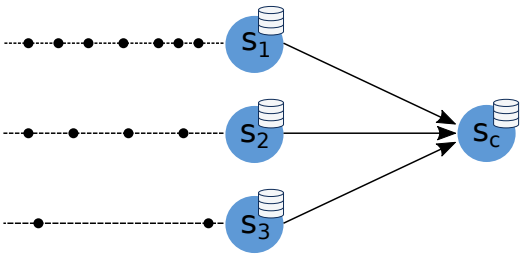

Figure 11: Distributed Sampling with different sampling and coordinator node(s)

Incremental Reservoir Transmission

In the context of distributed sampling scenarios the research is heavily concentrated on minimizing the communication effort of distributed sampling [**cormode2010optimal**]. Therefore, the presented first approach will not be optimal as the whole sub-samples are transmitted in batches and possibly not all elements of a sub-sample will be present in the final combined sample. For this reason, an incremental sampling approach is chosen.
In an incremental sampling approach the sample nodes will transmit new elements in the their sub-sample to the coordinator instance as soon as they occur. Therefore, the coordinator instance will consequently receive new sample elements. For this reason and that the combined sample is only maintained within the coordinator instance, the individual sampling nodes do not need to maintain their sample. This property will allow smaller sized machines for performing the distributed sampling procedure and therefore directly fits into the scenario of Internet of Things with small sensor nodes.
In order to support such a drastic change, while also maintaining the induced bias, the coordinator instance will need the information for how long an element should stay within the combined reservoir. Therefore, the protocol for transmission of new element from a sample node to the coordinator instance has to provide additional information. However, as the dropout scenario in EDS is deterministic this information can be computed after sampling a new element and can be attached to the information for the coordinator instance. In detail, the sampling procedure will follow this principle:
The individual sampling nodes will perform the proposed EDS sampling algorithm with the adaption, that the reservoir is not maintained locally in the node. Therefore, for each new occurring element the priority with its random influence is calculated. With this information the "lifetime" of each individual sampled element can be computed. The point of time, at which an element will drop out the sample is exactly given with $t_{dropout} = log_{\alpha}(\frac{p}{X_0})$ for an item priority value p and a static lower bound X_0. If the computed lifetime is lower than the actual time of sampling the element can directly discarded. For any other outcome the sampled element will be transmitted to the coordinator instance accompanied with its computed lifetime value.
The coordinator instance will collect the submitted pairs of its coordinated sampling nodes

individually for each node. With the help of the accompanied lifetime value the coordinator can dropout elements after a specific amount of time. For this reason, the coordinator will need to count the amount of transmissions for each sampling node and check for a possible dropout at this point of time.
This approach will allow incremental sampling of a distributed data streams while maintaining the sample within the coordinator instance. The different maintained samples of the coordinator instance will follow the same characteristics as it would if the sample would be maintained locally. Because of the differentiating of the maintained sub-samples within the coordinator instance, deviations in stream progression won't be a problem during the sampling procedure. Indeed elements in a sub-sample with a faster forwarding stream will dropout earlier as elements maintained from a stream with a slower progression and there will be no inference of each other. However, in order to achieve a combined sample an additional sampling step on top of the sub-samples might still be necessary, if a sample with smaller size as the combined sub-samples is required. Therefore, the presented Exact Bias Sampling or similar methods can be conducted.

Synchronized Interval Sampling

In both of the presented approaches, the coordinator instance s_c will collect the individual sub-samples in a separate storage and a merging procedure is performed as soon as a combined representation of the sample is required. However, in a specific scenario this requirement is not necessary and new sample element of the different sub-samplers can directly appended to a combined representation at the coordinator instance without performing an additional merging procedure. In such a merging less procedure the coordinator instance can not control and balance fast forwarding data streams versus data streams with a slow pace anymore. Therefore, the sampling nodes have to balance themselves and operate synchronized in the context of time progression during sampling. As the time progression within the EDS algorithm is not coupled to elements arrival and can also sample multiple element at the same timing step, a synchronized interval sampling will be possible. In such a synchronized interval sampling scenario, each sub-sampler will sample the observed data stream independently but timing steps are performed equally on each sub-sampler node. Therefore, a synchronization between all sub-sampler nodes is required. Such a synchronization can either be given with a synchronized wall-clock at each sub-sampler node and by enforcing a fixed interval of time progression or by an additional coordination protocol. For the fixed interval approach each sub-sampler will perform a timing step within the algorithm every time a specified time interval is passed. If the measurement of time progression is performed synchronized between each sub-sampler no further coordination is necessary. As a result, this method of sampling will generate multiple elements in the reservoir with same priority weight on each sub-sampler. These "batches" can directly appended to the combined sam-

ple of the coordinator instance, as it is guaranteed that they were performed at the same time interval. For further time intervals new batches with different priority weights will be generated. The coordinator instance can collect the different results from the sub-sampler either incrementally or in batches and can prioritize the recent batches against older ones, which will dropout after a specific size limit of the combined sample is reached. In order to discretize the resulting batches into smaller sized batches an additional synchronization protocol of time progression can be conducted. This protocol acknowledges the first element arrival after time progression of a sub-sampler to the different sub-sampler in the sampling network. If all sub-sampler have emitted such a acknowledgment message a time progression step will be triggered automatically. In order to prevent a deadlock situation, because of an inactive sub-sampler a timing step has still be performed after a specific time interval like in the previous result. This procedure will certainly induce more communication but can adapt more dynamically to fast forwarding data stream situations.

However, both approaches will produce a method for sampling a distributed data stream without performing an additional merging step at the coordinator instance. Nevertheless, this method will only be feasible directly in non event time based data stream scenario. For a event time based scenario, with potential out-of-order arriving elements no jump back in time can be performed directly anymore. As the time progression of the sampling method relies upon the fixed intervals, the matching interval for the out-of-order element has to be derived. Therefore, each sub-sampler has to remember such a matching during stream progression. With the help of such a matching, this scenario can also be performed in a event time based scenario.

6 Comparison to existing approaches

This chapter will provide a comparison between the well known and discussed methods from literature and the proposed EDS approach. In particular, the biased Reservoir sampling methods from Aggarwal are taken into account [**aggarwal2006biased**]. Hereby, the fixed reservoir sampling and its variable reservoir sampling counterpart are chosen for a comparison.

The procedure of EDS has some fundamental differences to the Aggarwal approaches. For the Aggarwal algorithm the reservoir is maintained without any further element information and dependencies between elements in the reservoir. Therefore, this approach follows a *memory-less* bias function. For such an approach the future probability of retaining an element in the reservoir is independent of its past history or arrival time. In order to support the EDS procedure of dropouts an associated priority weight has to be stored alongside the element in the reservoir for filtering old non recent elements consecutively. Furthermore, in contrast to the Aggarwal approach, EDS is not memory-less, as it respects the arrival time of an individual item. As EDS also support out-of-order arrivals this property is essential, in order to maintain a Reservoir, which follows event time progression appropriately. However during the sampling procedure an element remains in the reservoir independently from the history of the reservoir, similar to the Aggarwal sampling algorithm. Even more, for EDS, the time how long an element remains in the reservoir is deterministic and can be calculated after the element is inserted to the reservoir. In the Aggarwal sampling procedure this is not possible because elements in the reservoir will not dropout after time, instead they will be randomly replaced. This is also the fundamental difference between these methods as they induce the bias in a different manner. For the biased reservoir sampling approach the exponential bias is induced primarily in the replacement strategy, as for older elements the probability of a replacement is higher after some amount of sampling steps than for more recent elements. In the EDS sampling algorithm the bias is induced during the insertion with the help of the random influenced priority values and the dropout strategy afterwards. This behavior constitutes in a more complicated sampling procedure and higher computational costs. However, an optimized approached was proposed with a resulting amortized complexity of $\mathcal{O}(log(n))$ for sampling an individual element with a currently maintained reservoir

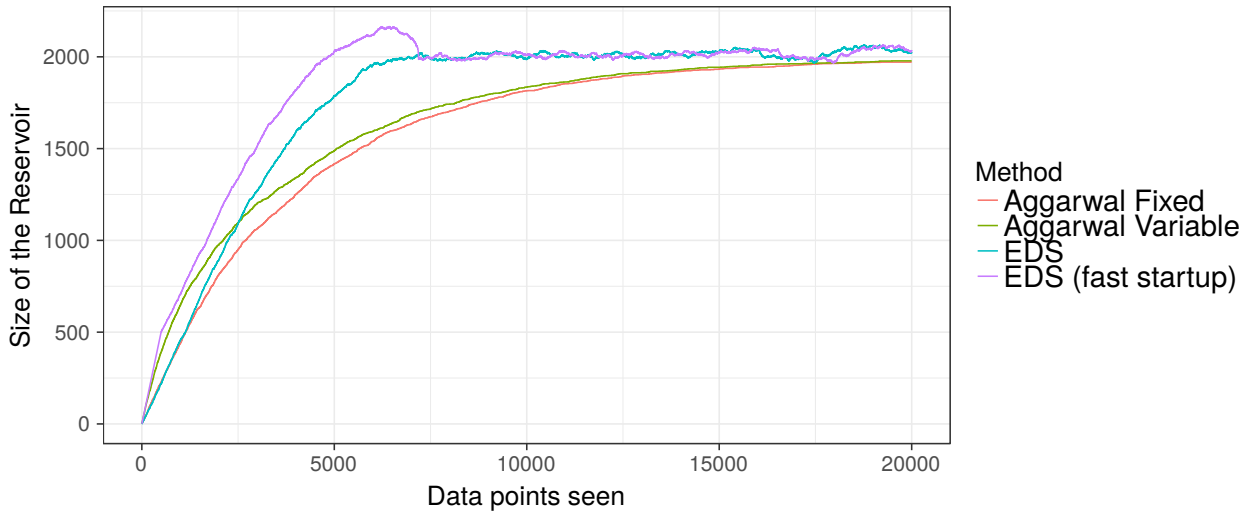

Figure 12: Reservoir size during stream progression of the different methods

of size n. As comparison both Aggarwal sampling algorithms will induce a computational complexity of $\mathcal{O}(1)$ for sampling an individual element. However, as the value n is bounded on the size of the maintained reservoir and not on the potentially infinite sized data stream the complexity can be seen comparable as they will only differ by some constant factor. This factor will differ during the progression of sampling as EDS does not maintain any fixed sized reservoir, unlike the Aggarwal and other popular sampling methods. Nevertheless, this value can be estimated with the help of a maximum bound or an expected outcome. Also its variance is known and therefore the size of the reservoir and the computational complexity can be predicted with high accuracy.

Figure 12 demonstrates the size of the reservoir for the different methods during an exemplary data stream and its progression. It can be observed, that EDS was faster in reaching the size of the requested size as the Aggarwal approaches, even for the EDS approach without any fast startup adjustments. After 7080 sampling steps, the EDS method reached the requested reservoir size, which was hold by the Aggarwal approaches at the end of the scenario. Afterwards, the Aggarwal approaches hold a static and fixed sized reservoir, as only replacement within the reservoir are performed. In the EDS approaches some fluctuations occurred, because of its random influencing priority values for each sampled element. However, the fluctuations are quite small and therefore the reservoir is always near its expected value. As discussed before, the Chernoff bound can be used to provide a probability estimate for the likeliness of a specific interval in size.

In order to provide a comparison of the induced bias of the different approaches, a further sampling scenario can be conducted. In Figure 13, a sampling of a stream with alternating ranges of zeros and ones was performed, which is symbolized as 'Stream' in the graph.

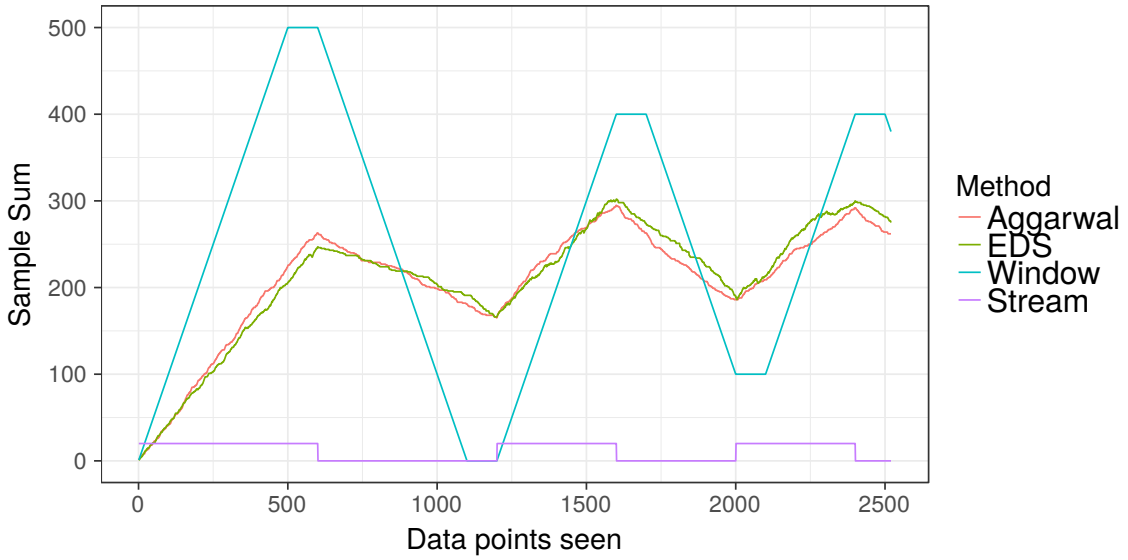

Figure 13: Sample sums of different sampling approaches using stream of zero and ones (Stream)

Figure 13 depicts the individual sum of the currently maintained sample for the different approaches with stream progression. For each method, a fixed reservoir/sample size of length 500 was chosen. Hereby, it can be recognized, that the sample sum of the Aggarwal and EDS approach followed a similar shape, in contrast to a sample using a Sliding Window approach. Because of the moving window, in the Sliding Window approach, steep changes in the sample sum occurred for any switch within the sampled data stream. The Aggarwal and EDS sampling methods showed a less steeper decay after a context switch from a high value stream to low value one. This characteristics appeared, because of their underlying sampling and replacement or dropout strategy. As the Sliding Window approach will sample any new occurring element and will dropout the oldest one in every iteration, the window will show a direct extract of the underlying data stream. The Aggarwal and EDS approach however, will include any new occurring element only with a certain probability (50% in the depicted scenario) and will also replace older elements only with a higher probability. Therefore, these methods are not affected for steep changes and will follow the stream with more historic data points within their maintained reservoir.

All in all, the EDS approach performed similar to the Aggarwal sampling methods, with some constant factor higher computational costs but with possible further applications like out-of-order arrivals or sampling in a distributed scenario. Both applications are not feasible to perform using the standard Aggarwal approaches. Also the induced bias showed a similar shape and therefore EDS can be performed in any scenario, in which Aggarwal can be conducted.

Recent research

In recent research most of the new developments in sampling following a biased decay were conducted to the area of (approximate) aggregates or quantiles processing. Such specialized algorithms can perform well under the use-case for a specific aggregates (such as minimum, maximum, average) or heavy-hitters in the data stream, but provide no generalized biased sampling approach for any generic application. For example [**cormode2008exponentially**] provides an efficient approach for computing quantiles and heavy-hitters over a data stream under an exponential decay. This approach also accommodates the requirement of out-of-order arrivals and can be therefore seen as comparable to EDS, if the requested aggregate is performed on top of the maintained biased sample. However, as an specialized approach the proposed method clearly outperforms EDS for such a scenario. Its computational complexity for maintaining a $\epsilon-$approximate quantile is $\mathcal{O}(log\, logW)$ for an insertion to a watched domain $[0...W-1]$ and $\mathcal{O}(\frac{logW}{\epsilon})$ for an afterwards query.
A more generalized research on decaying function was presented by [**cormode2009forward**]. In their results, the authors distinguish between a *backwards-decaying* bias function and their proposed term of a *forward-decaying* version of it. For a backwards-decay the priority of an item is defined at some point in time t and only valid for this specific point in time. As time is moving fast in the context of data streams such an approach can lead to problems for an efficient implementation because of the consecutively look "backwards" in the maintained reservoir. Therefore, the authors present the new term of bias functions, called forward-decaying. For a forward decaying scenario the priority of an element is not fixed to a moving point of time, instead to a defined *landmark*. With the help of this approach the priority values of the sampled elements are fixed upon arrival, which lead to a more efficient implementation. Following their definition, the proposed bias of the EDS algorithm is also forward-decaying, as their priority values are fixed after a performed sampling step. Hereby, the landmark is the monotonically growing priority function. The authors also conclude, that for sampling procedures following a forward-decaying bias function applications in a distributed or out-of-order arrival context are more feasible then for a backward-decaying counterpart. This result also followed during the analysis of sampling with EDS in such a context, showing the possibilities of such an induced bias.

7 Application

Different to the specialized approaches, like the previously mentioned heavy-hitters or quantiles computation for data streams, sampling will enable any generic application on the maintained sample. Therefore, the technique of sampling of data streams will adopt to any specific scenario and will even allow to extract multiple characteristics from the observed stream of data. This property might be essential for some scenarios, like feeding the maintained extract of the stream to different data mining algorithms or training different machine learning models at the same time. Even scenarios, that are not feasible to perform in an online setting are applicable to a data streaming scenario, with the help of the maintained sample.
For a biased approach in sampling, this property will also be embedded in any further analyzes, like in the trained machine learning models or outcomes from any Data Mining algorithm. As the maintained sample of EDS will yield such a bias and will therefore prioritize recent events before older ones any further analysis on top of its maintained sample will also contain this bias. But as data streams often impose a time dependency, this property might also be requested for any further analysis and hence a biased sample a feasible approach to deliver this bias into further analysis.

This chapter will provide such an application for the discussed sampling method on top of the *Apache Flink* stream processing framework. For this reason it will introduce the concepts and internals of Apache Flink, provides an implementation of the sampling method on top of the Apache Flink framework and will introduce an exemplary implementation of a Data Mining algorithm with the help of the maintained sample.

7.1 The Apache Flink Platform

Apache Flink [**Carbone2015ApacheFS**] is a universal platform for large scale data processing, for any analytics on unbounded data streams or finite "batches" of data. In particular Apache Flink provides an API for constructing a dataflow pipeline, which can be executed locally or distributed on multiple nodes in a network. Apache Flink consists of two core APIs. On the one side there is the *DataSet* API, which concerns finite sized data processing also known as batch processing and the *DataStream* API on the other side. A DataStream object

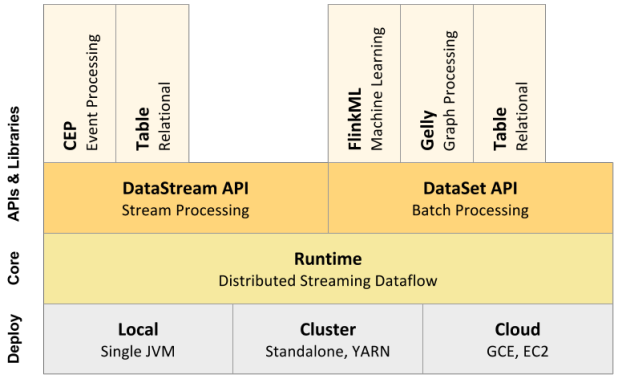

Figure 14: Structure of Apache Flink with DataStream/DataSet API and its extensional modules [**Flink**]

represents any unbounded stream of data, which has to be processed following a specified pipeline. Figure 14 depicts this core structure of Apache Flink and its potential extensional submodules on top of the DataSet or DataStream API. Each of the different APIs provide specialized operations on any data they represent, but there is a core functionality, which they have in common. The most important operators in Apache Flink characterize as [1]:

Map / FlatMap

The Map or FlatMap operator performs a transformation of the input data to different representation. Therefore, such a "mapping" has to be defined within the function but can contain arbitrary program code, as long as it is serializable by the platform and produces the defined output format. The Map and FlatMap operator only differ in the fact that each Map transformation needs to have a single output. A FlatMap operator can create multiple or an empty output for each call.

KeyBy / GroupBy

Using the KeyBy or GroupBy operator a DataStream or DataSet object can be transformed to a keyed representation. The key assists Flink to partition the data and is internally performed using a hash partitioning method. This method is necessary for the following Reduce or Fold operator and helps to perform such an operation in a distributed and parallel way.

Reduce / Fold

The so called *rolling reduce* operator of Apache Flink transforms the continuously occurring data elements to a combined representation. In detail the operator keeps always the last representation in its local state and performs a user-defined merging process on a new occurring element. The merged elements are feed forward for successive operators on the pipeline. In

[1] with current version of Apache Flink 1.3

order to be able to perform such an operation in parallel to others, a keyed partition is necessary and therefore a reduce operation has to be always combined with a keying operator beforehand. A fold operation behaves similar to the reduce operator with the one exception, that for a fold operation a starting value has to be defined. The reduce operator takes the first occurring element in the set or stream of data as a starting point for its internal state.

These operators can be combined with further transformation, filtering or aggregation functions like summation or maintaining the minimum/maximum of a specified value to form a combined processing pipeline. Because of the unbounded characteristics of data streams Flink provides further methods for performing micro-batching on top of their DataStream API. This method is available through the **Window** operator. The Flink framework embeds multiple window implementation to choose from, like *Tumbling Windows* or *Sliding Windows* and *Session based window implementations.* For each of the available windows a predefined size has to be provided. This is usually performed using a specific time range but defining a window with a specific size of elements is also possible. If the window is time-based, Apache Flink differs between three main time formats: Processing time, Event time and Ingestion time. The processing and ingestion times are defined using the wall-clock of the system. Hereby, ingestion time uses a centralized time measurement of the the Flink system if it is applied in a distributed environment. The processing approach relies on the wall clock of the each individual processing node. For the event-based approach the time information is embedded in any data object itself and a extractor for this information needs to be defined. In the event-based approach Flink supports also the technique of Watermarks in order to measure progress of the event time.
For all window functions a non key-based or key-based approach are existing in order to leverage the possibility of parallel window execution. Furthermore, defining a window on data streams are the only method to perform batch-like processing on a given set of data as whole directly. For this reason windows in Apache Flink provide a **apply** method call for user-defined transformations, with the whole window content as a parameter.

As a significant difference between Apache Flink and other large scale data processing systems like Apache Spark or Hadoop, Flink supports the concept of iterative processing pipelines, without reloading previous results from disk and performing the iteration on the program basis, like it would be the case for a Hadoop or Spark instance. This property provides a reasonable performance increasement for several applications. A typical example for iterative programs are training a machine learning model for several iterations or performing a data mining algorithm on the input. In order to define a iterative pipeline, Flink provides the **iterate** operator. For each defined iterator a step function has to be provided. This function evaluates the input and transforms it through the available operators from the

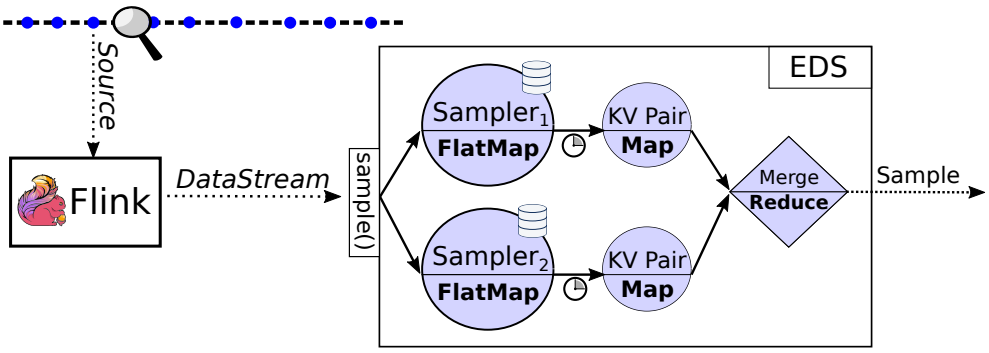

Figure 15: Structure of EDS sampling withing Apache Flink by using its core API

core API through a new iterative output. This output is either feedback for further iterations or provided as a finished output after a specific convergence criteria has been fulfilled. For data streams, the step function has to provide an output for every iteration, as its iteration may never finish through the continuously new incoming data points.

Using the presented operators from the core APIs in concatenation with different transformations applied a processing pipeline for data streams can be defined. Such a pipeline is also often referred as *Dataflow*, also in Apache Flink. A Dataflow represents a concatenation of the defined operators into independent stages, such that they can execute their processing concurrently. In Apache Flink the Dataflow is represented as a *Dataflow Graph*, which is a directed acyclic graph (DAG) with one or more sources and at least one sink. This DataFlow program is optimized internally, similar to query optimization in the database field, and transmitted to a *Job Manager*. The Job Manager of Apache Flink is responsible for distributing the different operations in the Dataflow program to the different compute nodes in the cluster and maintaining their state. Each of those nodes run a *Task Manger*, which can execute multiple operators and report their progress back to the Job Manger. Through this design the Dataflow program can be executed concurrently and distributed on the cluster and enables large scale data processing.

7.2 EDS Sampling with Apache Flink

In order to perform the EDS sampling algorithm with the Apache Flink framework, the algorithm has to be mapped to the presented core operators. However, by using Apache Flinks operators the procedure will also be performed distributed and therefore this behavior has to be concerned as well. Figure 15 provides an overview of the Dataflow, which will produce the biased sample of EDS in Apache Flink. The main sampling operation is performed distributed on multiple node in the cluster. In the depicted example, two sampler nodes

are performing the general sampling (Sampler$_1$, Sampler$_2$), within a FlatMap operator. The Apache Flink platform will concern the evenly distribution of the input between each sampler node and therefore balance the sampling procedure on multiple nodes. Every sampler node will maintain its own sub-sample and will perform sampling without any further coordination. The following Listing 7.1 shows a small excerpt of the FlatMap operation for each sub sampler:

```
1  override def flatMap(data: IN, out: Collector[(List[(IN, Double)]]) = {
2    // perform sampling operation
3    sample(data)
4    this.sampledItems = this.sampledItems+1
5    // search for possible dropouts elements
6    filter()
7    // emit new sample for further processing
8    if( shouldEmit() ) {
9      out.collect(this.reservoir)
10     this.sampledItems = 0
11   }
12 }
```

Listing 7.1: FlatMap operator for EDS sampling

This operator is called for each new occurring element (data in Listing 7.1) and will be the starting point for every sampling operation. It will trigger the internal sample method (Line 3) and perform a search for possible dropouts afterwards in the locally maintained reservoir (Line 6). For each sampled item a counter is incremented and the maintained sample is emitted either after a specific amount of sampled elements (for example n_{max}) or with a fixed time frequency (Line 8-10). As the sampling is performed within a FlatMap operator the emit of data elements has to be triggered manually, like it is performed with the *out.collect* call in Line 9 of the Listing. The filtering procedure can also be performed before any sample is emitted once, but as this operation can be performed very fast, because of the priority queue structure of the reservoir, this procedure will be performed for every sampled item. Otherwise, the maintained sample will grow quickly in size, as it will also store non recent elements.

After the emit, the different sub samples need to be merged to form one final sample. Therefore, the sample merging approach of Chapter 5 is used, which performs a uniform sampling on all sub samples in order to gain the combined sample. Such a combination of distributed results is performed using the reduce operator of Apache Flink. However, in order to perform such a reduce, a *KeyedDataStream* has to be created. For this reason, the emitted sub sample of the FlatMap operation is handed to a Map operator, that transforms the result to a key value pair representation. This tuple will be the input of the reduce operator, which will merge the different sub samples pairwise to form an overall sample. As the reduce op-

erator will emit each pairwise merged sub sample, even if not all sub samples have could have been considered an additional counting FlatMap operator can be appended afterwards. This FlatMap operator will count the pairwise sub samples, that have been already merged and will only forward this result to the following operations after all sub samples have been merged into the overall sample.
By using this design, the EDS sampling can be performed on top of the Flink platform and the different sampler nodes can be scaled and distributed to a larger range of nodes within a cluster. Only the finalizing reduce operation has to happen centralized on a single node, in order to provide a combined sample representation. The combined sample itself will be again of a DataStream type and either contain a combined list of the gained sample or a stream of the individual values. The specific characteristics of the sample (lower bound X_0 and decay ratio α) can be defined by a centralized property value, which will be distributed to each individual sampler node. Therefore, each sampler node will behave equally.

7.3 Distributed Decaying Frequent Pattern Analysis

If one considers the scenario of a data stream, which delivers transactional results, the underlying relationships and patterns contained in the data might be from further interest. The questions, that one might have can therefore be formulate as:

Which items share at least a specific co-occurrence? Can there be any implications derived from the data set?

As this scenario of mining the so called *frequent patterns* for the involved items in a large data set is known for many years, several algorithms exist, that deliver the frequent patterns and their corresponding *association rules*. A popular application for mining such relationships exist in market basket analysis resulting in better customer segmentation and more precise marketing efforts.

The Apriori Algorithm

A popular algorithm to extract such information of a data set is the so called *Apriori algorithm* [**agrawal1994fast**]. The Apriori algorithm requires a database D consisting of multiple *transactions* $D = \{t_1, t_2, \cdots, t_n\}$ as input. Each transaction t_i contains a subset of a set of possible items $I = \{i_1, i_2, \cdots, i_n\}$ that are available. As a result the algorithm will provide a set of all frequent co-occurrences of different length, that are contained in the database. In order to specify the amount of occurrences, which an item or co-occurring items should have in the database to be called frequent, the so called *minimum support ($minSup$)* is used.

The minimum support defines a percentile threshold, which a given co-occurrence of items has to exceed in order to be called frequent. This threshold needs to be provided to the algorithm beforehand. A low $minSup$ will deliver a larger set of frequent items and will therefore provide weaker implications as a higher threshold value of $minSup$.
Internal the algorithm is built upon the Apriori principle resulting in a "bottom up" approach to learn the underlying relationships:

Apriori principle [Goethals2009]

- If a set of items is frequent, then all of its subsets have to be frequent as well
- If an item is infrequent, then all of its super sets must be infrequent as well

With the help of the Apriori principle the item sets, which have to be considered as a candidate for a frequent item set can be minimized. As the naive approach would have to check every possible subset of I and their support, which will result in a total of $2^{|I|}$ necessary checks, a method for candidate reduction will result in better overall performance.
Therefore, the algorithm relies in every iteration on prior observed frequent item sets and grows them until no further growth can be performed or none of the resulting candidates can be called frequent anymore. In the first iteration, the algorithm will search for all one frequent item sets, meaning all items that have at least a $minSup$ occurrence in the data set. Every item, which occurs less then $minSup$ can be neglected for further iterations, because of the Apriori principle. The detected one frequent item set L_1 is provided to the next iteration, in order to search for all two frequent item sets L_2 and this procedure is followed for every larger frequent item set L_k.
In particular the Apriori algorithm splits into two major consecutive steps:

1. **Join step:** To find a new frequent item set L_k a join of L_{k-1} with itself is performed, in order to generate all possible candidates. This candidate set is referred as C_k. For the join an additional restriction has to be made: Two item sets out of L_{k-1} are only joined, if they share at least $k-2$ items. This property will guarantee, that C_k will only contain item sets with one additional item and will therefore be of size k.
2. **Prune step:** The resulting candidate set C_k will contain also item sets, which are not frequent according to the minimum support requirement. The naive approach to remove those non-frequent item sets would be to perform a database scan and remove all non-frequent item sets. For a better alternative the Apriori principle can be used: As an item set can only be frequent if its smaller subsets are frequent as well. Therefore, the filtering is performed using a comparison of all possible subsets with length $k-1$ for each candidate in C_k against the known frequent elements from the previous iteration

Algorithm 6: Apriori algorithm for mining frequent patterns [**agrawal1994fast**]

Input: D database of transactions, $minSup$ minimum support threshold

$L_1 =$ {frequent 1-itemset}
$k = 2$
while $L_{k-1} \neq \emptyset$ **do**
  // generate new candidates of size k through join of L_{k-1}
  $C_k =$ aprioriGen(L_{k-1})
  // remove unfruitful candidates
  foreach $(k-1)$-*subset* s *of* $c \in C_k$ **do**
    if $s \notin L_{k-1}$ **then**
      delete c in C_k
    end
  end
  // count support of candidates
  foreach *transaction* $t \in D$ **do**
    $C_t =$ subset(C_k, t) // get the subsets of candidates, that are contained in t
    foreach $c \in C_t$ **do**
      c.count++
    end
  end
  $L_k = \{c \in C_k | c.count \geq minSup\}$
  $k = k + 1$
end

L_{k-1}. If not all produced subsets of a candidate are present in the previous iteration, the candidate can be removed as it can not be frequent anymore. In order to provide fast results for such a matching a hash-tree can be used. For the remaining candidates their exact support has to be known and therefore a database scan is unavoidable. All remaining candidates, that share a larger support than $minSup$, are appended to the frequent item set L_k. This set can be used for a next iteration in order to find all frequent item sets of length $k+1$.

Algorithm 6 depicts the presented Apriori algorithm in pseudo code.
The Apriori algorithm is by far more efficient than the naive approach but nevertheless still has to perform multiple database scans in order to find all frequent item sets of length k. Also the candidates, that have to be considered for a frequent item set, will grow quickly. As an example, if there are 10^4 frequent 1-item sets, the algorithm will create more than 10^7 candidates for the 2-item sets. Therefore, different algorithms exists that avoid the costly candidate generation and will outperform the Apriori approach. A representative for such an approach without candidate generation is called *frequent-pattern growth* or *FP-Growth* [**Han:2000**].

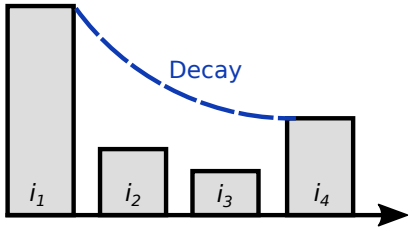

Figure 16: Frequent Itemsets (i_1, i_2, i_3, i_4) exponential decay of old entries in sample

Nevertheless, many improvements and implementations of the Apriori algorithm have been made, that increase the performance of the algorithm. Because of its structure the algorithm is also very suitable for a parallel or distributed computation. This property makes it possible to discover frequent patterns even in a scenario of Big Data. One approach, which exploits this property is a implementation of the algorithm using the *MapReduce* framework [**ParallelApriori**]. But also implementations using the cluster data processing framework Apache Spark [**rathee2015r**] and Apache Flink [**rathee2016exploiting**] have been made, which enable large scale frequent pattern analysis with the help of a distributed computation. The implementation on Apache Flink showed great performance compared to the other ones, as Flink supports iterative computation internally. Through this concept the intermediate results after each iteration can be preserved in memory for the next iteration and needs not to be reloaded programmatically.
However, the implementation only respects the DataSet API of Apache Flink and can therefore not be applied in a streaming scenario. Therefore, the following section will show a comparable implementation using the DataStream API of Apache Flink with using the obtained decaying sample as input.

Decaying Frequent Pattern Mining on Apache Flinks DataStream API

For Frequent Pattern mining on data streams many methods have been proposed. Most of the proposed approaches utilized a Sliding Window scheme (for example Moment [**chi2004moment**]) in order to follow the streams progression.

However Sliding Windows based approaches will induce a hard bound on the item history, which are used for frequent pattern analysis. Therefore, this section will present a method for mining the frequent patterns using the obtained decaying sample of the EDS sampling method. By relying on such a decaying sample as input also historic items have a certain probability of inclusion and there exist no hard boundary of inclusion, which enable a more continuous observation of the frequent patterns. The contained older values in the sample will also follow the exponential decay and will therefore gain less influence with stream progression. Figure 16 shows the decreasing influence of the frequent item set i_1 schematic. At the time of arrival, the itemset i_1 is dominating all other frequent patterns and this will also

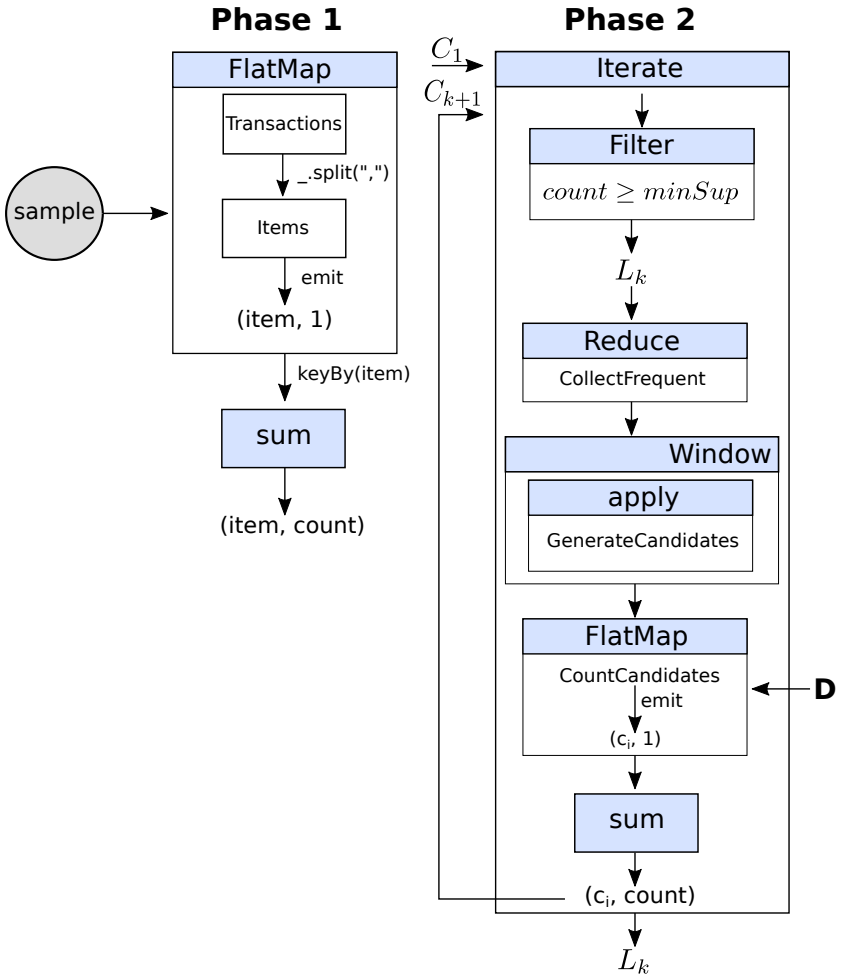

Figure 17: The two phases of frequent pattern mining on Apache Flink

extend through arrival of itemsets i_2 and i_3, until the decay function will limit the support to the support value of the new arriving itemset i_4. Therefore, the previous dominating itemset i_1 will have the same influence on the frequent patterns as the itemset i_4 does and after succeeding timing steps itemset i_1 will finally dropout of the frequent patterns set.

As Apache Flink supports iterative processing on top of their API and the Apriori algorithm also follows a iterative procedure an iterative DataFlow program is created [2]. Basically, the decaying frequent pattern analysis splits into two parts. First the decaying sample has to be generated. Therefore, the previously presented approach on Apache Flink is performed, resulting in a continuously emerging sample of the underlying data stream with the induced decay. After that, the frequent patterns can be computed. This procedure is performed into two phases. Similar to the basic Apriori algorithm in the first phase all 1-frequent item sets

[2] The complete Apache Flink application can be accessed at `https://git.rwth-aachen.de/georg.gross/eds-frequentpattern-flink`

have to be computed, in order to start the iteration for further frequent item sets. For this reason the obtained sample, containing the different transactions, has to be split into the individual items that have been observed. For each item the overall support has to be determined and matched against the provided $minSup$ constant. As this is basically only a counting operation, this can be performed on Apache Flink quite intuitively. A FlatMap operator transforms the obtained sample (represented as a list) into the individual transactions and their individual items. For each item a key value pair is emitted, containing the item as key and a fixed one as value: $(\text{item}, 1)$. This operation is performed distributed on each node within the managed cluster of Apache Flink. In order to gather a generalized representation and count for each item the emitted key value pairs are grouped around the specific item as key and a summation is performed for each group, resulting in an overall aggregate for each item. Figure 17 depicts the process of this Phase 1 graphically. The output of this operation is equal to the 1-frequent candidate set C_1.

In order to grow the 1-frequent item sets to k-frequent item sets the iterative approach of Phase 2 is conducted. As input for every iteration a candidate set C_k is provided. For the first iteration of Phase 2 the result of the previous Phase 1 will be used. Within the iteration, multiple operators of the Flink framework are combined in order to generate a new candidate set C_{k+1}. Additional to the new candidate set C_{k+1} the frequent item set L_k will be emitted.

In detail, the iterator will start with a filtering operation on the candidate set C_k against the provided $minSup$ value, resulting in the frequent item set L_k. This frequent item set will further on be used to generate the next candidate set. As the filtering is performed distributed, all frequent item sets first have to be collected with a reduce operation (*CollectFrequent*). Furthermore, the frequent patterns have to be known completely in a single set in order to perform the join step of the Apriori algorithm. Therefore, because of the property of the data stream, a window has to be inserted, which will collect all frequent item set in a given time range. With the help of the windowing operation, batch-like processing can be performed on Apache Flink DataStream API. The *GenerateCandidates* method will operate on the collected content of the window and emit all possible new candidates for a frequent item set. As a consequence of this procedure, this operation will be performed on a single node only. This potential bottleneck can be neglected, because the operation will only depend on the frequent item set L_k as input, which is typically far smaller than the resulting candidate set C_{k+1}. As next step the generated candidates have to be matched against the database D, in order to measure their specific support value. This operation (*CountCandidates*) is performed distributed again and for each match of a candidate within the database D a key value pair of the candidate c_i and one will be emitted: $(c_i, 1)$. The multiple pairs per key can be again aggregated to an overall sum and will form the candidate set C_{k+1} for the next iteration. The iteration will terminate after no new candidate can exceed the $minSup$ value.

As a result the proposed process will generate a stream of all frequent patterns with different length, which are given in the provided sample. Because of the decaying characteristic of

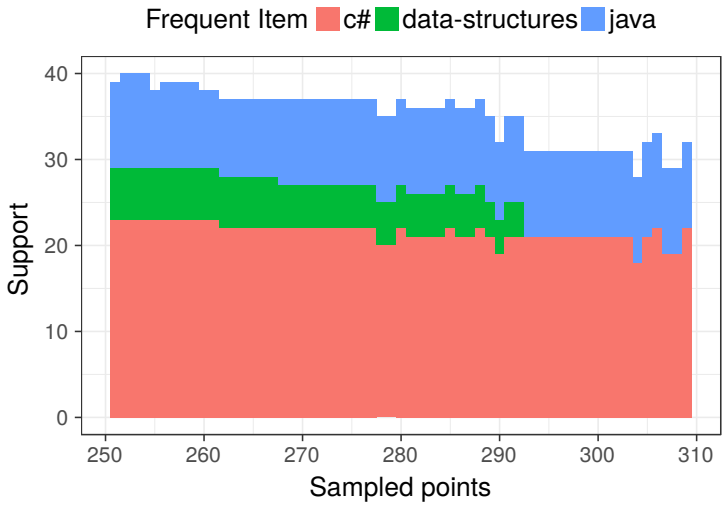

Figure 18: Frequent 1-itemset decay of unpopular Tag

the sample, this property will also be present in the discovered frequent item sets.

Decaying Patterns within the StackOverflow tags dataset

For evaluation purposes of the discussed method, a decaying frequent pattern mining analysis was performed on the StackOverflow tags dataset [**stackdata**]. This openly available dataset contains posts of the StackOverflow discussion board alongside with some assigned user-defined tags. In total, the data set consists of 300.000 entries, symbolizing an extract of posts in the discussion board of over a year.

The contained tags within the data set follow mainly a uniform distribution and therefore the collected frequent pattern show the same behavior. Most of the frequent patterns will remain during the sampling procedure, as they symbolize popular topics. Nevertheless, during the process of frequent pattern mining tags occurred, that have a high momentum at a specific point of time. Through this momentum and the sampling procedure of the underlying stream, these temporary popular patterns will also included to the collected online frequent item set. Figure 18 depicts such a development graphically. Beside the popular patterns "c#" and "java", which share a low variance in their support, also a temporary tag ("data-structures") is included to the frequent items sets. This particular tag got a temporary high momentum in the underlying data stream. Hence, this the induced decay of the support value of the frequent item set can be noticed. After the arrival of the last transaction, which contained this item, the item remained 25 succeeding steps in the maintained reservoir. During this succeeding step, the support value decreased continuously as more of its containing

transaction are evicted from the reservoir. Finally, its support value was lower than $minSup$ and the item is removed from the frequent item sets.

In general the method returned 25.6 1-frequent item sets and 3.7 2-frequent item sets on average in each iteration. For the $minSup$ bound a percentile value of at least 5% was used. This low value of the $minSup$ shows the wide range of possible tags that are contained within the data set and demonstrates the need for performing frequent item set mining, in order to perform some filtering for further processing. 3-frequent item sets were only noticed only once on average for every collected samples and no larger frequent item set could be retrieved for the specific scenario. For a representative the algorithm will produce the following frequent item set of an obtained sample with size ≈ 190:

1-frequent	**2-frequent**	**3-frequent**
windows, python, sql, database, visual-studio, language-agnostic, c, php, sql-server, vb.net, .net, asp.net, html, email, css, javascript, c#, mulithreading, version-control, svn, java, c++	(html, css), (version-control, svn), (javascript, css), (.net, c#), (javascript, html), (.net c#), (sql, database), (visual-studio, c#)	(javascript, css, html)

In the Appendix an additional plot of the distribution of the different frequent item-set with stream progression is provided.

8 Conclusion and Further Work

The scope of this thesis was to provide an analysis of the new proposed biased sampling method of Exponential Decaying Sampling (EDS). Because of its random influenced weight based structure, to form the reservoir and to induce the bias, EDS does not maintain any fixed sized reservoir. This main difference to existing approaches like Windowing or (biased) reservoir sampling made a further analysis of the obtained reservoir necessary, in order to be applied towards a practical application.

As first result, a maximum bound of its size was presented, which was extended to an expected size and its variance. It was demonstrated, that the obtained reservoir of EDS follow the Poisson binomial distribution. This result enabled further analysis, as existing bounds of literature can be applied to provide further estimation of the maintained reservoir. As an example, the Chernoff bound could be applied in order to provide confidence intervals of the maintained reservoir. This intervals were implemented in order to present a sampling approach, that does not need to reserve the maximum bound of the reservoir in storage anymore. With the help of this restriction, the storage requirement for sampling could be reduced by some large factor, as reaching the maximum bound of the reservoir in size is highly unlikely. This result also directly followed from the performed benchmark of the algorithm. The measurements of the benchmark supported the proposed theoretical findings and even showed that the Chernoff bound can possibly be replaced by a slightly sharper bound.

Furthermore, different approaches of initial fast reservoir startups were discussed. These approaches induce an additional bias in the beginning of process of sampling, in order to gain a filled reservoir more quickly. To achieve such a quick startup, different additional bias functions were introduced. Namely a constant, linear and a quadratic one. All functions resulted in a faster startup phase but also introduced an additional temporary elevation of the reservoir size. The quadratic approach showed the best balance between fast startup and lower elevation of the reservoir. However, it is still an open question if an approach exists, that does not introduce such an additional elevation. Future research can be conducted in this area in order to achieve fast startup without any additional elevation of the reservoir size. Beside of the benchmarks also an experimental comparison between the known method from literature of biased reservoir sampling and EDS was conducted. The experiments demonstrated, that EDS achieved a faster startup of its maintained reservoir as the popular Aggarwal biased sampling approach, while maintaining a comparable bias within the reservoir.

Different to the Aggarwal implementation, EDS also supports event time based sampling with respecting out-of-order arrivals. This property enables EDS to be applied to a wider range of applications. Any event time based data stream can be sampled with EDS, while respecting the event time progression of the data stream in the induced bias. This is an important property for many event time based scenarios. As in such a scenario a time skew between succeeding arriving elements can occur, this property has to be considered in the algorithm. For this reason, multiple approaches were discussed. Nevertheless, the intuitive method of direct computation every required element weight showed good performance results compared to further proposed methods.
As an additional extension through its design, EDS can also be applied towards a distributed scenario. Herein, the sampling will be performed by multiple distributed sampling nodes, which submit their reservoir to a coordinator instance. This instance will generate the combined representation for further processing. The generated combined sample hereby, should follow the same bias as each individual gathered sub-sample. The result from this section were applied within the Apache Flink stream processing framework and faced against a real world data mining task. The performed implementation of a distributed frequent pattern analysis of the obtained biased sample provided a good use-case for such a bias. The obtained frequent patterns of the StackOverflow tags data set, attended recent elements more attribution to older ones and also provided a soft handover of previous frequent tags to more recent occurred tags.
For future research, an adaptive variant of EDS can also be considered. In such an adaptive approach the requested reservoir can be increased or decreased during the process of sampling without changing the induced bias within the maintained sample. Approaches of adaptive reservoir sampling over data streams are existing and EDS can possible extended for such a scenario as well.

All in all, the biased sampling method of Exponential Decaying Sampling (EDS) presents a sampling technique, which derives comparable results as other popular biased data stream methods while enabling further applications. This result attributes the method to be conducted in further applications and research.

9 List of Figures

10 List of Tables

Appendices

Computational Results

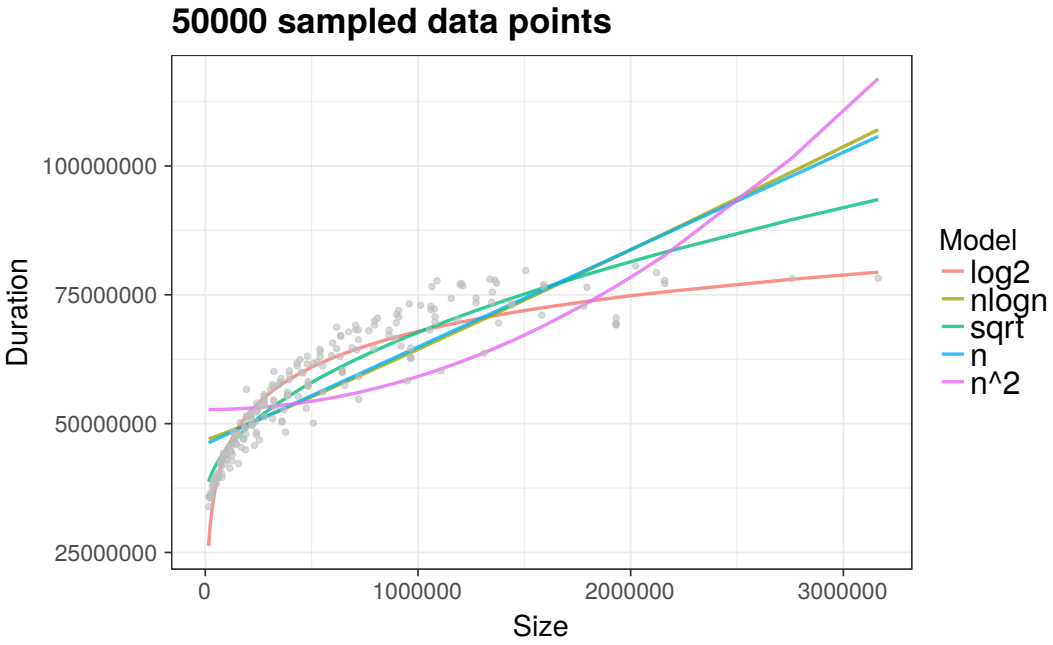

Figure 19: Duration for consecutively sampling 50k elements with different reservoir sizes involved

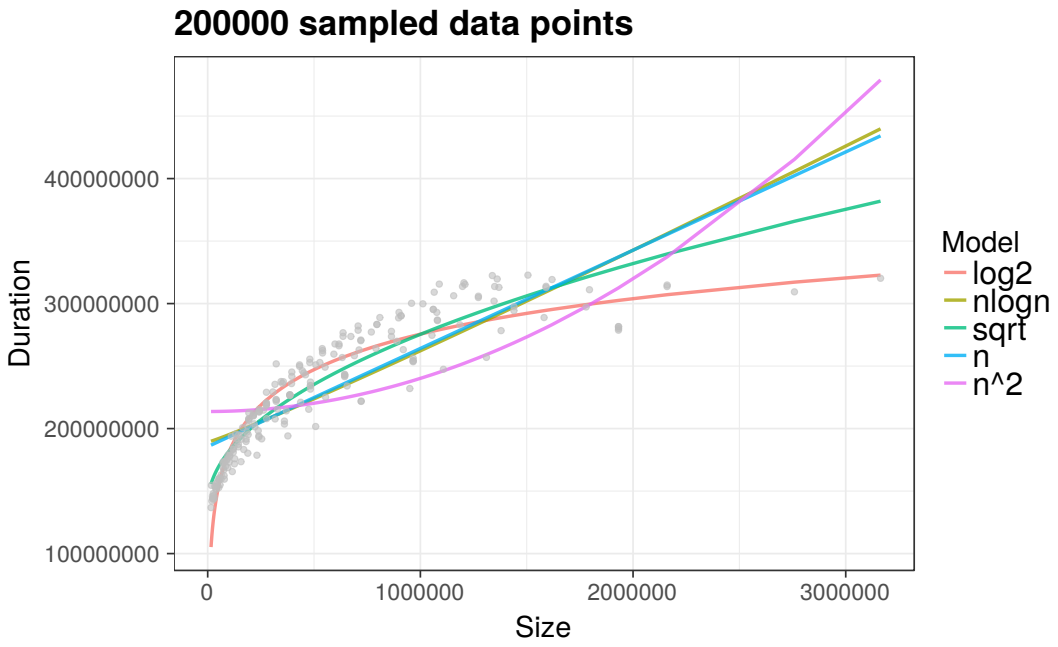

Figure 20: Duration for consecutively sampling 200k elements with different reservoir sizes involved

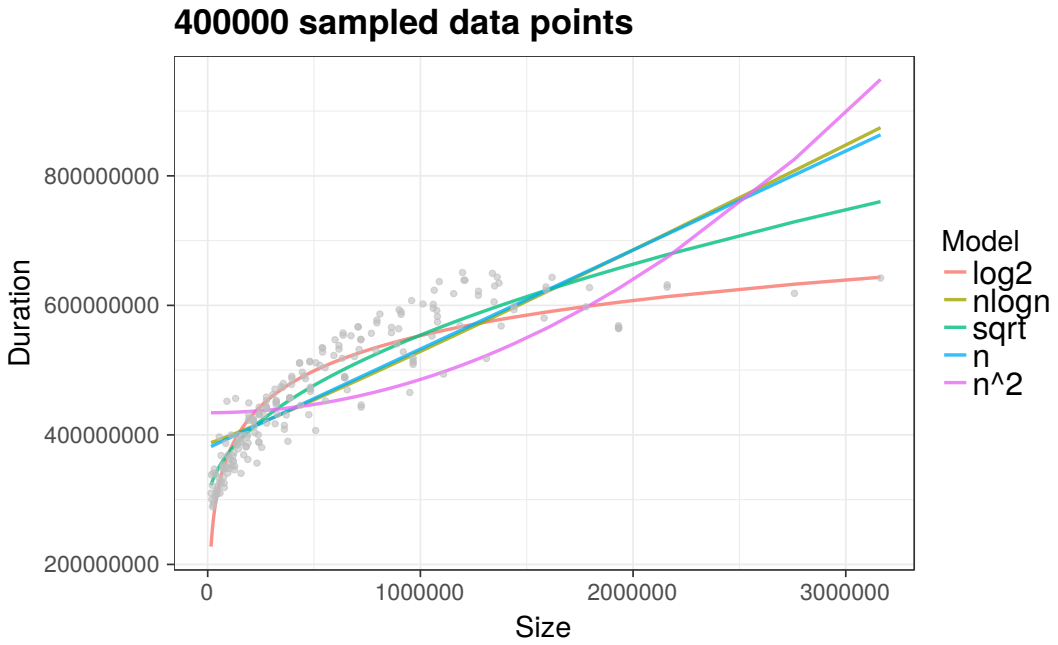

Figure 21: Duration for consecutively sampling 400k elements with different reservoir sizes involved

Frequent Item sets of StackOverflow tags data set

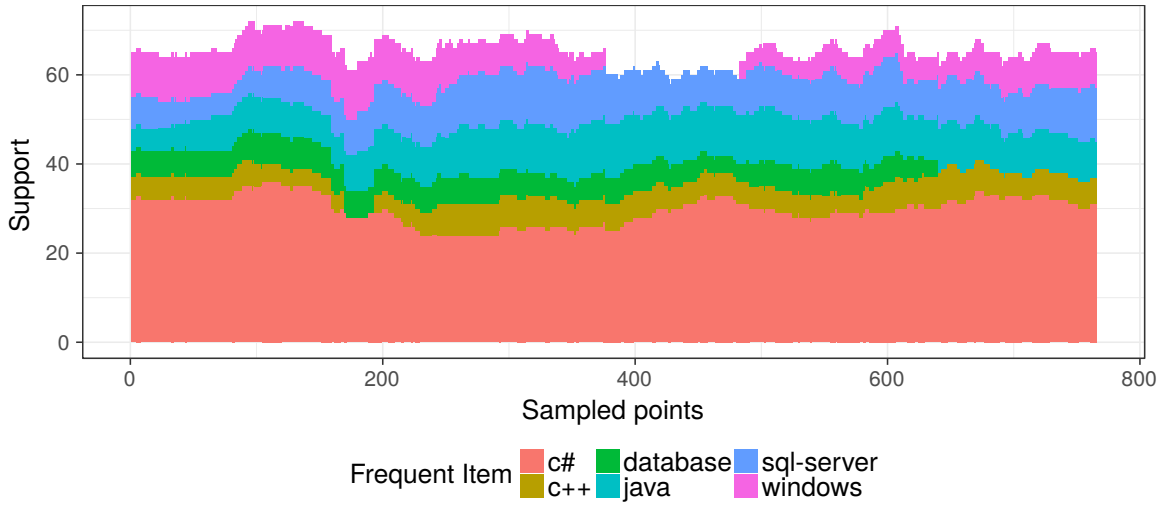

Figure 22: 1-frequent item sets over stream progression

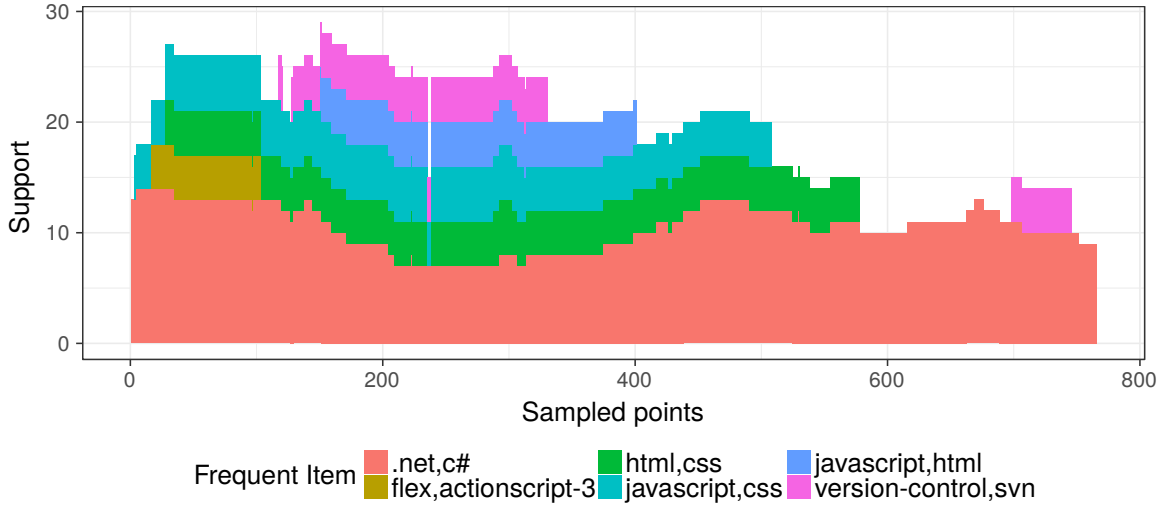

Figure 23: 2-frequent item sets over stream progression