Enabling Arbitrary Memory Constraint Standing Queries on Distributed Stream Processors using Approximate Algorithms

TOBIAS LINDENER
Enabling Arbitrary Memory Constraint Standing Queries on Distributed Stream Processors using Approximate Algorithms

TOBIAS SIMEON LINDENER
Abstract

Relational algebra and SQL have been a standard in declarative analytics for decades. Yet, at web-scale, even simple analytics queries can prove challenging within Distributed Stream Processing environments. Two examples of such queries are "count" and "count distinct". Since aforementioned queries require persistence of all keys (the value identifying an element), such queries would result in continuously increasing memory demand. Through approximation techniques with fixed-size memory layouts, said tasks are feasible and potentially more resource efficient within streaming systems. Within this thesis, (1) the advantages of approximate queries on distributed stream processing are demonstrated. Furthermore, (2) the resource efficiency as well as (3) challenges of approximation techniques, and (4) dataset dependent optimizations are presented. The prototype is implemented using the Yahoo Data Sketch library on Apache Flink. Based on the evaluation results and the experiences with the prototype, potential improvements like deeper integration into the streaming framework are presented. Throughout the analysis, the combination of approximate algorithms and distributed stream processing shows promising results depending on the dataset and the required accuracy.
Sammanfattning

Relationsalgebra och SQL har varit standard inom analys i decennier. Inom distribuerade strömmande datamiljöer på web-nivå kan dock även enklare analytiska frågor visa sig utmanande. Två exempel är frågorna ”count” och ”count distinct”. Eftersom de nämnda frågorna kräver att alla nycklar (de värden som identifierar ett element) är persistenta så resulterar detta traditionellt i en kontinuerlig ökning av minneskraven. Genom uppskattningsmetoder med bestämd storlek av minnes-layouten blir de ovan nämnda frågorna rimliga och potentiellt mer resurseffektiva inom strömmande system. I detta forskningsarbete demonstreras (1) fördelarna samt gränserna för approximativa frågor inom distribuerade strömmande processer. Vidare presenteras (2) resurseffektivitet samt (3) svårigheter med uppskattningsmetoder. (4) Optimeringar med avseende på olika dataset redovisas. Prototypen är implementerad med Yahoo Data Sketch biblioteket på Apache Flink. Möjliga förbättringar som djupare integration inom strömmande ramverk, baserat på evalueringsresultaten samt erfarenheter med prototypen presenteras. I analysen visas att en kombination av approximativa algoritmer och distribuerade strömmande processer resulterar i lovande resultat, beroende på dataset samt begärd noggrannhet.
Contents

1 Introduction ......................................................... 1
   1.1 Background ................................................. 2
   1.2 Problem ................................................... 3
   1.3 Research Question ......................................... 3
   1.4 Purpose .................................................... 4
   1.5 Goal ........................................................ 4
   1.6 Benefits, Ethics and Sustainability .......................... 4
   1.7 Methodology ............................................... 5
   1.8 Delimitations .............................................. 6
   1.9 Contributions .............................................. 6
   1.10 Outline ..................................................... 6

2 From Batch Processing to Real-Time Information Processing .......... 7
   2.1 Distributed Data Processing ................................ 7
   2.2 Stream Processing and Distributed Stream Processing ........... 8
      2.2.1 Processing Operators .................................. 9
      2.2.2 Queries on Streams .................................... 10
      2.2.3 Structured Query Language ............................ 11
      2.2.4 Continuous Query Languages ......................... 11
   2.3 Approximate Stream Analysis ................................ 11
      2.3.1 Approximate Identification of Frequent Items .......... 12
      2.3.2 Entropy Estimation of Streams ....................... 12
      2.3.3 Cardinality Estimation ................................ 13
   2.4 Distributed Stream Processing Frameworks ....................... 15
      2.4.1 Apache Flink ......................................... 15
   2.5 Related Projects and Frameworks ............................ 16

3 Design and Development of an Approximate Query Library for Dis-
   tributed Stream Processing .................................. 18
   3.1 Design Requirements ....................................... 19
   3.2 API Specification ........................................... 20
   3.3 Implementation Requirements ............................... 21
      3.3.1 Flink Data Types .................................... 22
      3.3.2 Flink Operators ..................................... 22
## List of Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CEP</td>
<td>Complex Event Processing</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CQL</td>
<td>Continuous Query Language</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>DSMS</td>
<td>Data Stream Management System</td>
</tr>
<tr>
<td>DSP</td>
<td>Distributed Stream Processing</td>
</tr>
<tr>
<td>FI</td>
<td>Frequent Item</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose Graphics Processing Unit</td>
</tr>
<tr>
<td>HLL</td>
<td>HyperLogLog</td>
</tr>
<tr>
<td>HPC</td>
<td>High Performance Computing</td>
</tr>
<tr>
<td>MG</td>
<td>Misra-Gries</td>
</tr>
<tr>
<td>ML</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relational Database Management System</td>
</tr>
<tr>
<td>SMT</td>
<td>Simultaneous Multi-Threading</td>
</tr>
<tr>
<td>SPS</td>
<td>Stream Processing Systems</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>SS</td>
<td>Space-saving</td>
</tr>
<tr>
<td>SSD</td>
<td>Solid State Disk</td>
</tr>
</tbody>
</table>
1 Introduction

In recent history, the term “data is the new oil” (Clive Humby, 2006) has been cited continuously. The focus on data processing, information extraction and machine learning is increasing both within businesses and academia. Over time, various approaches to data processing have been proposed. Starting from batch processing, over data warehouses to predictive and most recently cognitive computing. All those computational models share a common property, they assume data to be bounded. Businesses create quarterly reports of their finances and crime is summarized in yearly statistics. Both examples are processed in batches, sometimes long after they happened. In reality, data flow constantly, therefore, we could potentially extract meaningful information constantly. Why create a quarterly report, if we could have real-time information about the current financial situation of a company? Why wait for the data warehouse to process the latest business transactions to update predictive models?

There is huge potential in the capability of constant and near real-time processing of data. So far, this potential is only partially embraced, since few companies have the capability to redesign their applications and services around the concept of continuous stream processing. Nonetheless, especially since the development of Spark Streaming and Apache Flink, the approach of rethinking the flow of information has increased. Yet, the increasing amount of data does not only put a strain on batch systems but also on stream processing systems. At web-scale, even comparably simple analytical queries become computational demanding beyond what current systems can cope with. For example, frequent items, quantile calculation or count distinct queries in stream processing can generate a continuously expanding memory demand to store the respective key identifying the element. This is where approximate computing comes into play. Depending on the use case and in the context of large-scale data, exact answers are not necessarily needed. In such cases, approximate computing can provide answers to analytical queries which are challenging to compute on stream processors.

This work proposes the use of approximate algorithms on distributed stream processors to enable memory-intensive standing queries on unbounded data, and improve the efficiency of these queries. The work includes a prototype implementation of frequent item identification, count distinct queries, and quantile calculation. Within the evaluation (1) the advantage in performance is identified and presented, (2) the limitations in terms of accuracy are analyzed, and (3)
an overview of efficient default parameters for the respective algorithms is given.

1.1 Background

The difference between bounded and unbounded data is the foundation of stream processing. Bounded data is limited through an inherent concept. A potential example is the time, where data is bounded by the concept of hours, days or months. Alternatively, data can be bounded by other concepts like user sessions. Within batch processing systems these conceptual boundaries have been used to batch the processing and make it feasible to process. Often this comes down to the easiness of implementation rather than inherent usefulness. For example monthly or quarterly aggregation of sales data, there is no obvious value in only providing such aggregations bounded by time. The stream processing approach is a strategy to rethink data processing and the respective application design. Instead of artificially bounding data, the application continuously provides the most recent calculations for a given objective. This can result in potentially inaccurate intermediate results, which will finally become accurate. An example is a streaming average over a sequence of numbers. Given the impact of large outliers when calculating the arithmetic mean, a sequence of 2, 3, 4, 2, 10 would yield low averages in the beginning and generate a spike after the “last” element is processed. While in batch processing systems, all events of the sequence are gathered first and finally processed to create the result. Figure 1.1 visualizes this scenario.

Figure 1.1: Impact of outliers in continuous average
This example also highlights, why stream processing and approximate computing fit quite well together. The thought of “imprecise” intermediate results from processing continuous data streams, is already a characteristic in the application design. The reason to move from batch processing to stream processing is more often indicated due to the massive amount of data to process. In many cases, it is impossible to store data infinitely and often enough it is also not relevant. With the increasing amount of Internet of Things applications where large sensor networks create a massive amount of data this problem increases [1].

1.2 Problem

Web-scale data processing creates a variety of challenges related to data processing and data analytics. As mentioned before, some of them are solved using stream processing instead of batch processing. Yet, stream processing has its own unique challenges. Limited memory in the case of queries over unbounded data is one such challenge [2]. As a result, comparably simple tasks like continuously counting items or distinct counts become challenging with potentially infinitely growing datasets. The number of keys for example in the case of distinct counting is continuously growing in most web-scale analytics scenarios. While such a scenario is problematic when doing exact processing, it becomes solvable considering approximate algorithms. With these algorithms, the memory constraint can be circumvented if estimated results are acceptable. Whether approximate values are usable is dependent on the use-cases requirement and the respective error.

1.3 Research Question

The project at hand tries to create an understanding of approximate algorithms in the realms of distributed stream processing. This leads to our main research question:

Can we create accurate and efficient memory constraint standing queries for unbounded data on distributed stream processing while distributing data over multiple instances of approximate algorithms?

The evaluation focuses on the domains of e-commerce/advertising for frequent item queries and the telecommunications industry for distinct count queries. In both cases the accuracy of approximations is generally sufficient for realistic use-cases [3, 4].
The following sub-questions will guide the analysis throughout this work and will be a point of reference within the evaluation:

1. Under which circumstances do approximate algorithms work on distributed streaming environments?

2. What is the impact of parallel computing and the impact of the data distribution of datasets on the accuracy of the approximations?

3. How can the choice of algorithm parameters positively influence the approximation accuracy and what are the lower limits?

A detailed explanation of the research methodology and its advantages and disadvantages will be given within the dedicated methodology section.

1.4 Purpose

The thesis discusses the potential of using approximate computing techniques within large-scale distributed stream processing environments. This enables users to handle analytical tasks on a larger scale and illustrates the challenges of thinking in intermediate instead of exact results. Throughout this work, the challenges and advantages will be identified and analyzed.

1.5 Goal

The goal is to identify the advantages and limitations of approximation techniques on distributed stream processors. This includes the development of a prototype and respective evaluation of its performance.

1.6 Benefits, Ethics and Sustainability

The results of the degree project will mainly benefit the BigData community in general. This can include researchers as well as companies. Due to the nature of data analytics, an inherent risk of misuse of such technologies is always existent. The recent case of Cambridge Analytica has highlighted the problem of analytics and privacy [5]. Yet, the technology presented in this work is not more ethically relevant than other data processing techniques. From a sustainability perspective, the project can help make web-scale data processing more efficient.
by reducing computational resource demand. This, in turn, can reduce the overall use of resources. But it might also make certain analysis possible for the first time and therefore increase the overall amount of data analytics performed.

1.7 Methodology

The thesis builds on top of empirical and experimental methods to extract knowledge in the given area. The design is developed using qualitative criteria following industry-oriented best practices while the prototype is evaluated using an experiment-oriented research strategy. It is compared to a baseline using exact computation to verify claims with regards to accuracy, scalability and error bounds of approximate computing.

In approximate computing a common hypothesis is the relevance of the specific data and their respective use-case with regards to the required accuracy. For this reason, we employed an empirical research approach to provide first insights on a subset of publicly available data while also offering the means to reproduce said evaluation on other datasets. In recent work on use of approximate algorithms in batch processing, it is argued that "perfect Decisions are Possible even with Imperfect Answers" [6]. Yet, this needs to be evaluated on a case by case basis. In contrast, the potential of distributed execution of approximate algorithms could also be analyzed using a theoretical approach. Given the previously mentioned dependency on the use-cases, a theoretical analysis would mainly provide an estimate whether approximation is viable. This might not be sufficient when making a decision on the use of approximation outside of academia.

Alternative approaches could focus more on (1) the theory behind the underlying algorithms or (2) analyzing the dependency between algorithms and data. Scenario one, would require deep theoretical knowledge in the sphere of approximate algorithms. The analysis could identify potential improvements for existing algorithms when parallelizing execution. Such an analysis could involve learnings from other algorithm implementations (e.g. MapReduce based implementations). Scenario two, could present information related to the impact of varying data characteristics. Similarly, this relates less to the application of approximate algorithms but rather to the general research in this area. [7]

Finally, with the empirical research method in mind, the data collection was implemented using an automated approach. This enables us to run various tests with equal environment factors. It also reduces the risk of human error during the execution. The observations are analyzed both visually using exploratory data analysis, as well as statistical analysis on the original datasets as well as the
error bounds. The research method was instantiated based on typically cited use cases (refer to chapter 2.3), this also guided the choice of datasets. The selection of algorithms was influenced by the availability of existing high performance implementations of approximate algorithms. Since the implementation is not the focus of this project, we decided to build the work on top of an existing implementation which can handle large datasets.

1.8 Delimitations

The prototype is implemented using existing algorithms which are not specifically designed for distributed systems. It is to be expected that work within that area could positively influence the results of this work.

1.9 Contributions

The work at hand (1) demonstrates the use of approximation techniques to enable the use of - normally - memory-extensive standing queries on DSP, (2) it analyzes the effect of different data distributions, (3) based on this analysis favorable parameters for the approximation algorithms are identified. Finally, the thesis provides insights into (4) the performance and scaling characteristics of approximate algorithms on DSP compared to exact implementations. This analysis is based on an openly available (5) software framework for standing queries on Apache Flink.

1.10 Outline

In the following chapter, the theoretical background and necessary vocabulary are presented. This includes stream processing specific terms and aspects, as well as the fundamental knowledge of approximate computing. Chapter 3 on page 18 will introduce the research carried out and present the technical design. Within chapter 3.3 on page 21 the prototype implementation is demonstrated. Chapter 4 will demonstrate the evaluation performed on the prototype. This will also refer to the design guidelines outlined in chapter three. In chapter 5 on page 48 the results are critically discussed. Finally, chapter 6 on page 51 concludes the thesis.
2 From Batch Processing to Real-Time Information Processing

Traditionally, large-scale data processing is implemented following the batch processing model [8]. This means data are first created or gathered, then stored, and finally processed by a system. Data-warehouses (in the case of the ETL process) follow the same pattern of batched extraction, transformation and loading [9]. Many optimizations have increased the performance of such systems to accommodate the growing amount of data [10]. One optimization to handle the ever-increasing amount of data, is the MapReduce programming model developed by Google [11]. MapReduce is a method to handle large quantities of data while increasing cost efficiency. The idea follows the "divide and conquer"[12] paradigm where operations over data are split and intermediate results are aggregated. A key design aspect to increase cost efficiency is the use of commodity hardware (x86 hardware) instead of specialized systems. Effectively, the MapReduce algorithm still follows the same batch approach which requires all data to be stored initially. The latest development to handle large-scale data beyond batch processing is the use of stream processing. In the following chapter, the relevant characteristics for both batch and stream processing are presented. Furthermore, the advantages and limiting factors are explained.

2.1 Distributed Data Processing

A key solution to handle large-scale data is parallel or distributed processing. Parallel processing is an integral part of modern computation. Modern CPU use multi-core and vector extensions for extensive parallel processing. GPGPU processing goes even further to parallelize computation. Distributed processing goes beyond parallelization within CPU or GPU and parallelizes with multiple machines. Within parallel processing a distinction is made between parallelization by task and parallelization by data. Additionally, mixed models or hybrid models also exist. Task parallelization is a concept where different independent operations are executed over the same data. In contrast, parallelization by data executes the same operation over various (parts) of data. Within large-scale and distributed data processing, the parallelization by data is the predominant
model. In the HPC environment, Message Passing Interface is the main processing paradigm. Since Goggle’s development of MapReduce algorithm, MPI has become less popular for data-intensive tasks [13]. MapReduce works in a data parallel way, the “users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key” [14]. It builds on top of knowledge from functional programming and enables a comparably simple approach to parallelization. MapReduce provides fault-tolerance through re-execution and therefore allows the use of commodity hardware. MapReduce started to gain increased traction through the development of the Hadoop Ecosystem, an open source implementation of the map reduce programming model by Yahoo. More recently, the Spark project has additionally increased the performance of such systems through in-memory processing, sharing data dependencies through an application lineage graph and persistence of intermediate results [15, 16, 17].

2.2 Stream Processing and Distributed Stream Processing

Stream processing itself is an old discipline dating back to the beginning of computer science, file systems for example are traditionally accessed through streams [18]. This means the data is not fully loaded into memory but rather read sequentially and only partially kept within the application. Stream processing provides the ability to handle data sets which go beyond the boundaries of the main memory. This approach is specifically relevant in the ongoing trend of the Internet of Things. For example, large sensor networks can produce a constant stream of data which is not necessarily of relevance to persist on disk (e.g. no significant change of a temperature value). Within stream processing, different event-based processing architectures exist, for example Complex Event Processing (CEP), Stream Processing, or Actor Pattern. Each of those pattern stem from different backgrounds but capture a similar idea with varying size of data [19]. Comparably new is the concept of Distributed Stream Processing, basically scaling Stream Processing similarly to Batch Processing with the goal to increase performance through parallelization compared to non-distributed event processing [20].
2.2.1 Processing Operators

Similarly, to Batch Processing there are primitives with which to interact with Data Streams and the elements within that stream. Operators transform the elements within a stream. These transformations are expressed using operators like map, flatmap, filter or aggregations (e.g. sum, min or max). Depending on the implementation, user-defined functions (UDF) or higher level functions similarly to SQL can also be available (e.g. group-by).

Windows are a concept used to group elements within a stream. The selection criteria to group the elements within a window varies based on the application. In the following paragraphs, some examples of typical windows with their respective selection criteria are presented. The summary is based on the survey of Kleppmann and the work of Chandrasekaran et al. as well as Gehrke, Korn, and Srivastava. The least complex type of windows is the **fixed-size** or **tumbling window**. Elements are partitioned into fixed-length and independent chunks (no overlaps). The Stream Elements can only belong to one window.

**Sliding windows** also have a fixed-size but they slide over the stream based on an interval. The interval is usually defined by time and it can be smaller than the window size. As a result, the elements can belong to multiple windows depending on the window size and interval length.

**Session windows** are a complex type of windows which are usually application specific. Sessions are defined by some type of identification comparably to the session id generated when visiting websites. Figure 2.1 on the next page highlights the difference to the previous window types. First, sessions between users can overlap and, secondly their size can vary per session indicator and between each session.

Within stream mining, there also exists the concept of decaying windows. A sliding window is the most intuitive form of time-decaying windows, a more advanced version is the exponentially decaying window. Often, stream elements are kept for a longer time with an exponentially decaying weight used in the overall calculation. Figure 2.2 on the following page highlights the difference between a fixed sized window, where the weight stays constant for all weights of the most recent $1/c$ elements. [23]
2.2.2 Queries on Streams

Queries are separated into two categories, **ad-hoc queries**, and **standing queries**.

Ad-hoc queries behave similarly to the way traditional database management systems are used. Given a use case, the historic data is queried to find an answer to the use case at hand. The query is asked once about the current state of the data (in this case the stream). A problem compared to DBMS is the fact that the query will only provide the result based on all persisted elements of the stream. Depending on the persistence-criterion, not all historical elements are available at the time of the query. A common solution to this problem is the use of ad-hoc queries solely on windowed data, or alternatively to use sampling mechanisms over the whole stream.  

A **standing query** is a different construct and could be compared with the trigger in DBMS. The query is continuously processed over all incoming data. A
straightforward example for standing queries are alerting mechanisms, where for example a sensor value (like a temperature) exceeds a threshold and therefore a warning is sent out. Alternatively, a standing query can be used to identify the continuous maximum value seen within a stream or various other use-cases where the knowledge about the full history needs to be captured (which does not necessarily require complete persistence of the stream). [23]

2.2.3 Structured Query Language

SQL was developed as an intuitive interface to query tables. It builds upon the idea of separating internal representation of data in databases from the external interface to the user. SQL contains a set of simple operations and more advanced operations to interact with tabular data structures. Since its foundation SQL has been continuously improved and extended with various features. Nowadays, it is primarily used for static data or in RDBMS. [24]

2.2.4 Continuous Query Languages

Aside from their own programming interface, some Stream Processing Systems (SPS) also provide a SQL-like querying syntax. One of the first implementations of such a Continuous Query Language (CQL) is the STREAM DSMS at Stanford. Modern DSP like Flink or Spark extend the SQL syntax with stream-specific operators for windows, sampling techniques, relation-to-stream and relation-to-relation conversion. Especially noteworthy is the Table API in Flink because it provides a uniform interface over data at rest and data in motion. [25, 26, 27]

2.3 Approximate Stream Analysis

A component relevant to any analytics project is the capability to summarize data with the goal of identifying important characteristics. Within batch systems, it is easily possible to process various user-defined queries to extract information from the data at hand. SQL is a well-known query language originally designed to allow users without advanced computer knowledge to write their own queries. Interestingly, standing queries on streams like identifying frequent items, quantiles, count or count distinct are memory intensive and do not scale well. This is mainly because keeping track of all items on unbounded streams easily exceeds the maximum available memory [23]. Yet, in many cases especially in web-scale settings, the precise answer is not necessarily needed. In these situations, an approximate result can provide the required insights and is easier to compute on
web-scale. Such approximate summarization algorithms are also called sketches. The development of approximate algorithms follows the ideal of finding solutions to compute estimates with sublinear or at worst linear memory requirements [28].

### 2.3.1 Approximate Identification of Frequent Items

A typical use case of approximation algorithms is the identification of frequent items in data streams. The target of frequent item identification can be the extraction of frequently bought items in stores, track user behavior on websites or mobile apps as well as for spam detection. Frequent items are also relevant to mine association rules. A well-known example using frequent items for mining association rules is the Apriori algorithm [29]. Two popular algorithms for finding frequent items in data streams are Misra-Gries (MG) and Space Saving (SS) algorithm. Both the MG and SS algorithm are Counter-Based Algorithms [30].

The MG algorithm provides an approximate approach to find the most frequent items in a stream. An important criterion is its fixed memory requirement. Based on a desired frequency $n$, a map of key-value pairs is created. The map is of fixed size based on a selected $k$ value, $k$ is the number of items stored in the map and it should be optimized for the expected number of items seen in the stream since it is defined by

$$\frac{n(\text{number of items})}{k(\text{map size})} = n(\text{frequency}) - 1$$

Alternative approaches are called quantile-based or linear sketches. In their study Cormode and Hadjieleftheriou found that in general, Counter-Based algorithms perform considerably better than quantile-based approaches. With regards to linear sketches the difference depends highly on the use case. An extension to the MG algorithm has been proposed to also support weighted streams and increased efficiency. It is called the SMED algorithm. [31, 32, 30]

### 2.3.2 Entropy Estimation of Streams

Another interesting case of approximate computations on streams is Entropy. Entropy is a measure of randomness which can be used for various tasks like “network traffic monitoring for the purpose of anomaly detection or traffic clustering, analysis of commercial search logs, and signal processing” [33]. Calculating entropy is related to the identification of frequent items. As presented in the work
“A Near-Optimal Algorithm for Estimating the Entropy of a Stream”, said algorithm builds upon the MG algorithm. [34]

2.3.3 Cardinality Estimation

Within large datasets counting distinct items becomes especially performance intensive because it requires memory proportional to the cardinality of the dataset. As a result, approximation can provide meaningful insights when handling large-scale datasets. The Flajolet-Martin (FM) algorithm is a well-known algorithm which has been continuously extended. First with the LogLog estimator and later with the HyperLogLog (HLL) estimator. FM is a single-pass O(n) stream algorithm. Depending on the maximum number of distinct items, its space-consumption is logarithmic O(log(m)). [28]

The algorithm:

1. Initialize bit vector of size \( L \) where \( 2^L > n \) and set each bit to 0
2. Select a random hash function to map input to natural numbers
3. Hash each input, determine number of trailing zeros \( n_z \) and set the \( n_z \)-th bit to one
4. Determine \( R \) where \( R \) is the index of first zero in bit vector
5. Number of unique inputs is \( \frac{2^R}{\phi} \), where \( \phi \) is 0.77351

The HLL variant achieves near-optimal cardinality estimation. In their respective note, the authors claim it is capable "to estimate cardinalities well beyond \( 10^9 \) with a typical accuracy of 2%" [28]. A use case where web-scale distinct counting (cardinality estimation) becomes relevant is - for example - Link-Based Spam Detection presented in [35]. Another good example is the detection for Distributed Denial of Service attacks presented in “Efficient Distinct Heavy Hitters for DNS DDoS Attack Detection” as well as in “How can sliding HyperLogLog and EWMA detect port scan attacks in IP traffic?”

K-minimum Values Sketch A specialized case of probabilistic distinct value counting is the K-minimum Value sketch (KMV). In comparison to the HyperLogLog algorithm, KMV has been extended for multiset operations (AKMV sketch). Due to this extension, they can provide document representations including support for similarity operations. To understand how the K-minimum sketch works, it is
helpful to take a step back and reflect how a "good" hash function works. Based on input \( x \), a hash function \( h(x) \) should deterministically return a pseudo-random representation of the input. This is often either a decimal or hexadecimal number (refer to SHA512). A hash function should provide an output that is evenly distributed independently of its input. This property becomes relevant for the K-minimum hash. Consider evenly distributed hashes over a defined hash space (e.g. normalized from 0-1), looking at the average distance between all hashes the number of distinct items can be estimated.

In the case of a reasonable amount of data and aforementioned good hash function, the value could estimate the average spacing. Since only one value makes the algorithm highly dependent on the quality of the hash function and increases variance, this method can be improved by not only taking into account a single value but rather the \( k \) smallest values. Hence the name k-minimum value sketch.

\[
\text{Estimated DV} = \frac{k \cdot 1}{k_{\text{max}}} = \frac{3 \cdot 1}{0.3} = 6.7
\]

\[
\text{Actual DV} = 8
\]

Figure 2.3: K-minimum spacing of hashes [38]

**Theta Sketch Framework** The class of Theta Sketches is a generalization of KMV sketches “in that KMV sketches are a form of Theta Sketch, but not all Theta Sketches are KMV sketches” [41]. Theta Sketches are designed to answer approximate cardinality queries over set expressions. In the respective paper, the example is about estimating the number of unique users satisfying a property \( P \) where \( P \) can be an arbitrary complex property or set of subqueries.

The authors of the Theta Sketch describe this in the following quote:

"Consider an internet company that monitors the traffic flowing over its network by placing a sensor at each ingress and egress point. Because the volume of traffic is large, each sensor stores only a small sample of the observed traffic, using some simple sampling procedure. At some later point, the company decides that it wishes to estimate the number of unique users who satisfy a certain property \( P \) and have communicated over its network. We refer to this as the DISTINCTTONSUBPOPULATION\( P \) problem, or DISTINCTP for short. How can
the company combine the samples computed by each sensor, in order to accurately estimate the answer to this query?” [42]

In this quote the ingress point is the point at which elements of the traffic stream enter the system, while the egress point is the point where the elements leave the system [43].

2.4 Distributed Stream Processing Frameworks

Within the area of distributed data processing exists a variety of frameworks which provide the foundation for efficient machine learning. Apache Spark a data processing framework within the Hadoop ecosystem, for example, includes a Machine Learning library and emulates stream processing as a series of staged batch computations mini-batches. In comparison, the Apache Flink framework focuses specifically on real-time distributed stream processing with additional support for batch-processing. Both frameworks offer a fluid functional programming interfaces for data processing similar to Dryad and DryadLINQ [44, 45, 46, 47]. Due to its ability to perform real continuous stream processing and its increased flexibility through advanced state mechanisms, this work will focus on Apache Flink as an exemplary framework.

2.4.1 Apache Flink

Apache Flink is an open source project developing "a framework and distributed processing engine for stateful computations over unbounded and bounded data streams" [48]. It is a platform for distributed stream and batch processing. Flink provides massively parallel data processing and runs on various cluster environments. At its core, Flink is a "streaming data flow engine" [49, 48]. Flink provides a fluent API to process and analyze data with its Table API [45]. Examples are map or join functions and support for aggregates (e.g. min, max, sum, average). Additionally, Flink can run use-case specific libraries for complex event processing, graph processing or machine Learning. Flink supports different stream elements including tuples and both Java and Scala classes to define data models. The overall architecture is presented in figure 2.4 on the next page. The various deployment models support both local development, on-premise deployment and cloud integration. [50]
Apache Flink uses a Directed Acyclic Graph (DAG) to structure the program internally, the DAG contains all operations applicable to the dataflow. The DAG allows efficient scheduling to run operations simultaneously in a distributed environment. The idea of using a DAG for data-parallel programming stems from the work on Dryad [46]. Applications running on Flink are automatically parallelized based on the user-defined pipeline. An important characteristic of Flink is its support for stateful processing, parallel operators can define and locally access a persistent state. The state can be used to store summaries, aggregations or other values. Furthermore, it is possible to externally access this state through the "queryable state" feature. As highlighted in figure 2.4, it is possible to develop higher-level libraries on top of the different programming interfaces Flink offers. In the following section, thesis related work and alternative frameworks for stream processing are presented. In the context of parallelization, it is crucial to understand the limitations of Flink. Parallel operations are processed through independent workers and threads [51]. Based on the distribution scheme, not all workers access the same data. This can influence the output of computations and potentially needs to be addressed within the application design.

2.5 Related Projects and Frameworks

In this section, thesis related projects and other frameworks within distributed stream processing are presented. Table 2.1 presents a comparison of stream processing frameworks. From an application engineer’s perspective, the guarantees
for Exactly-once processing and the fluent API prove valuable. Especially with regards to these characteristics, Apache Spark and Apache Flink currently represent advanced and reliable frameworks. A comparative study of both frameworks has demonstrated that Apache Flink can regularly outperform Apache Spark when processing non-artificially created datasets.\[52\]

Table 2.1: Comparison of stream processing characteristics \[53\]

<table>
<thead>
<tr>
<th>Streaming Model</th>
<th>Spark</th>
<th>Streaming</th>
<th>Flink</th>
<th>Apex</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Micro-batching</td>
<td>Native</td>
<td>Native</td>
<td>Native</td>
</tr>
<tr>
<td>Guarantees</td>
<td>Declarative</td>
<td>Compositional</td>
<td>Declarative</td>
<td>Declarative</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>Checkpoints</td>
<td>Log-based</td>
<td>Checkpoints</td>
<td>Checkpoints</td>
</tr>
<tr>
<td>State</td>
<td>Dedicated DStream</td>
<td>Stateful Operators</td>
<td>Stateful Operators</td>
<td>Checkpoints</td>
</tr>
<tr>
<td>Latency</td>
<td>Medium</td>
<td>Low</td>
<td>Low</td>
<td>Very Low</td>
</tr>
<tr>
<td>Throughput</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Maturity</td>
<td>High</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
</tr>
</tbody>
</table>

Yahoo Data Sketches is an open source library of approximation algorithms developed to handle large web-scale datasets. It is focused on providing high-performance and production-ready implementations of different sketch types like Frequent Item Sketches, HyperLogLog Sketches, Quantile Sketches, Sampling Sketches and Theta Sketches. Furthermore, it provides integrations with common Big Data platforms like Pig and Hive. [54]

BlinkDB is a database extending the Apache Hive stack and makes extensive use of approximate computing techniques [55]. Compared to the work at hand, the system is a batch-oriented database and uses sampling techniques to provide approximate ad-hoc queries. Although it is possible to argue that all estimation techniques use sampling of some sort, BlinkDB uses pre-computed sampling while this work uses continuous sampling. This allows the use of standing queries compared to ad-hoc queries, for more details refer to chapter 2.2.2.

VerdictDB is a recently published Approximate Query Processor (AQP) similar to BlinkDB. Yet, instead of extending an existing database technology (Hive), VerdictDB functions as a middleware and enables approximate queries over external data engines (e.g Spark, Impala or Amazon Redshift). [56]
3 Design and Development of an Approximate Query Library for Distributed Stream Processing

Within this chapter the objective is to design and develop a framework to execute approximate standing queries on data streams. Approximate queries are useful to handle large-scale data efficiently or to resolve issues due to memory limitations [23]. The work at hand also evaluates, whether approximate queries are useful in real-world scenarios. Therefore, it is helpful to understand potential use-cases of approximate queries.

A good example is the list of items customers on e-commerce websites are interested in (e.g. Top Lists). Alternatively, the advertisement industry produces massive amounts of data as well. One example, is the analysis and reporting of impression logs. In advertising an impression is the moment the ad is fetched from the source by the users’ browser. Compared to Click-Through rates, the impressions do not include whether the advertisement was clicked [57]. Impression logs can be used to answer questions about the role of gender, income or educational levels during ad-campaigns[3]. In such cases the response time of a query is more important than the precise result.

Since approximate queries potentially solve challenges handling large-scale datasets, the query library is required to handle data at scale as well. As a result, it becomes necessary to investigate potential underlying cluster frameworks which support the efficient processing of large-scale data. In chapter 2.4 on page 15 these frameworks have already been introduced. The decision which DSP to choose directly enforces constraints on the API design. This chapter focuses mostly on the design, yet the mentioned constraints are be visible throughout the design process. In chapter 3.3 on page 21 the details of the constraints and the choice of framework will be discussed in-depth.

The following section discusses (1) the use of programming language, (2) the characteristics of the query design, and (3) the dataflow within the system.
### 3.1 Design Requirements

Most of the existing general purpose BigData frameworks are either written in Java or Scala [58]. Since the author is most familiar with the Java programming language the API is implemented in Java. Similarly to the choice of the cluster framework, the choice of programming language affects the design as well. The details of this influence will be discussed within the implementation chapter.

The design an implementation focuses on on standing queries compared to ad-hoc queries. Standing queries - for the aforementioned type of request - suffer severely from expanding memory requirements. As a result, characteristics like fixed size memory layouts of the approximation algorithms become most valuable [23]. This does not entail that the library cannot be used for ad-hoc queries though. Yet, for comparison between batch and streaming systems, it is more interesting to focus on standing queries. In both situations, the idea is to enable the approximation algorithm to include every ingested stream element in the approximation. This is different to the approach of sampling data first and executing the query over the sample.

![Dataflow from source to the emitted result](Figure 3.1)

Figure 3.1: Dataflow from source to the emitted result

Figure 3.1 presents the overall data flow. The user defines (1) where the data originate from, (2) optional preprocessing of the incoming elements, and (3) the type of query to execute. The steps one and two are the usual parts of data processing.
pipelines. For three, the user defines which key is used for the query. The queries for frequent items and cardinality estimation support a single key while the HyperLogLog query supports a grouping key (group by operation) and the value which is counted.

The execution of the queries is automatically parallelized over multiple tasks. The parallelization is based on the underlying cluster framework. As a result, the level of parallelism also depends on the configuration of the cluster. Most modern distributed stream processing systems use a partitioning scheme to distribute the workload over all available workers. This needs to be considered within the implementation to optimally use the approximation scheme. If the data distribution is ignored, our experiments have shown that many of the sketches increasingly (>= 50% error depending on map size) loose accuracy. One example is the Frequent Item algorithm. Given its fixed memory size, the algorithm removes data from the internal state once its capacity is exhausted. If the data are nearly similar, no frequent items can be extracted because the sketch has an internal noise[54,59]. Therefore, the accuracy of the estimates depends on the distribution of the data when processing them in parallel. In this scenario it is preferable to consistently distribute the data based on hashing to process the same keys in a sketch therefore reducing the number of removal operations executed within the sketch. For the HyperLogLog algorithm the same holds true except it is for different reasons. The implemented group by operation requires tracking of the key on which to group the data. As a result the hashing operation for distributing data over the task worker reduces the size of the hash maps since the same keys are processed in the same worker.

### 3.2 API Specification

The design of the library consists of multiple components. A group of data-models representing and implementing the respective approximation algorithms. Following the OO-pattern, the models include functionality to update, merge and retrieve the underlying sketches and their estimates.

Processing functions are attached to the data stream and feed the elements into the data-model. The processing functions are implementation dependent and require an understanding of the related cluster framework. These processing functions are encapsulated by a group of query functions. The query functions represent the high-level functionality of standing queries for cardinality estimation, frequent item estimation, and quantile estimation. The separate components are presented in figure 3.2 on the facing page.
Figure 3.2: Architecture of the query library

The following chapter will present the actual implementation of the architecture. It will also highlight external constraints based on the choice of language and the underlying cluster framework. A prominent focus is the objective to create high-level functions which are easy to understand and can be integrated into existing applications with minimal effort.

3.3 Implementation Requirements

In the following sections the details of the implementation are discussed. It is important to notice the impact of the underlying stream processing engine on the design. Although the library components are framework-agnostic, the overall structure is influenced by the Flink application model. Apache Flink is chosen because its scaling properties are well-proven, it offers stateful processing, a fluent API, and fault-tolerance [60, 61]. Given the objective of developing a DSP-based prototype the Flink framework provides a solid foundation. It provides the necessary environment for parallel execution and resilient processing. Furthermore, because Flink is a true streaming framework and has stateful functions, it provides valuable features for future work. As a result, it is chosen over Spark Streaming.

To understand the prototype implementation, a fundamental knowledge of the Flink specific operators is important. These will be explained in the following chapter. The prototype is available as an open source implementation.

https://github.com/tlindener/ApproximateQueries/
3.3.1 Flink Data Types

Flink has several data types which are relevant to understand the different design choices and potential constraints.

The **tuple** is the base item type within a Flink stream. A tuple has a number of attributes, starting with a tuple of one up to a tuple of 25 attributes. The tuple-type is generic and can store both Java primitives and Java objects.

**DataStream** is as an unbounded collection of elements within a stream. Like the tuple, it is generic and can include user-defined objects (POJO) as well as tuples.

The **KeyedStream** is a specific type of DataStream. It is created by applying the “key by” operation on a DataStream. This distributes the elements of the underlying DataStream - as well as its state - depending on the defined key. Since the key is hashed, this does not necessarily mean that an Operator on a KeyedStream only sees one specific key (e.g. on a sequence of A, B, C, D all items can end up in the same KeyedStream instance).

**WindowedStream** is a DataStream that has been discretized into windows: finite sets of elements. The use of a Window based on the KeyedStream allow parallel computation through multiple tasks, since the finite sets can be distributed over the available workers. [62]

3.3.2 Flink Operators

The main operators relevant to the development are the KeySelector, FlatMap, Aggregate and Window operator. Flink includes a variety of other operators which are not pertinent at this point.

The **KeySelector** operator is usually used to create a KeyedStream. It works as a user-defined function where the developer can directly influence how the attributes of a tuple or object are used to create the key. This is usual in this case to allows the user to combine multiple attributes to a composite key.

**FlatMap** is an operator which executes a function on each element of the underlying stream and can output zero, one or multiple elements. This characteristic allows for flexible output patterns. For example, it is possible to only aggregate on every 100th item. The resulting reduction of emitted items can potentially impact the performance positively.
The **Aggregate** operator is applied on a window and executes a user-defined aggregation over the items in a window.

Finally, the **Window** operator assigns a window to the DataStream and creates a WindowedStream.

### 3.4 API Implementation

The API is separated into two classes which provide the different types of queries as static Java methods. Due to language constraints in Java it is not possible to use runtime patching\(^2\) to add methods to the DataStream type [63]. As a result, it is not possible to add fluent API methods without modifying Flink Core. Overall the integration is done through the use of common Flink functions. The API presented below is a processing pipeline composed of one key-by operation and a flatmap operation. 3.3 on the next page demonstrates the chaining process where the key-by operation is necessary to distribute the data depending on the relevant key, while the flatmap operation connects the KeyedStream with the instances of the sketch model (Java class). The implemented operators are exposed separately and can be integrated into existing applications independently of the high-level static methods.

---

\(^2\)Runtime Patching is a technique known from dynamic programming languages to extend a library or class at runtime with additional methods. E.g. in C# so-called extension methods allow to add new functionality to an existing class as if the method was originally part of the class itself. [63]
3.4.1 Frequent Items and Quantiles

For quantiles and frequent items the minimal method signatures are listed below. The idea is to provide both customizable functionality as well as good defaults to reduce the number of parameters for the queries. In the extended versions, various sketch specific parameters can be defined as well.

1. `public static <T> DataStream<TopNQueryResult> continuousFrequentItems(DataStream<T> inputStream, KeySelector valueSelector, int maxItems, int emitMin)`

2. `public static <T> DataStream<QuantileQueryResult> continuousQuantiles(DataStream<T> inputStream, KeySelector valueSelector)`

The KeySelector is used in both cases to allow the user to customize which attribute (or attributes) is used for the processing. This allows the use of generics in the API. Furthermore, the KeySelector is also used as a hint to the system how to distribute the data over the workers. Details on the reasoning behind this decision are presented in chapter 3.5. The "maxItems" parameter is used to filter...
the number of estimates retrieved from the sketch. The "emitMin" parameter is used to define, how often the results from a sketch are emitted downwards of the dataflow pipeline (0 - on every element, otherwise every n-th element).

### 3.4.2 Cardinality Estimation

In the case of cardinality estimation, the HyperLogLog and the Theta algorithm are the most interesting. HyperLogLog usually performs well for most use-cases and is of comparably low complexity. The Theta sketch is preferable when set operations are performed on the results.

```java
public static <T> DataStream<HllSketchAggregation> runContinuousHll(DataStream<T> input, KeySelector keySelector, KeySelector valueSelector, int emitMin)
```

```java
public static <T> DataStream<ThetaSketchAggregation> runContinuousTheta(DataStream<T> input, KeySelector keySelector, KeySelector valueSelector, int emitMin)
```

Similarly, to the previously discussed sketches, the first KeySelector is used to distribute the events over the workers. This behavior represents a "groupBy" operation and optimizes the capacity of the sketches. In this case the value selector chooses the item type to apply the distinct count.

### 3.5 Challenges

The efficiency of distributed stream processing is highly dependent on the distribution of the data at hand. DSP use data parallelism to process data faster and therefore reduce latency, yet data in real-world applications is often skewed [64]. This can cause problems because workers have to process multiple hot keys while other workers have only cold keys assigned. Depending on the skewness of the keys in a dataset, some keys are significantly more frequent than others. These are called hot keys, while keys which are significantly less frequent are called cold keys. Depending on the distribution of hot keys over the workers, a high amount of hot keys can put increased computational load on single workers.

By default, Flink distributes data randomly over the worker using a shuffle mechanism. This shuffling can lead to inefficiencies of the presented algorithms. Taking the frequent items sketch as an example. Its capability of estimating frequent items is based on the internal map size. If the keys are randomly distributed over multiple workers (with their respective instance of a sketch), they
see the same keys. This reduces the overall capacity of the sketch and decreases accuracy. If in contrast, the DSP distributes the same keys to the same workers, the capacity of each sketch is used in an optimal way leading to a higher overall accuracy of the estimates. Intermediate tests during the prototyping, have shown that in many cases the estimates never go below 50% deviation.
4 Evaluating Constraints and Performance of the Prototype

The overall objective is (1) to identify whether significant performance gains are achievable through approximate queries (e.g., in terms of processing time or memory consumption), (2) analyze the accuracy of the approximation techniques and extract useful default parameters and (3) present the scalability characteristics of the sketches on the data distribution.

The first test-case compares the performance in terms of execution time and memory consumption of the approximate and exact frequent item implementation. This is interesting because it underlines from what point on exact calculations become limited due to their memory requirements while approximate query can still provide estimations. In this context, it is worth mentioning though that Apache Flink can make use of an external memory back-end to handle cases with excessive memory demands. Nonetheless, approximate techniques can prove useful if such an infrastructure is unavailable or too costly. The second test is about the accuracy of the sketches and analyzes the approximation characteristics in general. This test is relevant to identify the limitations of the approximation algorithms. Within the evaluation, the exact counts (or distinct counts) are compared with the estimates. The comparison is done both relatively (in percent) and with absolute values.

\[ \text{deviation\_percent} = 100 \times \frac{|\text{estimate} - \text{count}|}{\text{count}} \]

The third part of the analysis takes a closer look at the scalability characteristics. The previously analyzed results are taken into account for this analysis. The respective scripts for each analysis can be found in the github repository\footnote{Refer to \url{https://github.com/tlindener/ApproximateQueries/blob/master/analysis}}.

4.1 Datasets

Referring back to the research question in chapter 1.3, the datasets have been chosen based on existing usage of approximate queries in the e-commerce and...
telecommunication industry [3, 4]. Dataset one is the Amazon Reviews dataset [65, 66]. The dataset contains product reviews crawled from the Amazon e-commerce website. In this scenario, the “Ratings-only” subset is used. It has the following fields: user, item, rating, timestamp. The main difference from the full dataset is the absence of the review text. Figure 4.1 is a histogram over the reviewers (user) with customized bins. The majority of reviewers only wrote a single review. Users without review are absent since the dataset is created by scraping the existing reviews without taking into account the complete list of users.

![Figure 4.1: Histogram over reviewer count - bins = [0,1,2,3,5,50000]](image)

Potential queries could include the number of ratings per user or product which are reviewed more than the majority of products (heavy hitters). Figure 4.2 on the next page illustrates the reviews grouped by product rating. Since the sketches at hand are specifically relevant in the context of large key spaces, the further analysis focuses on user (21,176,521 distinct users) and item (9,874,210 distinct items) queries.
Secondly, the WikiTrace dataset is used as a comparison for the frequent item sketch. This dataset is an access log of Wikipedia resources (as URL) from September 2017 it has a total size of 9GB [67]. The URL is the only relevant key for analytical queries within this scenario. In total there are 6,708,723 distinct keys with a total row count of 79,093,443. The mean URL count is 11.7 with a skew of 491. This means the URL count is substantially skewed to the right. This is expected since in this scenario, certain resources (e.g. the main page) are accessed more frequently than others.

4.2 Evaluation Environment

The evaluation is run on an AMD Ryzen 1600 with 6 cores (12 with SMT) and 16 GB RAM. The operating system is Ubuntu 18.04 and Flink version 1.4.2 is used in standalone mode. The level of parallelism is based on the logically available cores (12). When analyzing the performance (based on processing time) of the approximation techniques it can be assumed that all measurements are limited by the I/O speed of the SSD. This can be observed because the processing time is only influenced significantly by the dataset size but neither map size or query key.
4.3 Identifying Limitations of Exact Frequent Item Extraction

The frequent item implementation is done naively in Java using a standard library hash-map. The results have highlighted a substantial performance difference between approximate and exact implementation. The settings of the JVM have been optimized to utilize the overall available RAM to its limits (heap size of 10GB and reserving 2-3 GB for Java related overhead). These optimizations are solely relevant for the exact calculations. The approximate implementation running on the Amazon Ratings dataset takes on average 90 seconds to complete the whole dataset. There is no clear correlation between execution time and map size within the experiments (see figure 4.3). Throughout the processing, the memory consumption of the java processes does not exceed 2 GB. System-wide the overall memory consumption stays at slightly (exact values vary minimally over multiple executions) more than 5 GB. This is in line with the fixed-memory design of the underlying sketch algorithm.

![Figure 4.3: Execution time depending on map size (Amazon Rating dataset)](image)

The exact implementation consumes most of the available heap within 15 to 30 seconds and starts to stay consistently at 10GB for the main process while taking another 900MB for a second Java runtime process (usually related to Garbage Collection). The CPU processing averages around 80%, it can be assumed that most of this result from the JVM garbage collection and the hash map operations. The
overall processing time varies significantly between 5 to 12 minutes to process the dataset using Flink Processing Time.

Evaluation with larger datasets like the WikiTrace dataset using the exact implementation fails with an OutOfMemory Exception. This behavior is expected because of the growing memory demand of the hash-map in the exact frequent item implementation. Figure 4.5 demonstrates the significant difference in memory demand, while figure 4.4 highlights the increased execution time for exact processing.

With regards to figure 4.6, the execution time varies slightly between the map sizes. A clear correlation does not exist though, in some cases the smaller map sizes, show an increased execution time, this could be caused by an elevated number of hashing operations. The hashing operations take longer than the value look-up and the "increase count" operation. While for different queries on the same dataset the execution time stays consistent (indicates an I/O bottleneck), the processing of the larger WikiTrace dataset is slower. Yet, again no clear correlation between map size and execution time becomes apparent.

---

2See detailed description at [https://docs.oracle.com/javase/8/docs/api/java/lang/OutOfMemoryError.html](https://docs.oracle.com/javase/8/docs/api/java/lang/OutOfMemoryError.html)
4.4 Accuracy comparison of Frequent Items

To analyze the accuracy of the sketch multiple tests are run over the datasets. By default the DSP continuously emits intermediate results. To reduce the amount of generated data within this process, the developed functions emit only intermediate results. This causes a level of uncertainty within the measurements. Throughout the tests, only every 1000th result is emitted to the dataflow pipeline. The process is deterministic throughout the tests; therefore, repetition of tests provide equal results but when - for some keys - the Flink task does not reach the limit of 1000 elements it is not emitted and therefore impact the accuracy. In general, we will look at either the deviation or the deviation in percent. To better compare scalability, different map sizes are compared in a single plot.

4.4.1 Amazon Ratings Dataset

The first query is selected to identify the highest contributing reviewers. Since the accuracy of the frequent item sketch depends on the map size parameter, four
different (128, 1024, 4096, 16384) parameters are used for the comparison. With an increase in map size the capacity of the sketch grows. As a result, the internal noise of the sketch is expected to decrease. Based on the increased capacity, the sketch can estimate the reviewer counts more precisely. Throughout the test run, the memory consumption (on Java process level) measured never exceeds the 2 GB for any map size. Figure 4.7 presents the results with map size 128. The deviation is computed by (1) calculating the count per element (in this case based on the reviewer id), (2) extracting the last output from all computed estimates, and (3) subtracting the count from the estimate.

![Figure 4.7: Deviation of frequent item estimation ordered by exact frequency (Amazon Rating, map size=128)](image)

Based on the map size the capacity of the sketch is insufficient to estimate more than three items and even within these three, there is a high deviation from the exact count. With an increase of the map size, the deviation develops a tail where the items with the high counts are estimated with varying deviation while the items at the bottom have a low deviation (see figure 4.8 on the following page). This behavior seems counter-intuitive and the following analysis will go into greater detail to understand this behavior.
When comparing only the top 300 items (based on exact counts) the results differ significantly as presented in figure 4.9 on the next page. On the X-Axis is the index of the count in descending order while on the Y-Axis is the deviation in percent. In this scenario, the higher map size significantly impacts the accuracy of the estimations. While at a map size of 4096 the deviations go as far as 100%, at a map size of 16384 the deviation merely reaches 10%.
Interestingly, the difference in memory requirement is not measurable (on a process level) between the map size executions. This behaviour enables a substantial increase in terms of map size depending on the dataset. The analysis has shown that the accuracy is highly dependent on the map size. As a result, the possibility to use large map sizes with potentially little increase in memory can increase the accuracy of the estimates. Larger map sizes have not been tested, because the amount of generated data exceeded the available disk space.

### 4.4.2 WikiTrace Dataset

In case of the WikiTrace dataset, the query identifies the web resource most frequently accessed in the logged time-frame (September 2007). We will look at the difference of varying map sizes again to understand the capacity limitations of the sketches. Interestingly, the sketch performs considerably better even with lower map sizes. In figure 4.10 on the following page up to 80 items are estimated.
with low deviations from the exact count. The difference could be explained due to the data distribution in the datasets. A detailed analysis of this is given in chapter 4.5 on page 39. Overall, this indicates a need to adjust the map size according to the data distribution. In the previous analysis, the increase of the map size demonstrated a higher accuracy with similar patterns.

Figure 4.10: Deviation from exact count ordered by exact frequency (WikiTrace, map size = 128)

In this scenario, figure 4.11 on the facing page displays a different pattern than previously seen. The initial estimates are quite precise and start to drop at index 500. This is expected to be due to the internal noise level where the algorithm cannot distinguish between similar items.
Figure 4.11: Deviation from exact count ordered by exact frequency (WikiTrace, map size = 1024)

Figure 4.12 on the next page presents the impact of the map sizes in percent for the WikiTrace dataset. Again, the top 300 items are selected. In this scenario the sketches perform significantly different than in the case of the Amazon Rating dataset. Despite the increase in dataset size (9 GB instead of 3 GB), and a small map size (1024), the capacity of the sketch allows predictions of the top 300 items at similar error to the larger map sizes. Overall, the number of distinct items (e.g. 21,176,521 reviewers, 9,874,210 products and 6,708,723 URL) is an important factor for this behavior. The number of distinct items combined with the count distribution provides a reasonable indication of the required map size within an application. With increased number of distinct items larger map sizes are necessary for accurate predictions. Additionally, depending on the distribution an even larger map size is needed.
Figure 4.12: Comparison of map sizes for the frequent item sketch (WikiTrace, Top 300)

The upper two plots in figure 4.13 on the facing page highlight the limits of sketches with small map size. The graph presents four sub plots with the map sizes 128, 1024, 4096 and 16386. The results plotted are limited to the top 6000 items (ordered by exact frequency). With map size 128, the sketch is limited to estimating 150 frequent items as a maximum. And its error is significant for all items beyond the top 100. Similarly, for map size 1024, the sketch is unable to provide estimates for more than 1500 items.
4.5 Impact of Data Distribution

Throughout the analysis, the importance of optimizing the sketch parameters to the respective dataset has become apparent. This chapter will analyze the impact of the data distribution in greater detail. This will also include the analysis of a cardinality estimation sketch based on the HyperLogLog algorithm.

4.5.1 Frequent Items

While the previously analyzed query was based on the reviewer Id, it is also interesting to look at the product Id. The SQL statement would look like this:

```sql
SELECT product, COUNT(product) FROM ratings GROUP BY product
```

In contrast to the previous query, this query identifies more heavy hitters. While the dataset contains nearly 10 million distinct products, there are around 21 million unique users present. Yet, the users have two distinct heavy hitters compared...
Table 4.1: Comparison between product and users (heavy hitters)

<table>
<thead>
<tr>
<th>Product</th>
<th>Product Count</th>
<th>User</th>
<th>User Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0054JZC6E</td>
<td>25368</td>
<td>A14OJS0VWMOSWO</td>
<td>44557</td>
</tr>
<tr>
<td>B00FAPF5U0</td>
<td>24024</td>
<td>AFVQZQ8PW0L</td>
<td>29044</td>
</tr>
<tr>
<td>B009UX2YAC</td>
<td>23956</td>
<td>A32889RN35MS68</td>
<td>9773</td>
</tr>
<tr>
<td>439023483</td>
<td>21398</td>
<td>A9Q28YTLREO7</td>
<td>9243</td>
</tr>
<tr>
<td>030758836X</td>
<td>19867</td>
<td>A2EDZH5IXHFA9B</td>
<td>6431</td>
</tr>
<tr>
<td>B0051VVOB2</td>
<td>19529</td>
<td>A320TMDV6KCFU</td>
<td>6195</td>
</tr>
<tr>
<td>B005SUHPO6</td>
<td>18688</td>
<td>A2F6N60Z96CAJI</td>
<td>6171</td>
</tr>
<tr>
<td>B00G5LQ5MU</td>
<td>18645</td>
<td>A2OJW07GQRNJUT</td>
<td>5692</td>
</tr>
<tr>
<td>B0074BW614</td>
<td>18244</td>
<td>A16CZRL23NOIW</td>
<td>5613</td>
</tr>
<tr>
<td>B000GF7ZRA</td>
<td>17948</td>
<td>A2AIMXT9PLAM12</td>
<td>5494</td>
</tr>
<tr>
<td>B00DR0PDNE</td>
<td>16454</td>
<td>A1S3C5OFU508P3</td>
<td>5378</td>
</tr>
<tr>
<td>B0064X7B4A</td>
<td>16239</td>
<td>AHD101501WCN1</td>
<td>5155</td>
</tr>
<tr>
<td>B00DJFIMW6</td>
<td>15601</td>
<td>A281NPSIM1C2R</td>
<td>5093</td>
</tr>
<tr>
<td>B0092CF6W</td>
<td>15294</td>
<td>A13QTZ8CIMHHG4</td>
<td>4797</td>
</tr>
<tr>
<td>B00AREIA18</td>
<td>15080</td>
<td>A2NJ8E954DBH</td>
<td>4794</td>
</tr>
<tr>
<td>B00AWH595M</td>
<td>14393</td>
<td>A1X8VZWTG8IS6</td>
<td>4730</td>
</tr>
<tr>
<td>B005ZXMUS</td>
<td>14114</td>
<td>A2R9M38FRB8608</td>
<td>4420</td>
</tr>
<tr>
<td>B007WTAJTO</td>
<td>14114</td>
<td>A2TX179KAT5GRP</td>
<td>4290</td>
</tr>
<tr>
<td>439023513</td>
<td>14114</td>
<td>A2WQY1B8Z57QRZ</td>
<td>4253</td>
</tr>
</tbody>
</table>

to a smoother distribution for the products. Figure 4.14 on the next page and figure 4.15 on the facing page illustrate the difference in data distribution as well. The plots present the log(item count) for the products as well as for the reviewer. Although the line plot looks similar, in the case of the reviewer there are twice the number of events counted.

If we limit the number of items selected from the results we get the results presented in figure 4.16 on the next page. At first, it looks similar to figure 4.9 on page 35. Due to the smoother distribution starting with map size 4096, the scatter looks more structured and less noisy. Especially with map size 16384, the deviation has a clear growth pattern. This means the heavy hitters in the dataset have lower estimation errors while the less frequent items see an increase in error. Again, this underlines that the map size has to be selected depending on the overall size of the dataset. Furthermore, depending on the distribution of the data per key the sketch requires a larger map size to accurately estimate the frequency of the elements. Within all test cases, selecting a larger map size seems advisable given the aforementioned immeasurable memory difference. As discussed before, an impact on the execution time is not measurable as well.

Comparing figure 4.13, figure 4.16 and figure 4.17 the impact of the map size is
Figure 4.14: Log Item Count for Products

Figure 4.15: Log Item Count for Reviewers

Figure 4.16: Deviation for product estimates limited by 1000

Figure 4.17: Deviation for reviewer estimates limited by 1000
again underlined. While in the case of the WikiTrace dataset, the sketch of same capacity can nearly estimate 6000 items with less than 20% error, only 400 users (reviewers) can get estimated before even at the largest tested map size the error is dominated by noise. In the case of the reviewers, the performance overall appears to be worse (lower right plot, figure 4.17 on the previous page). In contrast to this, for the product, the error gradually increases instead of appearing suddenly (lower right plot, figure 4.16 on the preceding page).

4.5.2 HyperLogLog

The execution behavior of the HyperLogLog sketch varies compared to the previously analyzed sketch. This is mainly due to the grouping operating. Without the grouping, the execution time is similar to the frequent item sketch. The query reflects the following SQL statement:

```sql
SELECT COUNT(DISTINCT(item)) FROM ratings GROUP BY user
```

Due to memory constraints on the test system, the query is executed on the largest published subset of the dataset (book reviews only). In comparison to the frequent item sketch, the mean deviation in percent is only two.

![Image of HyperLogLog](42)

Figure 4.18: Estimate deviation and exact count for the top 1000 items
Table 4.2: Statistical summary of HyperLogLog query

<table>
<thead>
<tr>
<th>summary</th>
<th>diff</th>
<th>diff_percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>2322221</td>
<td>2322221</td>
</tr>
<tr>
<td>mean</td>
<td>0.033</td>
<td>2.0709</td>
</tr>
<tr>
<td>stddev</td>
<td>13.621</td>
<td>5.828</td>
</tr>
</tbody>
</table>

Figure 4.18 on the preceding page illustrates how the estimates oscillate around the exact distinct count. Depending on the grouping keyspace size, this demonstrates a high potential for using a combination of exact data grouping (instead of a sampling approach) with approximate cardinality estimation. The usability is not as straightforward compared to frequent item analysis but can still be used in certain situations. Given the limited amount of interesting features in the WikiTrace dataset (no grouping key), the HyperLogLog sketch was not tested with this dataset.

Table 4.2 shows the statistical summary of the HyperLogLog query. The average deviation in percent is 2.07 and therefore in line with the accuracy presented in the original HyperLogLog paper [28].

4.6 Error Bounds

The presented sketches all include the possibility to retrieve the error bounds for the estimates. The error bounds are dependent on the chosen error type (e.g. no false positives/no false negatives). Throughout the analysis, the error bounds have shown varying value to understanding the deviation from the exact results. Depending on the error type, either the upper or the lower bound is equivalent to the estimate. Table 4.3 on the following page and table 4.4 on the next page provide a statistical summary of the results with map size 16368. In both cases, the upper bound is equivalent to the estimate (with slight deviations) while the lower bound is visibly below the estimates. Figure 4.19 on the following page plots the error bounds for the top 300 items. The blue line marks the absolute deviation with the upper bound plotted on top. The green line is the lower bound. Exposing the error bounds provides little value to the user because the estimation is always close or equal to the upper bound than to the lower bound. This behaviour is equivalent with regards to figure 4.20 on page 45. In this figure, the plot is created for the top 300 results of the Product Query from the Amazon Rating dataset.
Table 4.3: Statistical summary WikiTrace results

<table>
<thead>
<tr>
<th>summary</th>
<th>diff</th>
<th>diff_percent</th>
<th>diffLower</th>
<th>diffUpper</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>246697</td>
<td>246697</td>
<td>246697</td>
<td>246697</td>
</tr>
<tr>
<td>mean</td>
<td>-33.82</td>
<td>71.67</td>
<td>-41.36</td>
<td>-33.82</td>
</tr>
<tr>
<td>stddev</td>
<td>56.71</td>
<td>35.34</td>
<td>64.81</td>
<td>56.71</td>
</tr>
<tr>
<td>min</td>
<td>-534</td>
<td>0</td>
<td>-744</td>
<td>-534</td>
</tr>
<tr>
<td>max</td>
<td>191</td>
<td>100</td>
<td>0</td>
<td>191</td>
</tr>
</tbody>
</table>

Table 4.4: Statistical summary Amazon Rating results (Products)

<table>
<thead>
<tr>
<th>summary</th>
<th>diff</th>
<th>diff_percent</th>
<th>diffLower</th>
<th>diffUpper</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>285586</td>
<td>285586</td>
<td>285586</td>
<td>285586</td>
</tr>
<tr>
<td>mean</td>
<td>27.85</td>
<td>25.00</td>
<td>-20.84</td>
<td>27.85</td>
</tr>
<tr>
<td>stddev</td>
<td>82.43</td>
<td>32.81</td>
<td>63.00</td>
<td>82.43</td>
</tr>
<tr>
<td>min</td>
<td>-2</td>
<td>0</td>
<td>-735</td>
<td>-2</td>
</tr>
<tr>
<td>max</td>
<td>735</td>
<td>100</td>
<td>0</td>
<td>735</td>
</tr>
</tbody>
</table>

Figure 4.19: Error bounds for WikiTrace dataset (1000 elements)
Figure 4.20: Error bounds for Amazon dataset (Product, 300 elements)
4.7 Summary

In this section we want to summarize the results of the evaluation in reference to the research question in chapter 1.3.

Under which circumstances do approximate algorithms work on distributed streaming environments?
When distributing the sketches over multiple execution units (EU)\(^3\), great care has to be taken considering the distribution of the data over the EU. Apache Flink attempts to shuffle the data when distributing work over multiple EU. This can reduce the capacity of a sketch significantly and results in estimates with no less than 50% deviation.
The distribution scheme of the DSP heavily impacts the accuracy of approximate algorithms. Using a key-based scheme can successfully reduce this impact.

What is the impact of parallel computing and the impact of the data distribution of datasets on the accuracy of the approximations?
In general, the experiments have shown that skewed data negatively influence the accuracy of the sketches. Tests with reduced parallelism have shown, that this is independent of the parallel execution. Yet, execution over multiple threads increases the accuracy because multiple instances of the sketch increase overall capacity. This is the case because every EU has its own instance of a sketch with the defined capacity (e.g. through the map-size parameter). The tests with the WikiTrace dataset indicate, that not the number of overall elements (rows in the dataset) but rather the number of distinct elements impacts the accuracy. This is still dependent on the map-size of course.
In summary, the skewness of data impact the accuracy of the sketches significantly. The impact can be reduced by adjusting the parameters without measurable influence on performance.

How can the choice of algorithm parameters positively influence the approximation results and what are the lower limits?
Looking at the frequent item sketch, a large map-size always increases the overall accuracy of the sketch. Yet, depending on the required number of top results (e.g. top 50, top 100 or more) and the data distribution, even small map-sizes can be sufficient. Given the immeasurable increase in map-size, a larger map-size is generally recommended though. When implementing a solution, the developer

---

\(^3\)An execution unit can be a thread or a process, either on a single or on multiple workers in a cluster
should evaluate the impact of the map-size on the estimates based on an evaluation dataset. Within the given evaluation, optimizing map-sizes always improves the accuracy without impacting performance.

Finally, we want to answer the main research question:

Can we create accurate and efficient memory constraint standing queries for unbounded data on distributed stream processing while distributing data over multiple instances of approximate algorithms?

The efficiency of the frequent item sketch is successfully presented in chapter 4.3. In every scenario, the approximation significantly outperformed the exact implementation. Furthermore, the comparison of the accuracy in chapter 4.4 has also highlighted the high accuracy in many cases. Yet, it is also worth noting that whether the accuracy is sufficient or not is dependent on the test. The analysis of users within the frequent item queries has highlighted a lack of performance on highly skewed data. The work has identified potential solutions to mitigate this effect. Nonetheless, the accuracy is significantly lower than in other cases. In in-depth analysis of the dependency between accuracy and the data distribution is given in chapter 4.5. This needs to be considered when choosing the parameters for the sketches (e.g. map size). In conclusion, with the correct engineering decisions, the approximation algorithms work well in a distributed execution model while requiring significantly less memory than exact processing. Therefore, memory constraint standing queries with up to 2% accuracy are feasible.
5 Discussion

Within certain boundaries, the existing data sketch implementation works well on a DSP. Due to the consistent use of Keyed Streams, the capacity of the sketches is optimally used. Nonetheless, there is room for improvement. During the prototyping phase, the use of stateful execution resulted in a memory-drain. This could cause the processing to fail once the memory was exhausted. As a result, the stateful capabilities are not used in the current prototype.

5.1 Frequent Item Sketch

The frequent item sketch most clearly demonstrates the advantages of approximation techniques. By correctly optimizing both the map size as well as the number of items (TOP/LIMIT by n), the deviation from the exact counts stays below 10%. It also correctly identifies the heavy hitters with nearly one percent deviation. With the datasets used in the evaluation, the keys are well distributed over all threads within Flink. This is measurable through the output generated per thread. Compared to an exact equivalent, the memory consumption and the CPU consumption is lower. Even with comparably small datasets (e.g. 3GB for the Amazon Ratings), the native implementation shows a high memory consumption. In contrast, the approximate version scales well with varying map sizes. During the tests, the execution time stays nearly constant for the approximate execution.

Table 5.1 on the facing page illustrates the error and the accompanying error bounds for the top 20 estimates. The relatively low deviation underlines the ability of these sketches to approximate information in a potentially useful manner. The concrete usefulness depends on the required accuracy of the use-case.
Table 5.1: Snapshot of top 20 product estimates (Frequent Item, Amazon Ratings)

<table>
<thead>
<tr>
<th>product</th>
<th>estimate</th>
<th>lowerBound</th>
<th>upperBound</th>
<th>count</th>
<th>diff</th>
<th>diff_percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0054JZC6E</td>
<td>25859</td>
<td>25132</td>
<td>25859</td>
<td>25368</td>
<td>491</td>
<td>1.936</td>
</tr>
<tr>
<td>B00FAPF5U0</td>
<td>24724</td>
<td>23990</td>
<td>24724</td>
<td>24024</td>
<td>700</td>
<td>2.914</td>
</tr>
<tr>
<td>B009UX2YAC</td>
<td>24568</td>
<td>23843</td>
<td>24568</td>
<td>23956</td>
<td>612</td>
<td>2.555</td>
</tr>
<tr>
<td>439023483</td>
<td>21405</td>
<td>20753</td>
<td>21405</td>
<td>21398</td>
<td>7</td>
<td>0.033</td>
</tr>
<tr>
<td>030758836X</td>
<td>19870</td>
<td>19218</td>
<td>19870</td>
<td>19867</td>
<td>3</td>
<td>0.015</td>
</tr>
<tr>
<td>B0051VVOB2</td>
<td>20012</td>
<td>19287</td>
<td>20012</td>
<td>19529</td>
<td>483</td>
<td>2.473</td>
</tr>
<tr>
<td>B005SUHPO6</td>
<td>19156</td>
<td>18501</td>
<td>19156</td>
<td>18688</td>
<td>468</td>
<td>2.504</td>
</tr>
<tr>
<td>B00G5LQ5MU</td>
<td>19342</td>
<td>18617</td>
<td>19342</td>
<td>18645</td>
<td>697</td>
<td>3.738</td>
</tr>
<tr>
<td>B0074BW614</td>
<td>18736</td>
<td>18083</td>
<td>18736</td>
<td>18244</td>
<td>492</td>
<td>2.697</td>
</tr>
<tr>
<td>B000GF7ZRA</td>
<td>18153</td>
<td>17498</td>
<td>18153</td>
<td>17948</td>
<td>205</td>
<td>1.142</td>
</tr>
<tr>
<td>B00DR0PDNE</td>
<td>17136</td>
<td>16410</td>
<td>17136</td>
<td>16454</td>
<td>682</td>
<td>4.145</td>
</tr>
<tr>
<td>B0064X7B4A</td>
<td>16759</td>
<td>16040</td>
<td>16759</td>
<td>16239</td>
<td>520</td>
<td>3.202</td>
</tr>
<tr>
<td>B00DJFMW6</td>
<td>16887</td>
<td>16168</td>
<td>16887</td>
<td>16221</td>
<td>666</td>
<td>4.106</td>
</tr>
<tr>
<td>B008Q77Y0</td>
<td>16323</td>
<td>15589</td>
<td>16323</td>
<td>15601</td>
<td>722</td>
<td>4.628</td>
</tr>
<tr>
<td>B00992CF6W</td>
<td>15897</td>
<td>15172</td>
<td>15897</td>
<td>15294</td>
<td>603</td>
<td>3.943</td>
</tr>
<tr>
<td>B00AREIAI8</td>
<td>15647</td>
<td>14995</td>
<td>15647</td>
<td>15080</td>
<td>567</td>
<td>3.760</td>
</tr>
<tr>
<td>B00AWHS95M</td>
<td>15035</td>
<td>14301</td>
<td>15035</td>
<td>14393</td>
<td>642</td>
<td>4.461</td>
</tr>
<tr>
<td>B005ZXWMUS</td>
<td>14835</td>
<td>14100</td>
<td>14835</td>
<td>14310</td>
<td>525</td>
<td>3.669</td>
</tr>
<tr>
<td>B007WTAJTO</td>
<td>14743</td>
<td>14009</td>
<td>14743</td>
<td>14172</td>
<td>571</td>
<td>4.029</td>
</tr>
<tr>
<td>439023513</td>
<td>14125</td>
<td>13399</td>
<td>14125</td>
<td>14114</td>
<td>11</td>
<td>0.078</td>
</tr>
</tbody>
</table>

5.2 HyperLogLog Sketch

In the case of the HyperLogLog sketch, the situation is different compared to the frequent item sketch. The frequent item sketch can be used as-is and does not require grouping. In comparison, the distinct counts (or cardinality estimation) of the HyperLogLog sketch are mainly interesting when grouping the results. For example, how many distinct products did a customer buy. To make this possible, the state of each sketch must be stored per grouping key. This implementation creates an increasing memory demand similarly to exact processing. This reduces the applicability of the sketch compared to the frequent item implementation. While it works with a small grouping keyspace, a large number of keys causes the workers to run out of memory. The same holds also true for the class of Theta Sketches. This chapter discusses the HyperLogLog implementation as a representation of both sketches. In contrast to the previously analyzed sketch, the HyperLogLog implementation does not include a parameter like the map size. As a result, only one experiment is executed.
5.3 Future Work

The results from the evaluation provide valuable insights into applying approximate techniques to large-scale data analysis. Nonetheless, there are potential optimizations possible which have not been discussed so far. Future work and optimizations can be separated into two aspects, first the library design and secondly the algorithm usage. From an engineering perspective, a deeper integration into Flink is the next step, potentially using Scala Implicit Conversion\(^1\) the functionality could be added directly to the DataStream type. Even more interesting is a direct integration with Flink’s Table API. The Table API can map Data Streams into a virtual representation of a table. This allows usage of SQL queries on the Stream. In an analysis within this thesis project, the current state of user-defined functions was deemed insufficient to implement approximate queries within the SQL dialect. Yet, once the API has improved to implement complex user-defined functions a direct integration would be possible. This could enable the usage of memory intensive standing queries within the Table API.

The second aspect targets the algorithm implementation. Within the prototype, the implementations of the respective algorithms, were taken from the data sketch library developed by Yahoo. The full implementation of the algorithms is out of scope in this thesis project. The high performance of the implementation is mainly based on using off-heap memory management and the inherent properties of the algorithms themselves. Said memory management is also an important characteristic of the Flink runtime implementation. As a result, it might be possible to improve the performance by natively implementing the approximation algorithms directly on the Flink runtime.

Finally, Apache Flink also supports a feature called “queryable state”. This means a Flink Function attached to a Stream can store information about its current state and make this state externally accessible. This could be used to externally access the Sketch information. For example through a restful interface. As a result, it would not be necessary to emit the results on a continuous stream but rather access solely the state.

\(^1\)Scala Implicit Conversion allows locally scoped runtime patching. This means a class can be extended with custom methods without adding methods to the global scope (compared to e.g. Ruby)\(^\text{[68]}\)
6 Conclusion

The work builds upon a continuous demand for Big Data processing solutions. While similar work has been done in the area of Batch Processing, no known implementations for Distributed Stream Processing systems exist so far. Within this work, the potential of approximate algorithms and customizable queries within Distributed Streaming environments is illustrated. The work includes (1) the design of a customizable query interface for standing queries, and (2) a prototype implementation on Apache Flink. Furthermore, the behavior and specifics of approximate algorithms within distributed streaming environments have been evaluated and analyzed.

Throughout the evaluation, the resource demand of the analyzed techniques stays consistently below the exact implementations. Nonetheless, the implementation can estimate results with a deviation below 10%. Yet, the work has also presented challenges resulting from varying data distribution. While the estimates are quite precise for the WikiTrace dataset, the Amazon Rating dataset exhibited varying results depending on the query. The implementation of approximate frequent items has shown its qualities noticeably. When choosing the correct parameters, the error can be easily controlled over a defined set of most frequent items. In the case of the HyperLogLog and the Theta sketch, the main hurdle is the grouping operation. Performance-wise these sketches work similarly well, yet without the additional group operation, their overall value is reduced. Within the evaluation, various characteristics of the prototype were highlighted. This includes the fast-processing, the accuracy, and the impact of varying data distribution and the error bounds. Based on this evaluation the potential to optimize the parameters are identified and presented. Overall, this work has highlighted the viability of approximate techniques within Distributed Stream Processing. It demonstrates both opportunities and challenges including options to mitigate said challenges.

Bibliography


