

Advanced Sampling in Stream Processing Systems

Nikola Koevski

Thesis to obtain the Master of Science Degree in
Information Systems and Computer Engineering

Supervisors: Prof. Luís Manuel Antunes Veiga, Prof. Rodrigo Seromenho Miragaia
Rodrigues

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Luís Manuel Antunes Veiga
Member of the Committee: Prof. Nuno Manuel Ribeiro Preguiça

November, 2016

Acknowledgements

The work here presented is delivered as final thesis report at Instituto Superior Técnico (IST) in Lisbon, Portugal and it is in partial fulfillment of the European Master in Distributed Computing belonging to promotion of 2014-2016. The Master programme has been composed of a first year at IST, a second year's first semester at Royal Institute of Technology (KTH) and for this work and last academic term, based at the research lab INESC-ID Lisbon.

The work presented here is the final thesis report of the European Master in Distributed Computing at Instituto Superior Técnico (IST) in Lisbon, Portugal. The curriculum of the Master programme, which started in 2014 and concluded in 2016, was constituted of the first two semesters at IST, a third semester at the Royal Institute of Technology (KTH) and the last semester at the INESC-ID research lab in Lisbon, where this work was developed.

This work wouldn't have been possible without my two supervisors, Luís Veiga and Rodrigo Rodrigues, to whom I owe enormous gratitude. I am also especially thankful to the helpful advice of Sergio Esteves, who helped me through the hurdles of this work.

I am thankful to my parents, their support and their belief in the decisions I have made. Furthermore, I thank my whole family, who have always been there, to partake in my happiness, as well as to help with any difficulties I have had in these last two years.

Finally, I am thankful to all the professors from IST and KTH, especially my programme coordinators Luís Eduardo Teixeira Rodrigues and Johan Montelius respectively. They showed me what professionalism and dedication means and I will always be grateful for the time and input they supplied in order to expand my knowledge and capabilities.

November, 2016, Lisbon

Nikola Koevski

–To my parents

Abstract

The Big Data Revolution has caused an exponential growth in the amount of data that is generated. This growth, in turn, triggered an expansion in the methods with which this data is turned into valuable information. As the size of the data increased, so did the definition for fast and efficient data processing methods change. The batch processing methodology couldn't cope with the increased number of data sources and the rate at which they provide data. From this, a new method of processing data emerged, called stream processing.

Stream Processing is the new paradigm in data processing. It provides an efficient approach to extract information from new data, as the data arrives. However, spikes in data throughput, can impact the accuracy and latency guarantees stream processing systems provide. In order to cope with this data expansion, the system needs to be capable to scale its resources to meet this increased demand in resources. However, this may not be possible. Thus, an alternative is to reduce the amount of data. Currently, there are two methods of data reduction compatible with stream processing systems, load shedding, and sampling.

This work proposes data sampling, a type of data reduction, as a solution to this problem. It provides a user-transparent implementation of two sampling methods in the Apache Spark Streaming framework. Furthermore, a framework is implemented for the development of additional sampling methods. The results show a reduced amount of input data, leading to decreased processing time, but retaining a good accuracy in the extracted information.

Resumo

A revolução do Big Data causou um crescimento exponencial na quantidade de dados que são gerados. Este crescimento, por sua vez, provocou uma expansão na quantidade de métodos com que estes dados são transformados em informação valiosa. À medida que a velocidade de geração dos dados acelerou, assim também foi com a definição de métodos de processamento de dados mais rápidos e eficientes. A metodologia de processamento em lote (batch) não é capaz de lidar com o cada vez maior número de fontes de dados e ritmos a que estes são gerados. Assim, um novo método de processamento de dados surgiu, denominado processamento de streams.

O processamento de *streams* é o paradigma mais recente para processamento de dados. Oferece uma abordagem eficiente para extrair informação dos novos dados, assim que estes são recebidos. Contudo, picos no débito de entrada dos dados podem ter impacto prejudicial no cumprimento de garantias de precisão e latência oferecidas pelos sistemas de processamento de streams. Para lidar com esta expansão dos dados, o sistema tem de ser capaz de ser escalável em termos de recursos. No entanto, os recursos não são ilimitados. Assim, uma alternativa consiste em reduzir a quantidade de dados processada. Actualmente, existem dois métodos de redução de dados consistente com os sistemas de processamento de fluxo, load shedding e sampling.

Este trabalho propõe amostragem dos dados (sampling), como forma de reduzir o volume de informação a tratar, de modo a solucionar este problema. Oferece uma implementação transparente para o utilizador de dois métodos de sampling na framework Apache Spark Streaming. É também implementada uma framework para o desenvolvimento de métodos de sampling adicionais. Os resultados mostram que a redução do volume de dados de entrada leva à redução dos tempos de processamento, mas mantendo boa precisão na informação extraída.

Palavras Chave

Stream Processing

Sistemas de Computação aproximados

Redução de dados

Amostragem

Apache Spark

Keywords

Stream Processing

Approximate Computing Systems

Data Reduction

Sampling

Apache Spark

Index

1	Introduction	3
1.1	Context	3
1.2	Motivation	4
1.2.1	Current Shortcomings	5
1.3	Goals and Contributions	5
1.4	Document Structure	6
2	Related Work	7
2.1	Approximate Query Systems	7
2.2	Stream Processing Systems	10
2.3	Sampling Methods	11
2.4	Contributions	13
3	Solution	15
3.1	Use case example	15
3.2	Details on the Apache Spark Streaming Distributed Architecture	16
3.3	Sampling Algorithms	19
3.3.1	Congressional algorithm	21
3.3.2	Distinct Value algorithm	25
3.3.3	Algorithm Summary	27
3.4	Software Architecture	27

3.4.1	Implementation Details	28
3.4.2	Platform Specific details	28
4	Evaluation	35
4.1	Experimental Configuration	35
4.1.1	Assessment Criteria	36
4.2	Benchmarks and Assessment	36
4.2.1	Metrics used	36
4.2.2	Assessment goals	37
4.3	Apple NASDAQ Tweets	37
4.3.1	Discussion and analysis	37
4.4	US Technology Companies Stock	40
4.4.1	Discussion and analysis	40
4.5	New York Taxi logs	42
4.5.1	Discussion and analysis	42
4.6	Online Retailer	45
4.6.1	Discussion and analysis	45
5	Conclusions	51
5.1	Conclusions	51
5.2	Future Work	51
	Bibliography	56

List of Figures

2.1	Sampling as a Mediator of Constraints	12
3.1	The Music Videos Category and it's subcategories on a video sharing website with number of views per video	16
3.2	An Apache Spark Streaming DStream representation and an operation example over a DStream	17
3.3	Basic Architecture of Batching module in Spark Streaming	18
3.4	Batch Generation in Spark Streaming	19
3.5	Basic Architecture of Batching module in Spark Streaming with Sampling	28
3.6	Component Diagram of Spark Streaming with added Sampling Components . . .	29
3.7	Class Diagram of the OnePassSample Interface and the implemented algorithms	32
4.1	Results of data processing for the Apple NASDAQ Tweets: (a) Processing Time Speed Up, (b) Memory Variation, (c) Sampling Error	38
4.2	Results of data processing for the US Technology Companies Stock: (a) Processing Time Speed Up, (b) Memory Variation, (c) Sampling Error	41
4.3	Results of data processing for the Taxi Logs: (a) Processing Time Speed Up, (b) Memory Variation, (c) Sampling Error	43
4.4	Results of data processing for the Online Retailer Transactions: (a) Processing Time Speed Up, (b) Memory Variation, (c) Sampling Error	46

List of Tables

3.1	API modifications and Method Signature changes	30
4.1	Congressional Algorithm Results Summary	49
4.2	Distinct Value Algorithm Results Summary	49

Acronyms

IoT Internet of Things

API Application Programming Interface

FIT Feasible Input Table

RDDs Resilient Distributed Datasets

DStream Distributed Stream

1 Introduction

1.1 Context

Information has become the new currency in today's world. In order to gain more information, more and more data needs to be collected. However, this enormous amount of raw collected data is not inherently useful by itself. In order to gain practical information, the data needs to be properly processed and analysed.

In the past, information from this data was extracted with the help of *data mining*. Although effective, data mining had a big drawback. At that time, computers simply were not capable of processing all of the data in the time required for it to be valuable. Since information extraction was constrained by hardware limitations, many reductions and optimizations had to be performed over the data. The advent of *commodity hardware* allowed data processing to overcome this hardware obstacle. Furthermore, now that cheap hardware was available, constraints on the size of collected data had been significantly lowered. This resulted in a heavy increase in the volume of data, as well as the velocity with which it was collected. As a consequence, the need of a new paradigm in the field of data processing became evident.

The *Big Data Revolution* was a natural step forward in the data processing field. Big Data can be described with the "5 Vs" model ¹. First is the volume, or the amount of data that is available for processing. Next is velocity, or the speed with which these volumes of data are produced. Third, variety describes the diversity of the sources from where this data is generated. Next, veracity deals with the accuracy, or quality, of the data that sources generate, and the capability of Big Data systems to process this data. Finally, value represents the ability to convert the generated data into valuable information. Big Data processing enabled vast amounts of raw data to be rapidly transformed into useful data and insights of patterns and

¹Bernard Marr, "Big Data: The 5 Vs Everyone Must Know", <https://www.linkedin.com/pulse/20140306073407-64875646-big-data-the-5-vs-everyone-must-know>, (August 3, 2016)

future trends. In contrast to data mining, Big Data operates over whole data sets without having to sacrifice the amount of processed data to decrease the resulting delay. However, the way data is processed in Big Data systems led to the development of two different trends in Big Data processing.

When the Big Data paradigm first appeared, there already existed big data sets, which in the era of data mining had never been processed as a whole. Vast amounts of additional information and insights could be wrought out of this old data. Thus, the first trend of Big Data processing was to extract information from these big batches of data. Google's MapReduce paradigm, and its open-source implementation, Apache Hadoop (White, 2009), sparked a plethora of Big Data processing platforms, which constantly find new approaches of extracting information from the data they process. Since data in these systems is first accumulated and then the accumulated batches of data are processed, they are called Batch Processing systems.

However, as data throughput became higher, it became evident that the additional step of storing the data for batch processing impacts the latency of the results. This caused a new trend of Big Data processing to occur, where processing is done directly on the data stream of the producer of data, leading to the development of Stream Processing systems (Akidau et al., 2015).

1.2 Motivation

As mentioned in the previous section, lowering prices of hardware, as well as the improvement in hardware efficiency and network bandwidth have vastly increased the volume of data available for processing. In addition, data throughput has also significantly increased. Moreover, with the rise of the Internet of Things, as well as the increased dependence on the results of Big Data processing, the time interval in which results are considered fresh has become much shorter. This gave rise to the popularity of Stream Processing systems.

In spite of the advantage of stream processing, it has become apparent that the speed at which data is produced began to outpace the speed with which this data is processed.

Stream processing applications are constrained by the processing power of the hardware and by the time interval in which the results they provide are considered relevant. Since the hardware cannot keep track with the amount of data that is arriving in real-time, data starts to

build up in waiting queues. As a consequence, processing latency is increased, leading to delays in the results, which in turn may decrease the value they hold. Additionally, if a waiting queue fills its capacity, it may cause newly arrived data items to be dropped, producing an error in the results and, in extreme cases, may cause the system to run out of memory and crash.

An obvious solution to the problem is to add more machines to the cluster. This would provide the systems additional resources to cope with spikes in data throughput. As such, by increasing the size of data that is allowed through the system, this may alleviate the latency in the results (Das et al., 2014). Another alternative is to use controlled data reduction methods like load shedding (Tatbul et al., 2003, 2007; Tatbul and Zdonik, 2006; Sun et al., 2014).

1.2.1 Current Shortcomings

However, adding additional machines to the cluster may not be possible, since the cost of further increase in resources might be undesirable. Although increasing the data throughput of the system would enable for more data to be processed, this would increase the processing time, thus adding more latency to the results. Finally, even though load shedding is effective, it works by discarding data as it passes through the system. Because of this, it may skew the data distribution, lowering the result accuracy. In contrast, sampling (Krishnan et al., 2016; Goiri et al., 2015) decreases data size by producing a subset retaining the relevant characteristics of the whole data set. This provides smaller resource requirements and lower latency, but keeps a good accuracy on the result.

Despite the relative youth of stream processing and the Big Data paradigm as a whole, the problem of data size versus processing power is not a new one. It can be noted that data generation simply caught up with the advance of hardware and a recurrence of the problem that data mining was facing in the past can be seen.

1.3 Goals and Contributions

The goal of this work is to study how advanced sampling techniques can be used as a data reduction method in stream processing systems. For these purposes, a single-point, user-transparent sampling framework was implemented. The framework is coupled with the streaming library of the Apache Spark framework (Zaharia et al., 2013). Furthermore, by using the

sampling framework on top of Spark Streaming, two sampling algorithms were implemented to enforce data reduction. Finally, an evaluation of the solution's performance is carried out and the incurred advantages and costs of this usage in advanced sampling techniques in the accuracy guarantees of systems like Spark are discussed.

The result is an early-stage data reduction in the workflow, leading to a smaller processing load, and shorter execution times, while keeping a low result error.

1.4 Document Structure

The remaining structure of this document is organized as follows. The next chapter, Chapter 2, provides an overview on the current stream processing platforms, sampling methods and existing approximate computing systems developed with these platforms and methods. Next, Chapter 3 provides a description of the platform used for the solution, together with a general definition of its design and a detailed explanation of the implementation. Chapter 4 gives a description of the metrics and benchmarks used to evaluate the work. Furthermore, it provides a detailed discussion and analysis of the benchmark results. Finally, Chapter 5 gives a summary of the main points of this work and discusses future work.

2

Related Work

The solution described in this work is an approximate computing system. As such, it intersects the area of data reduction, by using advanced sampling techniques, with that of data processing platforms.

2.1 Approximate Query Systems

As mentioned in the previous chapter, the by-product of increased data generation, during the Big Data revolution, was that data processing systems couldn't cope with this heightened processing demand.

It has been established that for high rate streaming data, where transmission bandwidth and hardware resources are limited, maintaining a fast response time for queries and a summary of the data is much more preferable than trying to process the whole data that arrives (Duffield, 2016). In order to accommodate these requirements approximate computing systems have been developed. The goal of these systems is to employ various data reduction techniques to achieve a good balance between result processing time and accuracy, resource constraints and preserving data set characteristics (Cormode and Duffield, 2014).

Several of these systems have been developed on top of current popular Big Data processing platforms. These approximate computing systems use two approaches in data reduction. Many such systems employ load shedding to reduce the arriving data. This is done by probabilistically dropping certain data items. Another option is utilizing sampling techniques to reduce the size of the data.

While both methods have the same goal, load shedding works by cleaning the original data set of unimportant items. Thus, it focuses more on the data items and their importance, instead of the data set as a whole. By discarding each item based on certain pre-defined rules, its implementation can be more flexible, allowing for multiple points of data reduction. However,

because it doesn't take into consideration how discarding data influences the data distribution, load shedding has to adjust for error after it occurs. As a consequence, load shedding implementations have a higher overhead for calculating the error adjustments and have to provide data structures to keep the state of the load shedding operators. On the other hand, sampling focuses on the greater picture. It analyses the whole data set and probabilistically selects which items to include in a reduced set, so the data distribution is kept. As a result, sampling has to be performed at a single point in the application workflow. In addition, since sampling takes into consideration the data distribution of the data set, it attempts to provide the lowest error possible from the start.

The work by (Babcock, Datar, Motwani, et al., Babcock et al.) proposes a load shedding approach to approximate computing. The authors propose to distribute load shedding operators that would be able to discard data at any point of the system's workflow. This is done in order to minimize the error of the system, since discarding data in the beginning might introduce a higher data skew in the final, reduced data set. However, by introducing multiple points of data reduction, the probability of higher error is increased if the sampling rate of the load shedders is not properly balanced.

The work of (Tatbul et al., 2003) extends the Aurora framework (Carney et al., 2002) by employing the techniques suggested in the previously mentioned work. The solution tightly integrates load shedders into their system and employs a graph structure called a Load Shedding Roadmap (LSRM) to make its load shedding decisions. This alleviates the problem of improperly balanced sampling rates, but is a less flexible and much more intrusive solution. The implementation of the solution required changes in the workflow of the Aurora system. Moreover, the Aurora data processing system is not a distributed system, and, additionally it is only a prototype meant for academic research.

The authors of (Tatbul et al., 2007), as in the work on the Aurora approximate computing solution, propose a load shedding technique. This is implemented on the Borealis distributed processing engine (Ahmad et al., 2005), which is an update on the Aurora system towards a distributed system architecture. The solution addresses load shedding in a distributed environment, where the output of query operators may split into multiple downstream operators of the query path. This work allows for the usage of load shedders in this distributed environment by utilizing an advanced planning technique. Although this resolves some of the problems with the

solution implemented on Aurora, it introduces additional overhead in computation by employing the advanced planning technique. Furthermore, a data structure, called a Feasible Input Table (FIT) has to be kept throughout the execution of the query for the planning technique to be effective.

Another work on the Aurora/Borealis systems is (Tatbul and Zdonik, 2006). Similarly to the above mentioned works, it utilizes load shedding as a data reduction technique. The authors of this solution propose dividing the input data stream into windows. The system further encodes information to keep or discard about each window. If a spike in data throughput occurs, load shedders in the query path may probabilistically discard windows according to the encoded data sent by the system.

Similarly, the system (Sun et al., 2014) based on the Apache Shark (Engle et al., 2012) data warehouse system, uses load shedding by discarding blocks. The solution presents a fine-grained blocking technique that reorganizes the data tuples into blocks and generates metadata for each block. By evaluating this metadata, the system can choose which blocks to process and which to discard. However, as this solution is intended for a data warehousing system, it would require most data to be available in advance and thus is not a good solution for a stream processing system.

IncApprox (Krishnan et al., 2016) is built on top of a more established data processing platform, Apache Spark (Zaharia et al., 2010). Similarly to the solution described in this work, IncApprox uses sampling to reduce the input data. Additionally, it utilizes the incremental computing paradigm to increase the efficiency of the system. The solution uses Stratified sampling as an advanced sampling technique to sample over an already stored batch of data and generate a new, sampled batch. Next, it utilizes Spark’s caching mechanism to save the intermediate results, so it can allow for incremental computation. However, doing sampling on an already stored batch introduces additional computation in the system. This way the system has to spend resources to store data that will be discarded and additionally, to sample data that is distributed throughout multiple nodes with Spark’s RDDs.

The ApproxHadoop (Goiri et al., 2015) system employs both methods of data reduction. It uses multi-stage sampling as the first stage of data reduction and adds task dropping as a load shedding approach for the second stage. On the other hand, the system extends the Hadoop (White, 2009) framework, so it is optimized to work with more traditional data stores

and not with streamed data.

The BlinkDB system (Agarwal et al., 2013) is built on top of the Hive Query Engine (Thusoo et al., 2009). As a result, this approximate query engine can integrate with both Apache Hadoop and Apache Spark. Similarly to IncApprox, it uses the Stratified sampling technique to provide data samples. Additionally, it adjusts the sample size dynamically by considering response time and accuracy requirements. However, as with ApproxHadoop and (Sun et al., 2014), it is intended for more traditional data warehouses and not for streamed data.

2.2 Stream Processing Systems

At the moment of writing, there is an abundance of data processing platforms. Many of the currently popular data processing frameworks employ in-memory processing to decrease processing time and increase performance.

Foremost among these new frameworks is Apache Spark (Zaharia et al., 2010). Spark provides a streaming API called Spark Streaming to ease the development of stream processing applications. Since Spark was originally developed as a batch processing engine, its stream library implementation utilizes this batch oriented architecture. Spark Stream implements a batching module which aggregates the data into micro-batches which can then be processed as a regular Spark batch application. However, this solution introduces a delay while the data is accumulated into batches.

The relatively recent Apache Flink framework (Carbone et al., 2015) provides similar features to Apache Spark, including both batch and stream processing libraries. The difference is that, at its base, Flink has a streaming dataflow engine. A streaming data flow engine performs true streaming, meaning that each data element is immediately processed through the streaming application. Even though this is a faster implementation, it becomes an obstacle when trying to sample the data, since most sampling methods need to first build a sample set and then forward this set for processing.

Apache Storm (Toshniwal et al., 2014) is a distributed real-time computation system, whose processing engine has similarities with Flink's. Although, as with Apache Flink, it provides "real" real-time stream processing, it lacks any batch processing capabilities making it difficult to integrate sampling.

Apache Samza is another stream processing platform. The difference from the previous two, as well as Apache Spark, is that it is much more tightly integrated with Apache Kafka (Garg, 2013) for communication/messaging and Apache Hadoop YARN (Foundation, 2016) for resource management. Moreover, as Storm and Flink, Samza works on individual data elements, proving sampling difficult to implement on this platform. In contrast to Spark, it uses a streaming dataflow engine, as the two previously mentioned platforms. Thus, it performs true streaming, immediately processing each data element. As mentioned before, this becomes an obstacle when trying to sample data, since most sampling methods need to first build a sample set.

2.3 Sampling Methods

In the area of sampling, extensive research has been done on the usage of advanced sampling techniques in data stream environments.

The work by (Cormode and Duffield, 2014) describes the advantages of sampling in stream environments and an overview of sampling implementations in streaming. As seen in Figure 2.1, the authors describe sampling as a moderator of constraints. Sampling reduces the strain on hardware resources like bandwidth, CPU and memory. At the same time, it provides assurances regarding the result accuracy and the speed at which that result is obtained. Finally, with sampling, the resulting data set retains the data characteristics of the original data set, allowing for the original patterns and insights to be represented in it as well.

In (Hu and Zhang, 2012), a detailed description of sampling algorithms that can be adapted to stream environments is given. As with the above mentioned work, the authors here suggest that one-pass sampling algorithms are most appropriate for adaptation to streamed data. This type of algorithms can generate a sample from a single pass over a given data.

The Bernoulli sampling algorithm is the simplest of the sampling algorithms. It provides a fast, uniform sampling method, thus sampling each item with equal probability. However, Bernoulli sampling requires the size of the data to be known in advance, something that may not be possible for streamed data.

Reservoir sampling (Vitter, 1985), as Bernoulli sampling, is a uniform sampling algorithm. In contrast to the previous method, this sampling algorithm can generate a sample in a single pass over the data. Furthermore, the data size does not need to be known beforehand, or to be

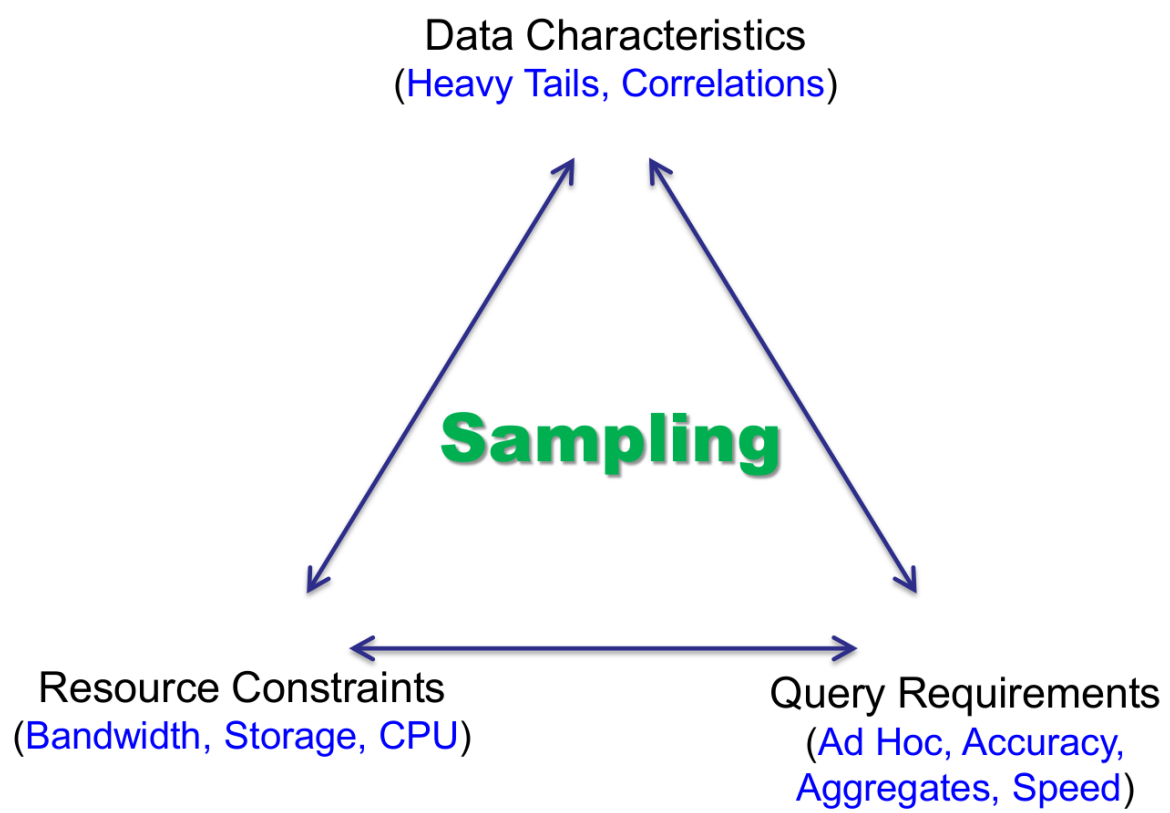


Figure 2.1: Sampling as a Mediator of Constraints

bound at all. Additionally, it provides a bounded error, but due to its uniform nature, similarly to Bernoulli sampling, may skew data distribution.

Concise and Count sampling (Gibbons and Matias, 1998) are sampling methods based on Reservoir sampling. Both improve upon the previous method, providing better accuracy. However, the Concise sampling algorithm continues to use uniform sampling, thus not removing the data distribution skew problem. On the other hand, the Count sampling algorithm employs a biased sampling method which removes the issue of skew. But, in contrast to the Reservoir and Concise methods, Count sampling does not provide error bounds.

Distinct Value sampling (Gibbons, 2001) is an algorithm of the Reservoir scheme. DV sampling is highly used for estimating the number of distinct values in a data set, so query optimization can be performed. It provides good accuracy with a low, bounded error of 0-10%. Although, as an algorithm that uses uniform sampling, it should introduce data distribution skew, it removes this problem by providing an upper bound on the number of items a single value can have in the sample, and randomly maps values to hashed values, so a uniform selection of original data is admitted in the sample.

The Congressional sampling (Acharya et al., 2000) algorithm was developed as a sampling algorithm for group-by queries. It is a hybrid of the uniform and biased sampling methods. As such, it gains the faster sampling time inherent to uniform sampling techniques. Furthermore, it performs a three-stage sampling process that allows for lower-occurring items to be included in the sample, thus providing a biased method to remove the problem of data skew. Finally, it provides a fixed error bound of maximum 10%.

Another one-pass sampling algorithm is Weighted sampling (Chaudhuri et al., 2001). The method samples each data item with a separate probability. Like Count sampling, it is a biased sampling method, thus able to handle the issue of skew in the data distribution of the sample. Nevertheless, Weighted sampling does not provide a bound of the error. Moreover, it requires information about the weights of the data items in advance, which introduces additional overhead

2.4 Contributions

This work was benefited by multiple past works on the topic of data reduction techniques. The work of (Hu and Zhang, 2012) provides an extensive overview of sampling techniques in data

stream environments. Furthermore, it gives a thorough analysis on which sampling techniques to use depending on different requirements.

Next, the works of (Acharya et al., 2000) and (Gibbons, 2001) were used to implement the sampling methods used for data reduction in this solution.

Summary

This chapter provided an overview of related works on the topic that this solution covers. First, it describes several approximate computing systems that share the same goal as this work. Next, the two areas that construct the field of approximate computing systems are detailed. An overview of the current stream processing platform is presented and finally, a description of additional sampling techniques that can be adapted to streamed data was given.

The next chapter introduces the architecture of the system, as well as its implementation details.

3

Solution

Chapter 2 described the work in data processing platforms and data reduction techniques. Furthermore, it describes approximate computing systems, the result of using data reduction methods in data processing systems. The solution proposed in this work represents an approximate computing system. This section first gives a use-case example to motivate a scenario where approximated operation would be more efficient over the standard operation of a data processing system. Second, it describes the basic architecture of Apache Spark Streaming, the system selected for the solution implementation. Third, an explanation of the choice of algorithms is given, and the operation of the selected two algorithms is defined. Finally, the chapter expands upon the platform-specific details of the implementation of the solution.

3.1 Use case example

A good example of a stream processing application is one that ranks the top N videos by category on a video-sharing website. Figure 3.1 shows the music category of such a video sharing website, together with several music subcategories. The top N ranked videos would need to be renewed at a short interval, for example, every minute. Thus, the streaming application might use a 1 minute sliding window interval. The application would need to recalculate the views for each video, every minute, which might belong to multiple categories and furthermore, the subcategory ranking would also need to be calculated. As the website becomes more popular, video views become more numerous, thus increasing the data input of the video ranking stream processing application. As the data throughput becomes higher, the system needs to utilize more resources in order to cope with this increase of videos and categories. Because the system receives a more substantial amount of data to be processed, the processing time of the data is also increased. Hence, the system has to start queuing new information about video views while taking longer time to process the currently viewed videos. The higher this processing latency, the later the new data is processed. As a consequence, the application may start reporting older

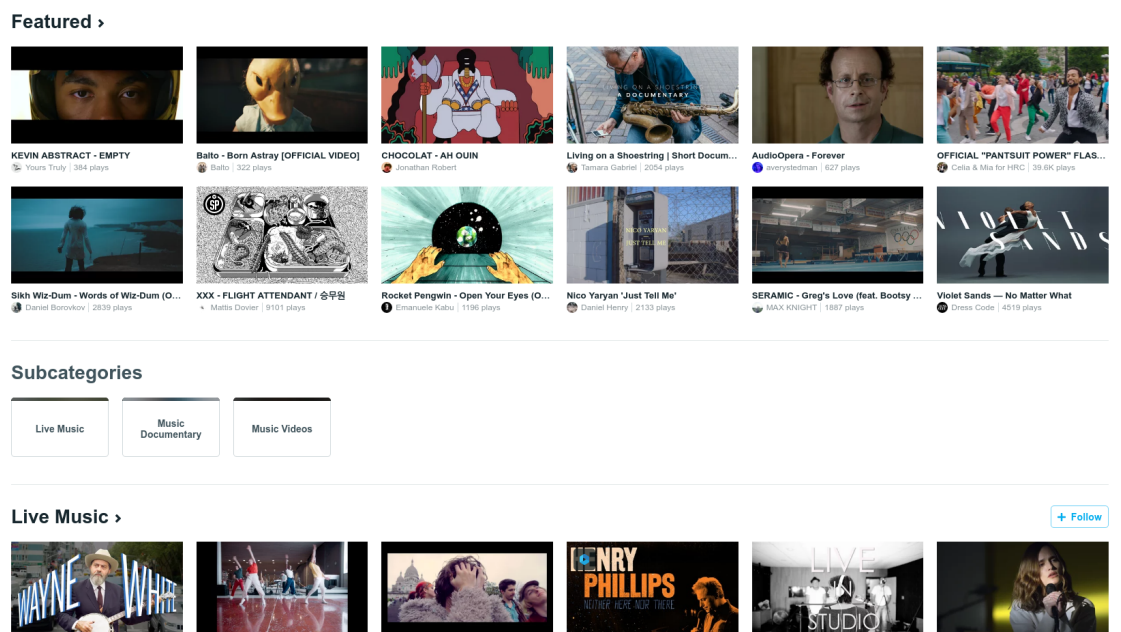


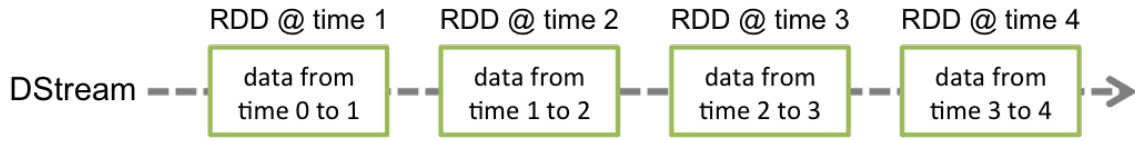
Figure 3.1: The Music Videos Category and its subcategories on a video sharing website with number of views per video

videos as the current top ranked videos. In the worst case, the waiting queue may overflow, causing the data processing system to crash and the the video categories feature to become of the website to become simply unavailable.

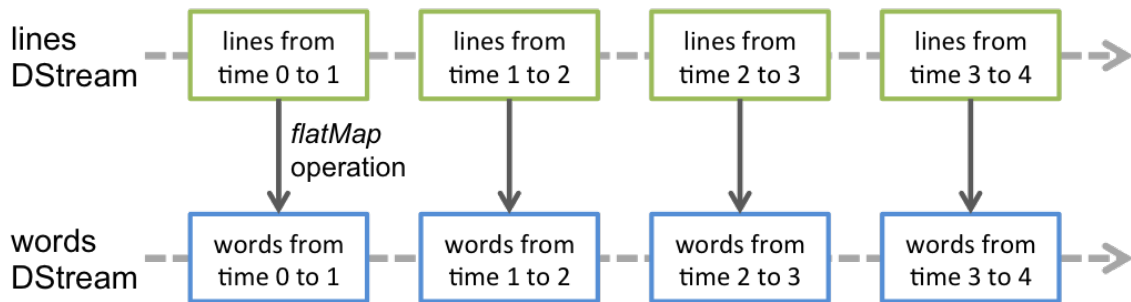
3.2 Details on the Apache Spark Streaming Distributed Architecture

As mentioned in the introduction of Chapter 3, Apache Spark was selected as the platform to implement the solution. Spark is a mature data processing framework. As a platform that performs processing in-memory, thus speeding up processing times, it is widely used as a replacement and upgrade on Apache Hadoop's MapReduce framework.

Furthermore, Spark Streaming implements stream processing as a continuous series of batch processing jobs. Spark Streaming provides a high-level abstraction of the stream, called a Discretized Stream or DStream. As seen in Figure 3.2a, the DStream is composed of continuous series of micro-batches, represented by Spark's RDD. Micro-batches are the reason Spark does not provide “true” real-time stream processing, instead each micro-batch is processed as a regular Spark batch application, as can be seen on Figure 3.2b. However, a spike in data throughput



(a) The Distributed Stream abstraction in Spark Streaming



(b) Data Processing Operation over a DStream in Spark Streaming

Figure 3.2: An Apache Spark Streaming DStream representation and an operation example over a DStream

can cause an increase in the batch size, leading to a delay in batch processing.

Moreover, Spark's modular design allows it to integrate with a multitude of different technologies, from Hadoop's HDFS for distributed storage, YARN or Apache Mesos for resource management, to providing libraries for connecting with data sources like SQL, Apache Kafka, Cassandra, Kinesis, Flume as well as Twitter.

These data sources provide a continuous stream of data which Spark Streaming processes. As seen in Figure 3.3, the data is admitted into the system through the Receiver module, represented as step 1 in the Figure. The Receiver provides Spark the flexibility to connect with data sources beyond the ones mentioned previously. Moreover, as shown on step 2 of Figure 3.3, it allows data items to be pre-processed before being admitted into the workflow. The Receiver then accumulates the data into blocks through the Receiver Supervisor, by forwarding the data items (step 3) to the Block Generator of the Supervisor, shown in step 4. When a data block achieves a certain size, the Receiver Supervisor proceeds to push the completed block to

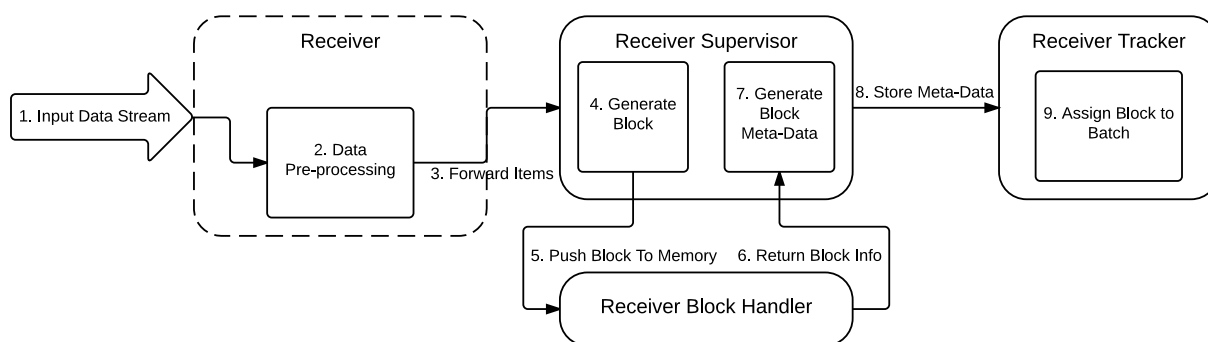


Figure 3.3: Basic Architecture of Batching module in Spark Streaming

memory, as shown in step 6 through the Receiver Block Handler. The Receiver Block handler then generates a block id, which is returned in a `BlockInfo` object to the Receiver Supervisor (step 7). When the Supervisor receives this block information, it packages it with the block size and generates the block’s meta-data, in step 8. In step 9, this meta-data is forwarded to the Receiver Tracker and is put into a waiting queue until it is assigned to a Batch job in the final step.

Next, Figure 3.4 depicts the continuation of the data flow through the batch job generation from the generated block meta-data. Every Spark application is built around a `SparkContext` object. This object provides information on how to connect to a cluster, contains application specific information and provides methods for generating RDDs. Spark Streaming provides a wrapper to the `SparkContext` ¹, called a `StreamingContext` ², which provides streaming capabilities to the application. The `StreamingContext` provides batch job generation through the Job Scheduler ³. The scheduler contains a `JobGenerator` class that utilizes a recurring timer to generate micro-batches at a user-defined interval, as seen in step 0 of Figure 3.4. Each time the timer runs out, a `generateJobs` method is called (step 1). This method then proceeds to call a block allocation method at the Receiver Tracker, which returns all the block meta-data that it has waiting in its queue (step 2). The `generateJobs` method then continues on to encapsulate this meta-data into a single batch job and submits this job for scheduling through a method of the `JobScheduler`, as seen on step 3 in the figure. The job is then scheduled for execution in the user-defined application. The length of the batch interval determines the size of the

¹“Initializing Spark”, <https://spark.apache.org/docs/1.2.0/programming-guide.html#initializing-spark>, (August 8, 2016)

²“Initializing StreamingContext”, <https://spark.apache.org/docs/1.2.0/streaming-programming-guide.html#initializing-streamingcontext>, (August 8, 2016)

³“Job Scheduling”, <https://spark.apache.org/docs/latest/job-scheduling.html>, (August 8, 2016)

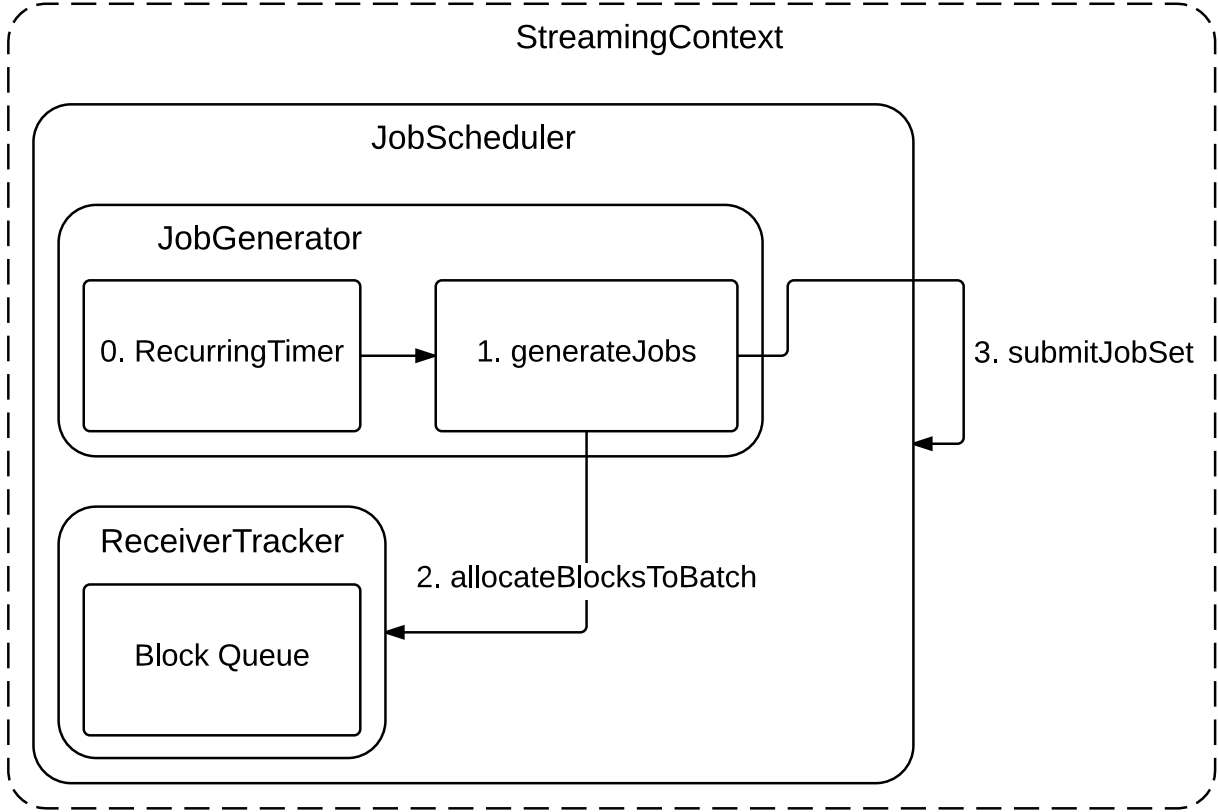


Figure 3.4: Batch Generation in Spark Streaming

micro-batches which are then processed by a user-defined streaming application.

3.3 Sampling Algorithms

Before the framework implementation, several sampling techniques from the uniform and biased sampling group methods were considered. By consulting (Cormode and Duffield, 2014; Hu and Zhang, 2012), the desired properties for the algorithms were determined and the following list of criteria were used for selecting the sampling methods:

1. Provide a fixed-size sample with a single pass over the data.
2. Prevent data distribution skew when sampling.
3. Provide accuracy guarantees for the sampled subset.
4. Provide timeliness guarantees for the sampling algorithm.

The first criteria provides a requirement that the algorithm will be capable of generating a sampled data subset with a pre-defined size by passing the original data set only once. This is the most important thing when sampling a data stream, since data items arrive only once and the final size of the data stream may not be known. The Bernoulli uniform sampling scheme doesn't satisfy this requirement. Bernoulli sampling methods require that the size of the original data set is known before-hand. Furthermore, they don't provide an upper bound on the sample size. The Biased sampling scheme doesn't always satisfy this requirement as well. Biased sampling methods require additional information to provide sampling, similarly provided before-hand as with the Bernoulli scheme. For streamed data, this may not be possible. Finally, the Reservoir uniform sampling scheme satisfies this criteria completely. It provides a reservoir data structure, which ensures a fixed-size of the sample. Furthermore, it will continuously sample the data stream until data items run out, independently of the size of the data stream.

The second criteria establishes that the sampling algorithm needs to implement techniques that will prevent the distortion of the data distribution of the original data set. This is important, since by only sampling highly occurring data items, the final sample might have a lack of rare data items, thus skewing the results of the application that uses this sampled set. Uniform sampling methods introduce this problem, since with these methods, there is an equal probability for each data item to be included in the sample. As mentioned before, this will enable higher occurring items to be allowed more easily into the sample, while less occurring items to have a decreased presence in this subset, which leads to data distribution skew. As a consequence, the Bernoulli and Reservoir sampling schemes don't satisfy this criteria. On the other hand, the Biased sampling schemes satisfy this criteria. Biased sampling algorithms sample each data item with a different probability, thus providing a more accurate admission of differently occurring data items in the sampled set. This keeps the data distribution of the original data set in the sample and prevents skew.

The third criteria deals with the accuracy provided by a sampling algorithm. Even though sampling algorithms strive to produce a sample that is representative of the original data set, some level of error is introduced nonetheless. Thus, sampling algorithms that provide a bound on the error that they produce are more desirable than algorithms that can produce an arbitrary sized error. Because of the uniform sampling method that Bernoulli and Reservoir sampling methods utilize, they tend to produce a higher error than Biased sampling methods. However,

accuracy depends on the algorithm-specific implementation as well, thus a general judgement of which methods provide bounded errors cannot be determined and is left to be evaluated with each algorithm.

The last criteria provides a requirement for the time with which a sampling algorithm generates a sample. In a streamed environment, where data items can arrive at an extremely high rate, the speed with which an algorithm can sample and produce a sampled subset is highly important. A slower sampling process may cause congestion in the workflow of the application and cause more problems than it tries to solve. Since uniform sampling methods sample each data item with equal probability, algorithms that use the uniform sampling method are faster than Biased sampling algorithms which require additional information, so each data item's probability can be determined. Thus, algorithms that implement the Bernoulli, or Reservoir sampling schemes are faster than those that implement a Biased sampling method.

To summarize, the first criteria eliminates algorithms that implement the Bernoulli sampling scheme. Although the Reservoir sampling scheme satisfies the first and fourth criteria, it fails to prevent data distribution skew and the uniform nature of its sampling method lowers the accuracy of the results. In contrast, Biased sampling methods satisfy the second and third criteria, but fail the fourth criteria. Thus, a solution was to select algorithms that rely on the reservoir sampling method of generating samples, but also use biased sampling techniques that can counter the data distribution skew and improve accuracy.

3.3.1 Congressional algorithm

Congressional sampling (Acharya et al., 2000) is an efficient method of performing sampling when data is partitioned in groups. The algorithm attempts to maximize the accuracy of a sample on a given set of group-by keys. A considerable number of data processing applications, foremost the MapReduce paradigm, use data grouping by key in order to implement their algorithms. Congressional sampling is a Reservoir sampling scheme, providing a one-pass algorithm for performing the sampling operation. Thus, it provides a fast, single pass method of generating a sampled set, satisfying the first criteria for selection. The algorithm, inspired by the organisation of the United States Congress⁴, where the Congress consists of two differently

⁴"Two Bodies, One Branch", <https://www.visitthecapitol.gov/about-congress/two-bodies-one-branch>, (September 18, 2016)

organised bodies, the House and the Senate, implements a three-stage sampling technique. The first, House, stage allows for item groups to be represented proportionately to their size in the data set. The second, Senate, stage assigns equal space to each group, while the last stage attempts to even out the sub-group representations in each group. By doing this, the algorithm uses a biased sampling technique at a higher level of the sampling process. On the other hand, each item is sampled with uniform sampling. However, the probability that it will be sampled in the House, Senate or Congress stage is different. Because of this, Congressional sampling is a hybrid of uniform and biased sampling. By guaranteeing that both large and small groups will be represented in the sample, the algorithm satisfies the second criteria for selection. By using this hybrid type of sampling, the method also addresses the poor accuracy obstacle introduced by uniform sampling, providing a bounded error of maximum 10%, satisfying the third criteria. Finally, the algorithm introduces an efficient method of calculating the sample slots that each group is assigned, as well as a decision-making method for picking the best of the House, Senate and Congress sets for the final sample. This, coupled with the sampling efficiency of uniform sampling, satisfies the last criteria for selection. The Congressional algorithm only requires two parameters for a correct execution. The first parameter is the sample size, while the second is a list of the attributes to group by. This allows for a simple and user-friendly usage of the algorithm. Algorithm 1 shows the algorithm for Congressional sampling.

In the first part, the algorithm performs the three stages of sampling. First, it performs a **House** (standard uniform reservoir) sample over all of the data. Since in this stage every item is sampled with the same probability, higher occurring items will be more likely to be present in the sampled set. Next, a **Senate** sample is performed, which assigns an equal slot of the sample size to each group. When a group slot is full, uniform reservoir sampling is performed, which may replace an item in the sample slot with a newly arrived item. The uniform reservoir sampling algorithm is described in Algorithm 2. Finally, a **Grouping** sample is performed. Since when grouping by key, several key attributes can be used, the grouping method samples an item for each attribute in the group set. As with the **senate** sample, an equal slot of the sample size is assigned to each value of an attribute. Correspondingly, when an attribute value slot is full, uniform reservoir sampling is performed to try to replace a sampled item with a newly arrived item.

When a certain event occurs, for example, the end of the batch interval in Spark Streaming,

Algorithm 1 Congressional algorithm

```

1: initialize(sampleSize, group)
2: for all item  $\in$  dataStream do
3:   sampleCount  $\leftarrow$  0
4:   houseSample  $\leftarrow$   $\emptyset$ 
5:   senateSample  $\leftarrow$   $\emptyset$ 
6:   groupingSample  $\leftarrow$   $\emptyset$ 
7:   if batchInterval  $>$  0 then
8:     doHouseSample(item)
9:     doSenateSample(item)
10:    for attribute  $\in$  group do
11:      doGroupingSample(item, attribute)
12:    end for
13:  else
14:    getFinalCongressionalGroups(groupingSample)
15:    calculateSlots(houseSample, senateSample, groupingSample)
16:    scaleDownSample()
17:    sampleCount  $\leftarrow$  0
18:    houseSample  $\leftarrow$   $\emptyset$ 
19:    senateSample  $\leftarrow$   $\emptyset$ 
20:    groupingSample  $\leftarrow$   $\emptyset$ 
21:  end if
22: end for

```

Algorithm 2 Uniform Reservoir Sampling algorithm

```

1: initialize(reservoirSize, itemCount)
2: Reservoir  $\leftarrow$   $\emptyset$ 
3: while item  $\in$  dataStream do
4:   if itemCount  $<$  reservoirSize then
5:     Reservoir.push(item)
6:   else
7:     position  $\leftarrow$  Random(0, itemCount)
8:     if position  $<$  reservoirSize then
9:       Reservoir[position].replace(item)
10:    end if
11:  end if
12: end while

```

the sample can be built. To build the sample, first the groups in the **Grouping** sample need to be defined. In order to do this the slot size for each group is recalculated from the attribute samples of that group.

$$GroupSlotSize = \frac{S}{mT} * \frac{N_g}{N_h} \quad (3.1)$$

Equation 3.1 shows the equation, where S is the sample size, mT is the number of distinct attribute values, N_g is the number of items in the attribute value slot that belong to the same group, N_h represents the total number of items in the attribute value slot. From these four parameters, the size of the sample S impacts the processing time of the application, as well as the sample error. Thus, a smaller sample size would produce a larger decrease in processing time, but also an larger increase in error. On the other hand, the mT , N_g and N_h show how many groups are present in the data set and impact memory consumption. Large mT and N_h values mean that more memory will be used for the data structures which keep track of all the groups. However, a large value of the N_g parameter means that a high number of items of an attribute overlap a certain group, resulting in a higher computation overhead in the slot calculation. Additionally, a combination of very small values for the S and N_g parameters and high values for the mT and N_h parameters may lead to much higher inaccuracy. This can result in a group slot size smaller than one, which may remove a whole group from the sampled set. The group slot is re-calculated for each attribute value of the group and the maximum value calculated, together with the corresponding attribute sample is submitted as the sample for that group.

In the next stage, the group sizes of the **House**, **Senate** and **Grouping** samples are evaluated and the final slot size for each group is calculated from the **House**, **Senate** and **Grouping** samples.

$$SlotSize_G = S * \frac{\max_{t \in Samples} S_{tG}}{\sum_{t \in Samples} S_{tG}} \quad (3.2)$$

In Equation 3.2, S is the sample size, $\max_{g \in G} S_g$ is the size of the largest slot for a group from the **House**, **Senate** and **Grouping** samples and it is divided by the sum of all the slot sizes (**House**, **Senate**, **Grouping**) for that group. Finally, each group is re-sampled with uniform reservoir sampling to generate a sample slot with the new size.

By employing three different sampling techniques, the Congressional algorithm prevents the introduction of skew in the sample data distribution. By using the **House** sample, which allocates more space for larger groups, it allows frequently occurring items to enter the sample. On the other hand, the **Senate** sample, by providing equal sized sample slots for each group, allows smaller groups to enter the sample. Finally, by using the **Grouping** sample, the algorithm optimizes the attribute representations inside each group.

3.3.2 Distinct Value algorithm

As its name suggests, the Distinct Value sampling method approximates the number of distinct values of an attribute in a given data stream. As with the previous algorithm, determining the distinct values of a certain attribute is frequently used in the optimization of the computation flow. The algorithm implements the reservoir sampling scheme, thus fulfilling the first requirement for selection. Although it uses uniform sampling, by employing a random mapping of item values to hashed numbers, the algorithm provides a method to obtain a more proportional selection of the items in the data set, thus satisfying the second criteria. This is explained more thoroughly in the explanation of the algorithm below. In addition, the DV sampling algorithm provides a low, 0-10% relative error, while providing a low space requirement of $O(\log_2(D))$, where D is the domain size of the attribute values. This satisfies the final requirement for selection.

Algorithm 3 presents the Distinct Value algorithm. Besides the sample size, the algorithm requires two additional parameters. The second parameter, called the *threshold* parameter, determines the maximum number of items that can be allowed in the sample reservoir per attribute value. The third parameter is the *domain* size, representing the number of possible values that can occur for the selected attribute. Furthermore, the algorithm uses a *level* variable, which controls which values are allowed to be sampled.

The algorithm works as follows. As each data item arrives, the domain size is used to generate a hash integer value of the item's attribute value. Next, if the hashed value is at least as large as the current level, an attempt to put the item in the appropriate hash value slot is performed. If the slot size is smaller than the threshold value, the item is simply placed in the slot. Otherwise uniform sampling is performed over the data item, as described in Algorithm 2. This can result in the new item replacing an item currently in the slot. When the items in the

Algorithm 3 Distinct Value algorithm

```

1: initialize(sampleSize, threshold)
2: level  $\leftarrow$  0
3: sampleCount  $\leftarrow$  0
4: Sample  $\leftarrow \emptyset$ 
5: CountMap  $\leftarrow \emptyset$ 
6: for all item  $\in$  dataStream do
7:   hashValue  $\leftarrow$  dieHash(item)
8:   if hashValue  $\geq$  level then
9:     if Sample(hashValue) < threshold then
10:      Sample(hashValue).add(item)
11:      CountMap(hashValue) = CountMap(hashValue) + 1
12:      sampleCount = sampleCount + 1
13:     else
14:       Sample(hashValue).sample(item)
15:     end if
16:   end if
17:   if sampleCount > sampleSize then
18:     sampleCount = sampleCount - Sample(level).count
19:     Sample(level).remove
20:     level = level + 1
21:   end if
22: end for

```

sample exceed the sample size, the slot whose value equals the current level number is removed from the sample and the level is incremented. The DV algorithm requires a hash function, called a *dieHash*, to be used in order to map the attribute values to hashed integer values. By randomly mapping the attribute values to hashed values and only allowing hashed values equal or greater than the current level to enter the sample, the algorithm ensures that the sample contains a uniform selection of the scanned portion of the data stream. As an addition, the threshold value prevents frequently occurring values from filling up the sample and prematurely incrementing the level. This prevents the occurrence of skew in the sample's data distribution.

From the above mentioned algorithm parameters, the sample size influences the processing time and accuracy of the application. As the value of the sample size decreases, the processing time is decreased, but a larger error is possible. Specifying the correct value of the data set domain size is very important. An incorrect domain size value will lead to improper mapping of values to hashed numbers in the *dieHash* method, resulting in additional computing overhead. However, the threshold value can have much higher impact on memory consumption and accuracy, and thus, its calculation is very important. The author of the paper that describes the

Distinct Value sampling algorithm suggests Equation 3.3 to calculate the threshold value.

$$Threshold = \frac{SampleSize}{50} \quad (3.3)$$

Since the threshold size determines how many items are allowed per attribute value, the parameter can severely impact memory consumption. In addition, because the over-filling of the reservoir is connected to the threshold, incorrect values to this parameter can lead to more frequent evictions of reservoir slots and increments to the level parameter, leading to a higher processing time.

3.3.3 Algorithm Summary

The previous two sections described the two selected sampling algorithms, how they work and the parameters on which they depend for correct execution. Even though both algorithms satisfy the criteria mentioned above, each has its own advantages and disadvantages. The Congressional sampling algorithm requires a simple input from the user. It needs only the sample size and the attributes to group by. However, the implementation of the algorithm is more complicated and sets a higher space requirement. This is because it needs to set up additional data structures for the House, Senate and Grouping samples which it later combines in a single sample. In contrast, the implementation of the Distinct Value algorithm is much more simple. It requires only a single reservoir for its sampled data, thus having a much lower space requirement. However, the Distinct Value algorithm requires two additional parameters to be provided by the user. Since the threshold and domain size parameters can greatly influence the performance of the algorithm, the user has to know the data set really well, and has to take great care in setting these parameters. This makes this algorithm less user-friendly than the Congressional algorithm.

3.4 Software Architecture

The micro-batch abstraction is what allowed a seamless integration of the solution with Spark Streaming. Figure 3.5 shows the sampling framework implemented as an extension of the Receiver module, called a Sampling Receiver.

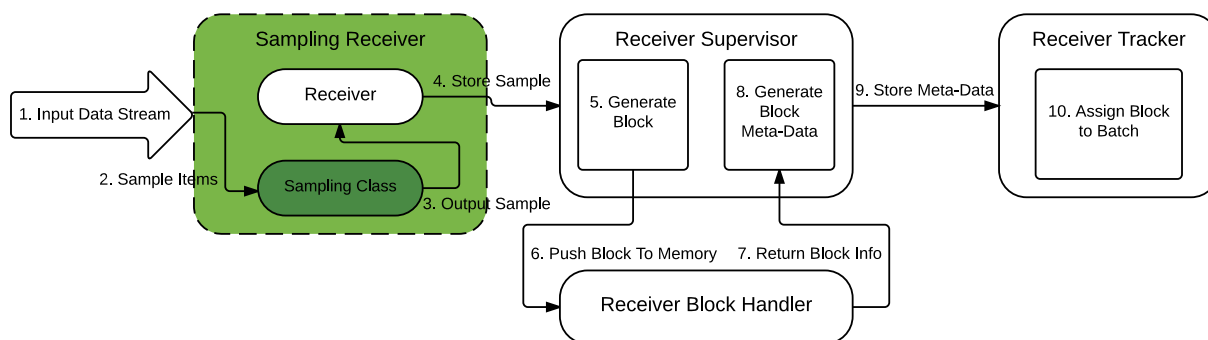


Figure 3.5: Basic Architecture of Batching module in Spark Streaming with Sampling

3.4.1 Implementation Details

The components of the framework can be seen on Figure 3.6, where the new components are coloured with red. The framework intercepts each data item through the `SamplingReceiver` class before it is stored. Here the item is passed through a class implementing the `OnePassSample` interface. Currently, there are two algorithms implementing this interface, the Congressional and Distinct Value sampling algorithms. One of these sampling algorithm classes samples each arriving data item and keeps the sampled items in a `HashMap` until the sample is requested by the `Sampling Receiver` class. Similarly to the job generation class described in Section 3.2, the `Sampling Receiver` class utilizes a recurring timer. The interval of the recurring timer is defined to be smaller than the user-defined interval of the application. This is done in order to allow for blocks to be built from the sampled data and stored in memory before the job generation timer calls for a new batch to be created.

When the `Sampling Receiver` timer runs out, the `Sampling Receiver` obtains the sample from the sampling class, restarts the timer and uses the methods provided by the `Receiver` to pass the sampled data to the `Receiver Supervisor`. The Supervisor uses the sampled data to generate blocks, thus continuing a standard batching operation. As a result, the old functionality of the batching module remains unaltered.

3.4.2 Platform Specific details

As mentioned in the previous section, the center of the framework is the `Sampling Receiver` class. This class extends the `Receiver` class, thus a `Sampling Receiver` object can be passed as an argument of the `receiverStream()` method of the `Streaming Context`. Table 3.1 shows

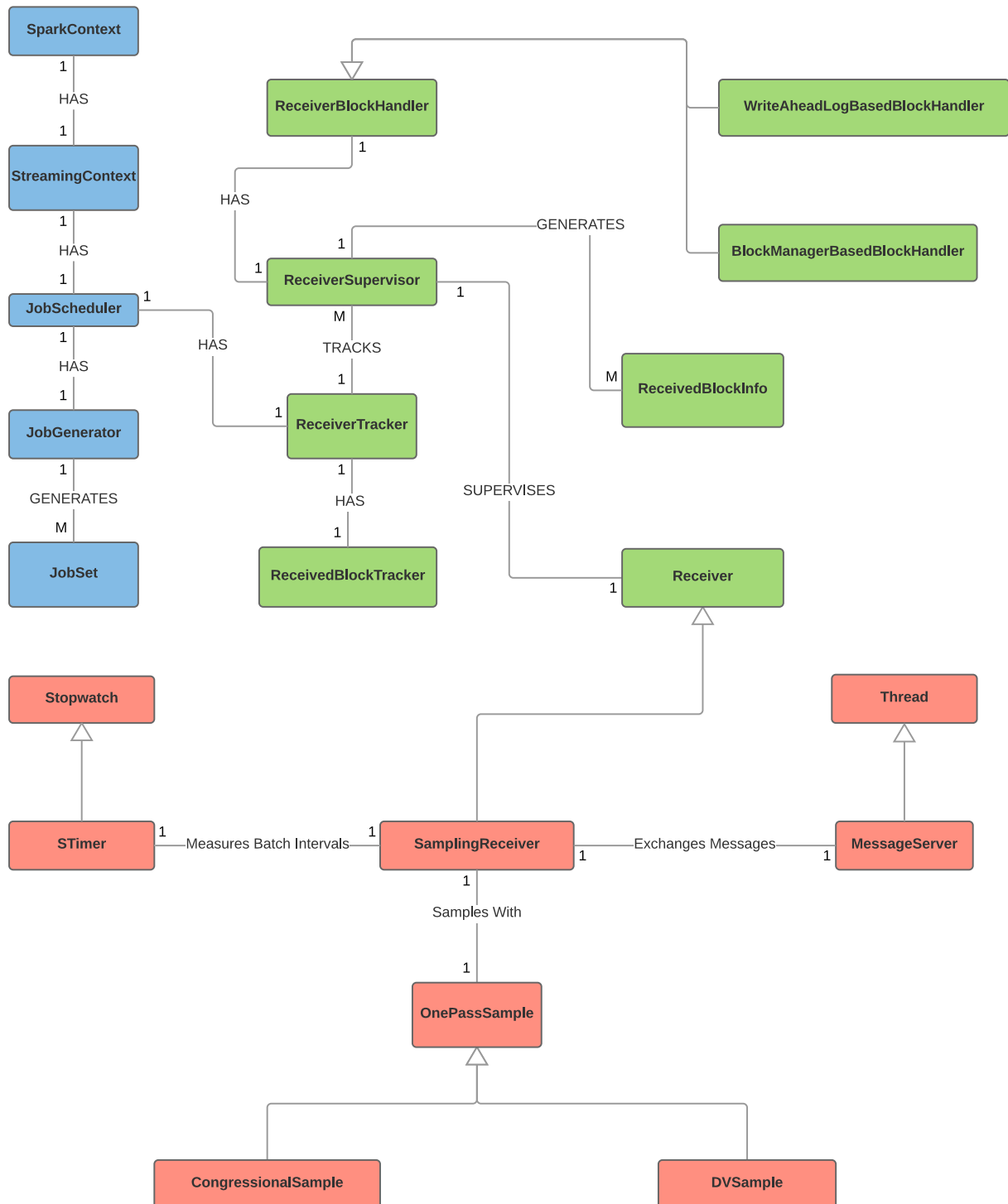


Figure 3.6: Component Diagram of Spark Streaming with added Sampling Components

the changes done in the Receiver class API and its method signatures. The SamplingReceiver provides a new constructor. In addition to the StorageLevel parameter, which determines how RDDs are stored, the constructor requires the length of the batch interval, the sampling algorithm class to be used for sampling and the port on which the message server will listen on. Furthermore, the class overloads the Receiver's *onStart()* and *store(T)* methods. While there are no changes to the method signature of the *onStart()* method, it is overloaded in order to start a thread which runs the interval timer class, STimer, which sends the sampled items to the Receiver Supervisor and resets the sampling class object. Furthermore, the *onStart()* method starts the message server that is used to communicate with the Streaming Context and provide the sampling error. The *store()* method is overloaded so that calls to it pass its argument to a *sample* method of the specified sampling algorithm class instead of the Receiver Supervisor. Additionally, two more methods are added to the API. The *storeSample* method is called by the STimer class to perform the sample generation of the sampling class and pass this data set to the Receiver Supervisor. Next, the STimer class calls the *newSampler()* method, which resets the sampling class, preparing it for the new data. As well as the Receiver class, the Sampling Receiver class uses type parameters to define the type of the data that is received, but it requires an additional type parameter to define the sampling class that is being used.

Table 3.1: API modifications and Method Signature changes

Receiver	SamplingReceiver
Receiver(StorageLevel)	SamplingReceiver(StorageLevel, BatchInterval, SamplingClass, MessageServerPort)
onStart()	onStart()
store(Item)	store(Item)
/	storeSample()
/	newSampler()

A feature for defining sampling classes is implemented through the *OnePassSample* interface is provided. Figure 3.7 shows the properties and methods defined by the interface, as well as the additional methods and fields defined by the implementing algorithms.

The interface defines the *sampleSize*, *sample* and *itemCount* properties. The *sampleSize* property is a number which defines the size of the sampled data set. The *itemCount* property is used to count the total number of items that pass through the sampling algorithm. This is

used in uniform sampling to determine if an item will be allowed in the sample or not. Finally, the *sample* property is a List object that contains the final set of items after they pass through the sampling algorithm.

In addition to the above mentioned properties, the *OnePassSample* interface defines several methods. The *singleSample(Item)* method should implement sampling when a single item is passed to the algorithm. The *listSample(List < Item >)* method is intended to implement sampling on a list of items. The *getSample()* method should return the final sample list after all the desired items have passed through the algorithm and the *calculateError()* method should calculate and return the sample error of the sampled data set. Additionally, the *getSampleSize()* and *setSampleSize(Integer)* methods are provided to implement the getting and setting of the size of the sample. The *getCount()* method should return the current number of items that have passed through the algorithm and the *reset()* method should reset the data structures of the sampling class for the sampling of data items for a new batch. Finally, a *doBefore()* method has been added which should be used to provide users with additional transformations or filtering decisions on data items.

As mentioned, the interface can be used for implementing additional one-pass sampling algorithms, which can be used in the Sampling Receiver class. The two sampling algorithms mentioned before were implemented by using this interface.

The **Congressional** sampling algorithm is implemented as a template class. It uses type parameters to define the type of the data being sampled as well as the type of the group-by keys. In addition to the data structures provided by the *OnePassSampling* interface it implements, the class keeps a *List* object of the attributes it should group by. Furthermore, the implementation uses *HashMaps* to represent the house, senate and grouping reservoirs. The Spark-specific *Tuple2* data structure is used in the grouping reservoir *HashMap* in order to enable the keeping of each separate attribute sample. Finally, a *get(item, key)* method is required to be overloaded so it can provide the attribute value of a given item.

An optimisation of the algorithm is performed by determining the largest sample size of a group while the group size of the grouping sample is calculated. In addition, the sum of the group samples is also calculated in this step. Thus, the algorithm is shortened by one step and the computation in the next one is simplified.

The **Distinct Value** sampling algorithm is implemented as a template class as well and

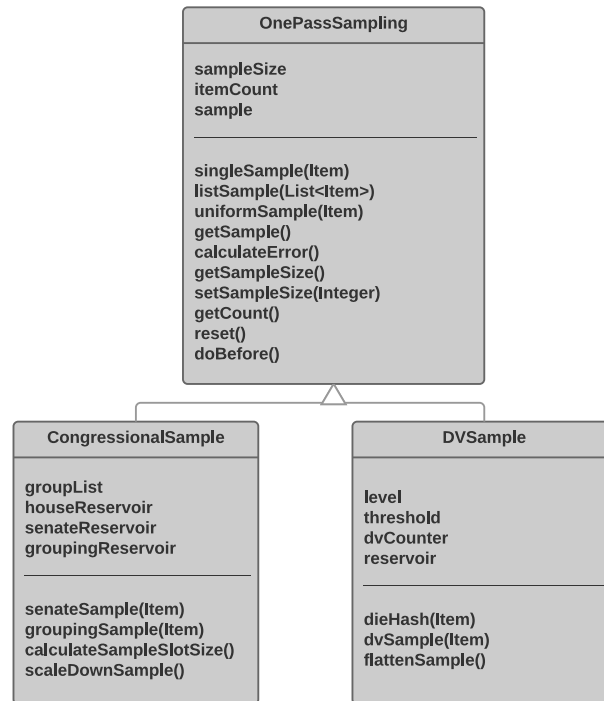


Figure 3.7: Class Diagram of the OnePassSample Interface and the implemented algorithms

it requires the type of the data being sampled to be defined as a type parameter. In addition to the interface-defined properties, it keeps the algorithm-specific threshold and level values. Furthermore, it utilizes the *dvCounter HashMap* to keep track of all the passed items per value and the corresponding *reservoir HashMap*.

Since the *dieHash* function of the Distinct Value algorithm described in the work by (Gibbons, 2001) can only work with integer values, an optimisation was performed. The DV algorithm in the solution implements a more general *dieHash* function that employs the *Random* class and a *HashMap* to map the attribute values to random integer values. Thus, the algorithm is not constrained to working on target attributes of type integer.

Summary

This chapter provided a use-case example where the solution described in this work is applicable. It continued on to introduce the inner workings of the Spark Streaming platform and the components of the platform that were used in the implementation of the system. Next, a set of requirements for the sampling techniques to be used with the system was defined and the two

selected algorithms were described. Finally, a more detailed explanation of the implementation was given together with a report of the platform specific methods and data structures used.

The next chapter discloses the experimental evaluation performed by using the solution described in this chapter.

4

Evaluation

This chapter describes the experimental configuration that was used for the evaluation of the solution. Next, the solution is evaluated by using accuracy as a qualitative metric and memory usage and throughput as quantitative metrics. First, the assessment criteria are explained. Second, the benchmark applications used in the evaluation are described and the attained metrics for each benchmark application are discussed. Finally, a summary of the gathered results is provided.

4.1 Experimental Configuration

For the experimental evaluation a single server was used. The configuration of the server was a 8-core, 2.93GHz Intel i7 with 12GB of memory running a 64-bit Ubuntu 14.04 LTS operating system.

On the software side, version 1.6.0 of Apache Spark was used. The input data was streamed locally through the Netcat Linux command line tool ¹, provided with the operating system. Data items from the data set files were streamed to Apache Spark every 10 milliseconds.

A lightweight console application, Jvmtop ² was used to measure the heap memory usage of the benchmark applications.

Each of the benchmark applications was run for 25 batch processing intervals, resulting in a total application runtime of 10 minutes. The applications were first run in normal mode, to acquire the metric values of a normal run. Then, each was run in sampled mode, with a sample size of 2, 5, 10 and 25%. It should be mentioned that for the DV sampling method, several values for the threshold parameter were tried. This was done to confirm that Equation 3.3 would provide correct values even in a streamed environment. Since even small changes

¹Giovanni Giacobbi, "Netcat", <http://netcat.sourceforge.net/>, (August 8, 2016)

²"Jvmtop", <https://github.com/patric-r/jvmtop>, (August 8, 2016)

in the threshold value showed much larger changes in the measurement of the metrics, it was important to examine it. The experiments showed that the specified threshold equation provided the optimal threshold value for each sample size. At the end of each run, the total execution time and error estimate of each batch was collected and the average value for the metrics was calculated. The same was done for the heap memory usage of the application during the execution.

4.1.1 Assessment Criteria

In order to gain a better understanding of the acquired gains of the implemented system, three assessment criteria were used. From them, two are performance metrics, evaluating the speed-up in processing time and the variation in memory consumption. The last is an error metric, estimating the relative error in the generated sample of the benchmark applications.

4.2 Benchmarks and Assessment

In order to better test the performance of the system, four distinct benchmark applications were used. The goal was to measure the execution of the work with data sets with different data distributions.

4.2.1 Metrics used

The metrics used for the evaluation of the work include the benchmark applications' total execution time, memory consumption and the sample error. All of the metrics are expressed in percentages. For the sampling error metric, the relative percent error was calculated. For the execution time and memory consumption, the relative number representation was to show the general difference between a normal execution and a sampled execution of a benchmark application. Thus, a more generic conclusion can be deduced from the evaluation results from the various applications that were run.

4.2.2 Assessment goals

There were several assessment goals to be achieved through the evaluation results. The first goal was to show the speed up of the total execution time on a sampled run of a benchmark application over a normal run of the same. The second was to analyse the differences in memory consumption between the sampled and normal runs. Next was to evaluate the size of the error a sampled run would incur and assess if this decrease in accuracy is acceptable. Finally, by doing an overlap of the taken metrics, the goal was to find the optimal values for the sampling parameters of the two sampling algorithms used.

4.3 Apple NASDAQ Tweets

The first benchmark application is based on the example provided from the UC Berkeley AMPLab website³. It does an analysis over a data set of Apple NASDAQ tweets⁴. The total size of the data set was 50MB, containing 282.786 tweets. The tweets were taken over a time interval of 79 days, from March 28th, 2016, to June 15th, 2016. The application provides an insight of the top ten language speaking groups that post statuses on Twitter connected to Apple stocks.

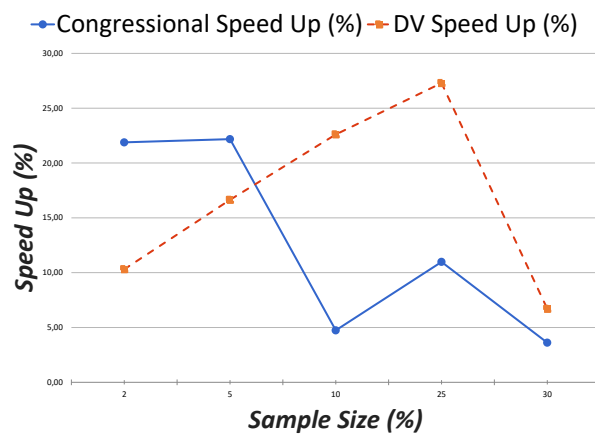
4.3.1 Discussion and analysis

As mentioned before, the Apple NASDAQ tweets data set contains 282.786 tweets, amounting to 50 MB of data. By using the previously mentioned streaming configuration, an average ingestion rate of 1300 data items was received per second. Figure 4.1 depicts the measurements taken for the Congressional, as well as Distinct value sampled executions of the application. The results show the variation in percentage of the sampled over the normal execution.

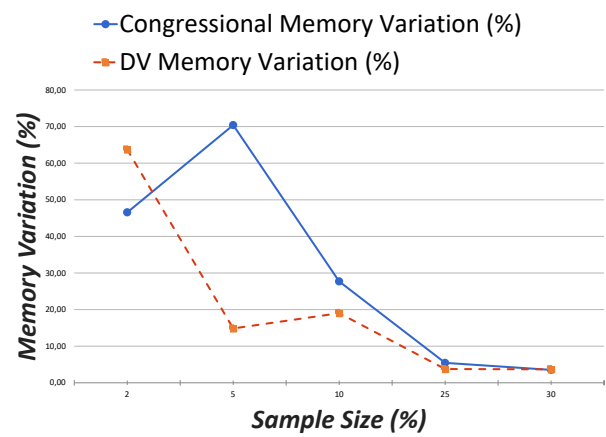
As can be seen in Figure 4.1a, the speed up in total execution time is improved over the normal execution time in both sampling algorithms. The Congressional sampling runs showed that the speed up for this algorithm is inversely proportional to the sample size, providing a 22% speed up for the 5% sample size, a 21% speed up for the 2% sample and a 10% and lower

³"Real-time Processing with Spark Streaming", <http://ampcamp.berkeley.edu/3/exercises/realtime-processing-with-spark-streaming.html>, (June 10, 2016)

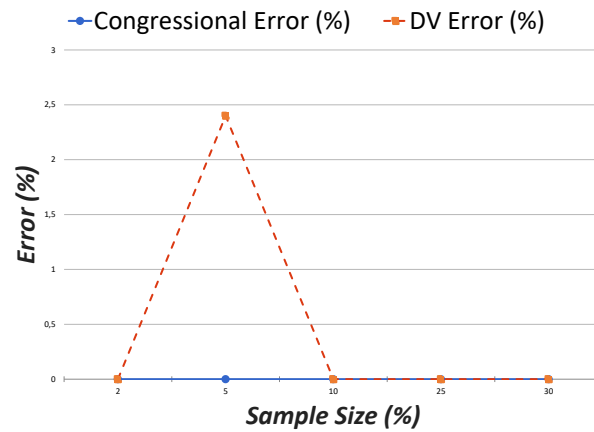
⁴FollowTheHashTag, "One hundred NASDAQ 100 Companies – Free Twitter Datasets", <http://followthehashtag.com/datasets/nasdaq-100-companies-free-twitter-dataset/>, (June 10, 2016)



(a) Tweets Speed Up



(b) Tweets Memory Variation



(c) Tweets Sampling Error

Figure 4.1: Results of data processing for the Apple NASDAQ Tweets: (a) Processing Time Speed Up, (b) Memory Variation, (c) Sampling Error

speed up in processing time for the 25%, 10% and 30% samples. On the other hand, the Distinct Value algorithm executions reported a high speed up in the sample size interval between 5% and 25%, providing a 27% speed up for the 25% sample, 22%, 16% and 10% speed up for the 10%, 5% and 2% samples correspondingly. As the sample size increased, it showed a smaller speed up of 6% for the 30% sample size. In addition, the sampling operations of both algorithms took up less than 1% of the total execution time. The Distinct Value algorithm was faster taking up less than 0.5% of the total execution time for all sampling sizes, while the Congressional algorithm maintained an average of 1% of the total execution time for its sampling operations. From the obtained performance values, it can be seen that the Distinct Value algorithm provides higher processing speed ups than the Congressional algorithm. However, the Congressional algorithm outperforms the DV algorithm for the smaller sample sizes of 2% and 5%.

Shown on the plot in Figure 4.1b is the memory variation of the sampling algorithms over the normal execution of the application. From the plot lines, it can be seen that as the sample size decreases, both algorithms use up more memory than a normal run. The Congressional algorithm uses up the most memory, with an additional 70% more memory for the 5% sample size, while the Distinct Value algorithm utilizes up to 63% additional memory for the 2% sample size. The reason for this is that the sampling operations in the algorithms have to increase to cope with the decreased sample size.

The sampling error can be seen on Figure 4.1c. Both algorithms report a 0% error rate for all of the sample sizes. An exception is the 2% sample size of the Distinct Value algorithm. This is due to the sample size being too small to be efficient for this algorithm, since the sampling method utilizes the threshold value to control how many data items are admitted in the sample and the level value to evict sampled slots from the sample. For such a small sample size, none of the threshold values proved to be efficient and sample slots were evicted at a faster rate, leading to a higher error. Together with the reported values in the plots on Figures 4.1a and 4.1b, it can be concluded that the 2% sample size is not as efficient for the algorithm on this benchmark application.

In conclusion, while both algorithms consume more memory than a normal execution, the Distinct Value evaluates better. The DV algorithm provides higher speed ups than the Congressional algorithm, while providing a 0% error, the same as the Congressional sampling algorithm, with the 2% error being an exception due to the extremely small sample size.

4.4 US Technology Companies Stock

The second benchmark application uses a stock analysis application⁵ as a guide. It does data analysis over a data set of the US stock market⁶. The data set size is 786 MB, numbering 7.103.833 items and contains rows of the stock values of the top forty US technology companies. The earliest data in the data set is from February 7th, 2016 and the latest, at the time of download, was from July 8th, 2016. By analysing this data, the application shows the company that had the largest positive growth in stock value in a given sliding window interval.

4.4.1 Discussion and analysis

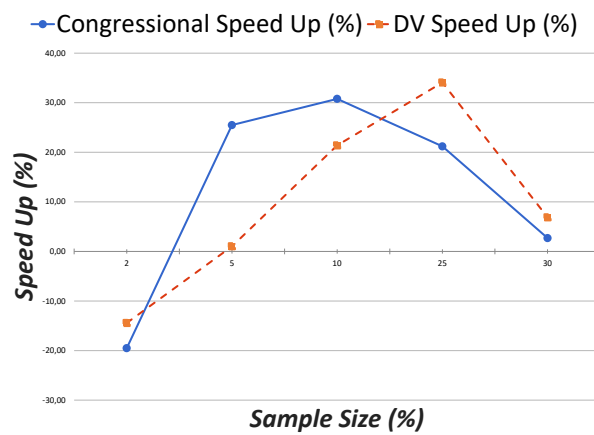
The Stock data set provides an average ingestion rate of 1800 data items per second when streamed over the configuration described in section 4.1. In Figure 4.2 a comparison of the sampled execution metrics can be seen over the normal execution.

Figure 4.2a depicts the execution time speed up of the Stock benchmark application. In both plot lines, it can be seen that the highest speed up is gained between the sample size interval of 5% and 25%. The sampling operations for both algorithms were taking up an average of 1% of the total execution time, with the Distinct Value algorithm being slightly faster. The speed up starts decreasing with sample sizes bigger than 25%. The highest speed up is provided by the Distinct Value algorithm, with a 34% for the 25% sample, and 21% speed up for the 10% sample size. The Congressional sampling algorithm provides a high, 30% speed up for the 10% sample size, with 25% and 21% speed ups for the 5% and 25% samples accordingly. However, the execution time of the sampling algorithms is even slower than the normal execution times for the 2% sample size. This shows that a 2% sample size, for this benchmark application, causes a larger computation overhead with a frequent re-sampling of data items in the reservoir.

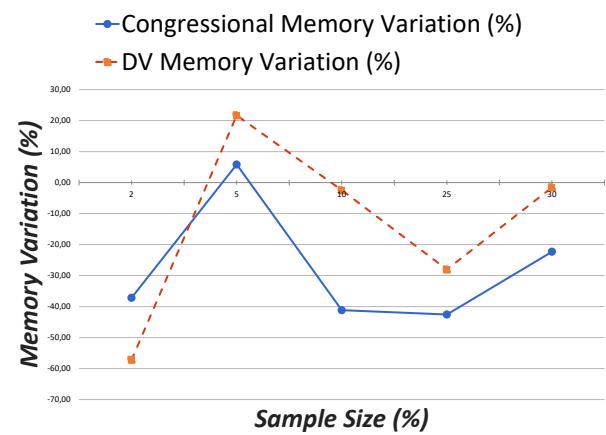
The plot lines on Figure 4.2b show the memory variation of the sampled application runs. As can be seen, both algorithms show that, with sampling, less memory is used, steadily increasing the memory consumption as the sample size is increased. As shown with the plot lines, the Congressional algorithm uses less memory, in general, than the DV algorithm, using 42% and

⁵James Phillpotts, "Real-time Data Analysis Using Spark", <http://blog.scottlogic.com/2013/07/29/spark-stream-analysis.html>, (June 10, 2016)

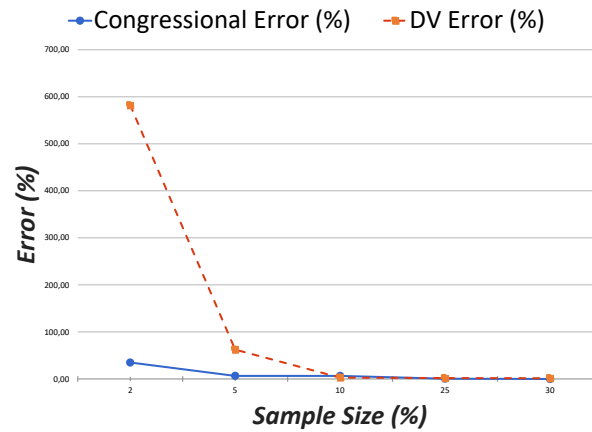
⁶Quandl, "Wiki EOD Stock Prices", <https://www.quandl.com/data/WIKI/documentation/bulk-download>, (June 10, 2016)



(a) Stock Speed Up



(b) Stock Memory Variation



(c) Stock Sampling Error

Figure 4.2: Results of data processing for the US Technology Companies Stock: (a) Processing Time Speed Up, (b) Memory Variation, (c) Sampling Error

41% less memory for the 10% and 25% sample sizes appropriately. For the same sample values, the DV algorithm uses 2% and 28% less memory than the normal execution runs.

Figure 4.2c shows that although the 2% sample size uses the least memory of the experimental runs, it produces results with very large errors. Thus, as with the previous benchmark application, the 2% sample size proves to be inefficient, with 581% error for the DV algorithm and nearly 35% for the Congressional algorithm. The error reports significantly lower values for the larger sample sizes, reporting 6.5% for the 5% and 10% samples, and 0.5% and 0% for the 25% and 30% samples of the Congressional algorithm. The Distinct Value algorithm shows a high error value of 62% for the 5% sample, but decreases the error to 2% for the 10% sample and 1% for the 25% and 30% sample sizes.

From the plots in Figure 4.2 it can be concluded that the algorithms show a more efficient execution for larger data sets. However, better-than-normal execution is gained in the sample size interval of 5% to 25%, with the Congressional sample providing better all-round metrics than the Distinct Value algorithm.

4.5 New York Taxi logs

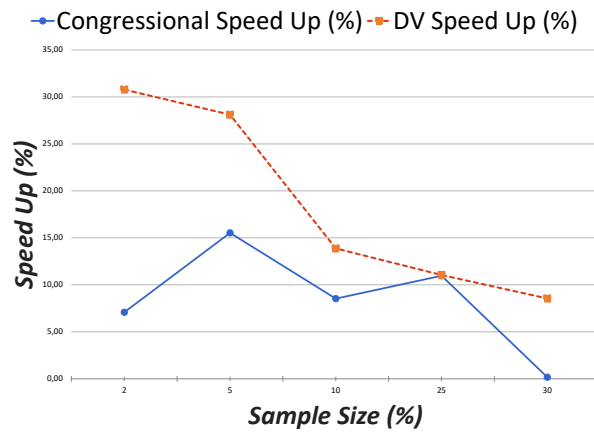
The Databricks-provided example⁷ is used as the basis of the last benchmark application. The purpose of the application is to perform an analysis over a data set of New York taxi logs⁸. It is a 7.388.307 log data set, with a size of 1.7 GB. The data was collected over a period of one year, from January 1st, 2013, to December 31st, 2013. By processing this data set, the application provides the most used payment type in taxi rides in New York city.

4.5.1 Discussion and analysis

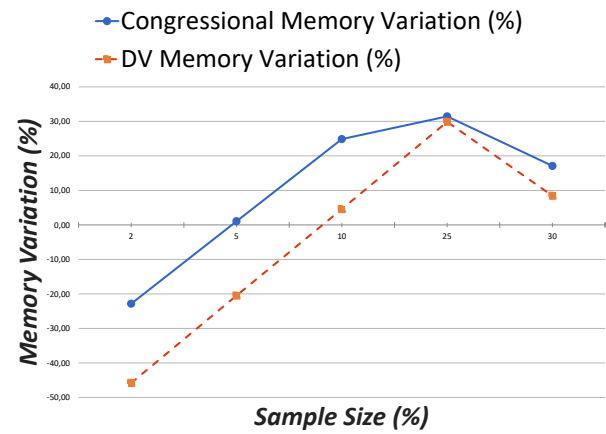
By using the configuration described in section 4.1, the New York City Taxi Log data set provides an average ingestion rate of 1790 items per second. Figure 4.3 plots the measurements taken from the application's normal and sampled executions.

⁷Databricks, "Logs Analyzer Application", https://databricks.gitbooks.io/databricks-spark-reference-applications/content/logs_analyzer/app/index.html, (June 10, 2016)

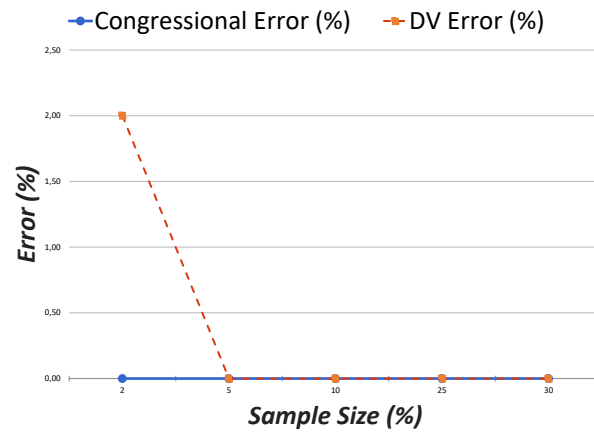
⁸NYC Taxi and Limousine Commission, "NYC Taxi Trip Data 2013 (FOIA/FOIL)", <https://archive.org/details/nycTaxiTripData2013>, (June 10, 2016)



(a) Taxi Log Speed Up



(b) Taxi Log Memory Variation



(c) Taxi Log Sampling Error

Figure 4.3: Results of data processing for the Taxi Logs: (a) Processing Time Speed Up, (b) Memory Variation, (c) Sampling Error

In Figure 4.3a, it can be seen that both algorithms provide an execution time speed up that steadily decreases as the sample size is increased. However, the Distinct Value algorithm provides higher speed up values than the Congressional algorithm, with a 30% speed up for the 2% sample size and 28%, 13% and 11% for the 5%, 10% and 25% samples accordingly. The Congressional algorithm provides the highest, 15% speed up for the 5% sample, and 7%, 8% and 11% for the corresponding 2%, 10% and 25% sample sizes. The sampling operations of the algorithms took up 0.5% of the total execution time for the 2% Distinct Value sample and steadily increasing to a 1% of the total execution time for the 30% sample. For the Congressional algorithm, the fraction of time that was taken up by the sampling operation was larger than the DV algorithm, starting with a low, 0.5% for the 2% sample, but increasing to 9% for the 25% and 30% samples.

Figure 4.3b shows the plots for the memory variation of the algorithms for this benchmark application. While both algorithms show a faster execution time, they use up more memory to achieve this speed up. The Congressional sampling uses the most additional memory, 31%, with the 25% sample, while the Distinct Value algorithm follows closely, with 30% for the same sample size. Additional memory is used with the 10% and 30% samples, where the Congressional algorithm consumes 24% and 17% more memory, and the Distinct Value algorithm consumes 5% and 8% more memory accordingly. An exception is the 5% sample of the DV algorithm, which uses 20% less memory, and the 2% sample size which uses 45% less memory than a normal application run for the Distinct Value sample and 23% less memory for the Congressional sample.

However, this memory efficiency by the 2% Distinct Value sample run costs this execution a 2% lower accuracy compared to the 0% error rates of the rest of the experimental runs of both of the algorithms, as seen on Figure 4.3c.

In conclusion, the Distinct Value algorithm provides better performance in all of the metrics, with higher speed ups, lower additional memory usage and 0% error. However, as with the previous benchmark applications, it shows a low accuracy with very small sample sizes.

4.6 Online Retailer

The last benchmark application processes the data of a UK-based online retailer (Chen et al., 2012). As the previous benchmark, it uses the Databricks example⁹ as a guide. The data set size is 30 MB, numbering 541.909 items and contains all the transactions that occurred between December 1st, 2010 and December 9th, 2011. The output of the application shows the top ten countries that have the most customers on the retailer’s web site.

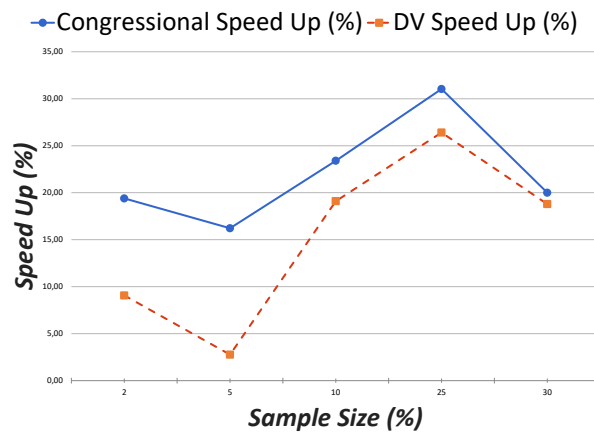
4.6.1 Discussion and analysis

The Online Retailer benchmark application receives an average ingestion rate of 1770 data items per second with the provided experimental configuration. During the sampled executions of the application the measurements seen in Figure 4.4 were taken.

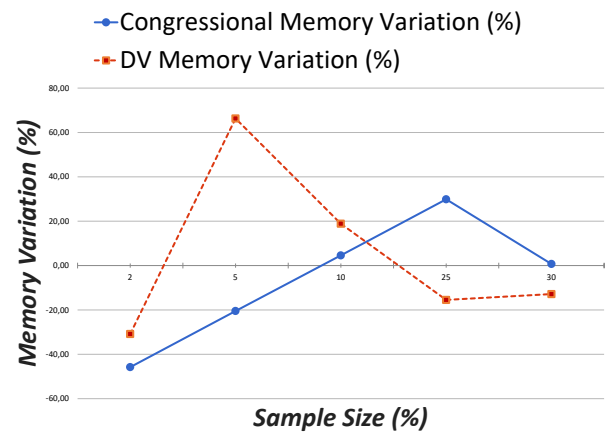
Figure 4.4a shows the execution time speed up of the sampling algorithms. Both, the Congressional and Distinct Value algorithms show considerably better execution times in the sampling size interval between 10% and 30%. The Congressional algorithm shows better performance with 31%, 23% and 20% speed up for the corresponding 25%, 10% and 30% samples, while the Distinct Value algorithm reports a 26%, 19% and 18% speed up for the appropriate values. Speed up is present, but decreasing outside of this interval, with the Congressional sampling algorithm continuously showing a 16% and 19% speed up for the 5% and 2% samples, while the Distinct value algorithm shows a lower, 2% and 9% speed up for these sample values. For both algorithms, the time that was occupied by the sampling operations averaged 2.4% of the total execution time for all of the sample sizes.

However, Figure 4.4b shows that the Congressional algorithm achieves the gain in speed up at the cost of higher memory usage in the previously mentioned interval. It uses up 29% more memory for the 25% sample size, 4% more memory for the 10% sample, and 1% additional memory for the 30% sample. On the other hand, the sensitivity to lower sample sizes of the Distinct Value algorithm depicts a higher memory usage for the lower sample sizes, with a maximum additional memory usage of 66% for the 5% sample size, and a more efficient memory usage than the normal execution for the bigger sample sizes, showing a 15% memory decrease

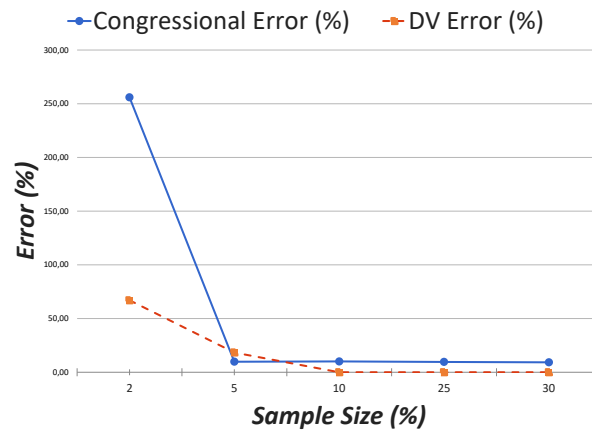
⁹Databricks, "Logs Analyzer Application", https://databricks.gitbooks.io/databricks-spark-reference-applications/content/logs_analyzer/app/index.html, (June 10, 2016)



(a) Retail Speed Up



(b) Retail Memory Variation



(c) Retail Sampling Error

Figure 4.4: Results of data processing for the Online Retailer Transactions: (a) Processing Time Speed Up, (b) Memory Variation, (c) Sampling Error

for the 25% sample. The reason for this performance of the DV algorithm is the nature of the algorithm, which causes a more frequent level increase for smaller sample sizes, thus evicting sample slots at a faster rate. However, the Congressional algorithm uses less additional memory than the DV algorithm.

The error rate for the sampled executions is shown on Figure 4.4c. The plots show that the algorithms provide a low error for most sample sizes. As with the previous benchmarks, the results of the 2% sample size shows that very small sample sizes are not suitable and cause higher error rates of 255% for the Congressional algorithm, and 67% for the Distinct Value algorithm. The Congressional sample lowers this error to 9% for the larger sample sizes, while the DV algorithm provides 18% error for the 5% sample, and 0% for the larger sample sizes. As a result, the Distinct Value algorithm provides better accuracy for this benchmark application.

When summarized, these results show that for this benchmark application, the Congressional sampling algorithm provides better performance than the Distinct Value algorithm. It provides higher speed ups, with lower additional memory usage and low error.

Summary

In this chapter, the experimental evaluation made over the solution presented in this work was detailed and the gathered results were discussed. Tables 4.1 and 4.2 show a summary of the collected results of the metrics for both algorithms.

From Table 4.1, it can be seen that the Congressional sampling algorithm provides significant speed ups in execution time for the 5%, 10% and 25% sample sizes for all of the benchmark applications and positive values in speed up for the 2% sample. In addition, the error rate at most of the sample runs is decidedly low. With two exceptions, the maximum error mounts to 10%, but generally shows a 0% error. However, in most of the benchmark applications, this efficiency in execution time and low error is gained at the cost of higher memory usage. By using the results, it can be concluded that the Congressional sampling algorithm can be used for most sample sizes. While the 2% and 5% sample sizes show a higher error rate, they continue to provide a high processing speed up, but additionally incur less additional memory usage than the higher sample sizes.

The Distinct Value algorithm results shown in Table 4.2 depict a different picture. Speed

ups in execution time can be noticed in the 10% and 25% sample sizes, additionally reporting a low error rate. The 2% and 5% samples, show a mixed view of the speed up data, providing several high speed ups in processing time, but also some very low, and negative values. These two sampling sizes provide a better performance in memory usage, with decreases in memory consumption of up to 57%. However, this decrease in memory is done at the cost of accuracy, providing error rates as high as 581%. On the other hand, the 10% and 25% sample sizes have provided good, or low additional, memory usage. Combined with the all-round better performance in processing time and error rates, it can be seen that the Distinct Value algorithm performs better for larger sample sizes and the performance improves when using large data sets. In the case of streamed data, getting higher ingestion rates would improve the performance of this algorithm.

When combining the data from both tables, it can be concluded that when smaller sample sizes are needed, it is better to use the Congressional sampling algorithm. It provides a high processing time speed up of up to 25% for the 2% and 5% sample sizes. In addition, it provides up to 45% lower memory consumption than a normal run, and low error values. As the sample size increases, Congressional sampling can still be used, but it starts to provide lower speed ups in processing time, with an average of 13% speed up between the sample sizes of 10% to 30%. In these cases, it is better to use the Distinct Value sampling algorithm which provides a 18% average for the before mentioned interval. In addition, in this interval, the Distinct Value algorithm provides less additional memory consumption than the Congressional algorithm and, furthermore, shows a lower error rate, averaging to 2% error. Both algorithms showed that the sampling operations that they perform, take up a small, less than 3%, fraction of the total execution time, with Congressional sampling being slightly slower due to the larger complexity of the algorithm.

The following chapter completes the thesis by presenting the conclusions gained during the development of this solution and its evaluation. Furthermore, additional discussion is provided for future work on the solution.

Table 4.1: Congressional Algorithm Results Summary

Congressional Algorithm				
Sample Size (%)	Stock Data	Twitter Data	Log Data	Retail Data
		Speed Up (%)		
2	-19.52	21.88	7.08	19.39
5	25.50	22.18	15.53	16.22
10	30.78	4.74	8.53	23.40
25	21.20	10.98	10.97	31.04
30	2.70	3.61	0.18	20.00
Sample Size (%)		Memory Variation (%)		
2	-37.17	46.57	-22.84	-45.79
5	5.84	70.41	1.06	-20.48
10	-41.16	27.70	24.86	4.59
25	-42.56	5.44	31.45	29.87
30	-22.32	3.53	17.09	0.72
Sample Size (%)		Error (%)		
2	34.97	0.00	0.00	255.92
5	6.55	0.00	0.00	9.81
10	6.55	0.00	0.00	10.05
25	0.57	0.00	0.00	9.60
30	0.00	0.00	0.00	9.24

Table 4.2: Distinct Value Algorithm Results Summary

Distinct Value Algorithm				
Sample Size (%)	Stock Data	Twitter Data	Log Data	Retail Data
		Speed Up (%)		
2	-14.43	10.31	30.78	9.07
5	1.02	16.64	28.11	2.77
10	21.40	22.61	13.88	19.10
25	34.08	27.29	11.05	26.40
30	6.88	6.70	8.55	18.79
Sample Size (%)		Memory Variation (%)		
2	-57.21	63.72	-45.79	-30.86
5	21.69	14.85	-20.48	66.30
10	-2.45	19.03	4.59	18.86
25	-28.05	3.76	29.87	-15.49
30	-1.56	3.68	8.43	-12.86
Sample Size (%)		Error (%)		
2	581.76	0	2.00	67.018
5	62.33	2.4	0.00	18.3836
10	2.87	0	0.00	0
25	1.82	0	0.00	0
30	1.50	0	0	0

5

Conclusions

5.1 Conclusions

The system implements the approximate computing paradigm by leveraging the advantages of sampling as a data reduction technique. It utilizes the modularity of the Apache Spark framework to create a seamless merging of this established data processing platform with the sampling framework provided by this solution.

The system provided a user-transparent sampling framework that provides two features. First, it enables the rapid development of one-pass sampling algorithms in Apache Spark. Second, it provides a sampling module for the usage of the sampling algorithms in stream processing applications. Finally, two advanced sampling techniques were incorporated in Spark by using the sampling framework. The Congressional and Distinct Value sampling methods cover a wide area of use-cases in the field of data processing.

The system showed that current data processing systems can still benefit from advancements made before the Big Data Revolution. The experimental results indicate that the system can be employed in heavy data stream environments and provide up to 34% faster execution time, while maintaining a low error bound and limited memory overhead.

5.2 Future Work

Although the system is fully functional for stable data streams, the introduction of a variable arrival rate in the data stream may impact the accuracy of the results. This is because the sample size would maintain a fixed value while the amount of data fluctuates.

Future work may address the implementation of a self-adjusting sample size depending on the error measurement and processing time. This may be further expanded by recording the results of prior executions to remember the best parameters of a sampling algorithm and adjust

these parameters for each future job. Additionally, allowing the definition of QoS thresholds for error and accuracy overheads would gain the best resource usage for the best available speed up. Finally, a module to detect resource usage and shift the execution of an application from normal to sampled mode would provide the optimal performance and resource utilization.

Bibliography

- Acharya, S., P. B. Gibbons, and V. Poosala (2000). Congressional samples for approximate answering of group-by queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, New York, NY, USA, pp. 487–498. ACM.
- Agarwal, S., B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica (2013). Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, New York, NY, USA, pp. 29–42. ACM.
- Ahmad, Y., B. Berg, U. Cetintemel, M. Humphrey, J.-H. Hwang, A. Jhingran, A. Maskey, O. Papaemmanouil, A. Rasin, N. Tatbul, et al. (2005). Distributed operation in the borealis stream processing engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pp. 882–884. ACM.
- Akidau, T., R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle (2015, August). The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.* 8(12), 1792–1803.
- Babcock, B., M. Datar, R. Motwani, et al. Load shedding techniques for data stream systems. Citeseer.
- Carbone, P., A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas (2015). Apache flinkTM: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*.
- Carney, D., U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik (2002). Monitoring streams: A new class of data management applications. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pp. 215–226. VLDB Endowment.

- Chaudhuri, S., G. Das, M. Datar, R. Motwani, and V. Narasayya (2001). Overcoming limitations of sampling for aggregation queries. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pp. 534–542. IEEE.
- Chen, D., S. L. Sain, and K. Guo (2012). Data mining for the online retail industry: A case study of rfm model-based customer segmentation using data mining. In *Journal of Database Marketing and Customer Strategy Management - Volume 19, No. 3*, pp. 197–208. 10. Accessed: 2016-08-10.
- Cormode, G. and N. Duffield (2014). Sampling for big data: A tutorial. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14*, New York, NY, USA, pp. 1975–1975. ACM.
- Das, T., Y. Zhong, I. Stoica, and S. Shenker (2014). Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing, SOCC '14*, New York, NY, USA, pp. 16:1–16:13. ACM.
- Duffield, N. (2016). Sampling and interference problems for big data in the internet and beyond. Rutgers University - DIMACS.
- Engle, C., A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica (2012). Shark: Fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12*, New York, NY, USA, pp. 689–692. ACM.
- Foundation, A. S. (2016). Apache hadoop yarn. <https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/YARN.html>. Accessed: 2016-08-07.
- Garg, N. (2013). *Apache Kafka*. Packt Publishing.
- Gibbons, P. B. (2001). Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proceedings of the 27th International Conference on Very Large Data Bases, VLDB '01*, San Francisco, CA, USA, pp. 541–550. Morgan Kaufmann Publishers Inc.
- Gibbons, P. B. and Y. Matias (1998). New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, SIGMOD '98*, New York, NY, USA, pp. 331–342. ACM.

- Goiri, I., R. Bianchini, S. Nagarakatte, and T. D. Nguyen (2015). Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, New York, NY, USA, pp. 383–397. ACM.
- Hu, W. and B. Zhang (2012). Study of sampling techniques and algorithms in data stream environments. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2012 9th International Conference on*, pp. 1028–1034. IEEE.
- Krishnan, D. R., D. L. Quoc, P. Bhatotia, C. Fetzer, and R. Rodrigues (2016). Incapprox: A data analytics system for incremental approximate computing. In *Proceedings of the 25th International Conference on World Wide Web, WWW '16*, Republic and Canton of Geneva, Switzerland, pp. 1133–1144. International World Wide Web Conferences Steering Committee.
- Pettey, C. and L. Goasduff (2011, June). Gartner says solving 'big data' challenge involves more than just managing volumes of data. <http://www.gartner.com/newsroom/id/1731916>. Accessed: 2016-08-07.
- Sun, L., M. J. Franklin, S. Krishnan, and R. S. Xin (2014). Fine-grained partitioning for aggressive data skipping. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14*, New York, NY, USA, pp. 1115–1126. ACM.
- Tatbul, N., U. Çetintemel, and S. Zdonik (2007). Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB '07*, pp. 159–170. VLDB Endowment.
- Tatbul, N., U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker (2003). Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29, VLDB '03*, pp. 309–320. VLDB Endowment.
- Tatbul, N. and S. Zdonik (2006). Window-aware load shedding for aggregation queries over data streams. In *Proceedings of the 32Nd International Conference on Very Large Data Bases, VLDB '06*, pp. 799–810. VLDB Endowment.
- Thusoo, A., J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy (2009, August). Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.* 2(2), 1626–1629.

- Toshniwal, A., S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. (2014). Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 147–156. ACM.
- Vitter, J. S. (1985, March). Random sampling with a reservoir. *ACM Trans. Math. Softw.* 11(1), 37–57.
- White, T. (2009). *Hadoop: The Definitive Guide* (1st ed.). O’Reilly Media, Inc.
- Zaharia, M., M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica (2010). Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud’10, Berkeley, CA, USA, pp. 10–10. USENIX Association.
- Zaharia, M., T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica (2013). Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP ’13, New York, NY, USA, pp. 423–438. ACM.