TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRICAL & COMPUTER ENGINEERING

DIPLOMA THESIS

Synopses over Streaming Data at Apache Flink

Author:
Gkoutziouli Dimitra

Thesis Committee:
Prof. Deligiannakis Antonios (Supervisor)
Prof. Garofalakis Minos
Prof. Samoladas Vasilis

A thesis submitted in partial fulfillment of the requirements for the degree of
Diploma in Electrical and Computer Engineering
October 10, 2019

Synopses over Streaming Data at Apache Flink

by Gkoutziouli Dimitra

Abstract

A growing number of applications demand algorithms and data structures that enable the efficient processing of data sets with gigabytes to terabytes to petabytes. Massive amounts of information that are generated continuously from numerous types of sources are called Big Data. Big Data is data that contains greater variety arriving in increasing volumes and with ever-higher velocity. Nowadays, many applications receive data in a streaming fashion way that must be processed on the fly as it arrives. Thus, the use of data structures called Synopses, is essential for managing such massive data, as handling large data sets is not often efficient to work fully on them. Synopses summarize the data set and provide approximate responses to queries.

One of the main families of synopses are sketches. A sketch of a large amount of data is a small data structure that is able to calculate or approximate certain characteristics of the original data. In this diploma thesis we focus on various streaming algorithms of sketches such as Bloom Filter, Count-Min, Flajolet-Martin and AMS sketches.

We propose a parallel implementation of query registration of the above sketches to be updated as more data arrives, insert dynamically new instances of these sketches in real-time execution and compute several functions. These functions may estimate the cardinality of the elements, the amount of distinct elements, or inform about the existence of an element in a stream. In order to develop that, we used the Apache Flink framework. Flink is a distributed streaming engine with high-throughput, low-latency and fault-tolerant computations over unbounded and bounded data streams.

First of all, we expound the theoretical background of the implemented algorithms and the distributed framework. Then, we explicate the implementation of the code as we use a Kafka connector, the transformations of Datastream API and finally the Queryable State feature of Flink. Through that method, users query the most up-to-date values of the sketches while other platforms can use this information as source.

Συνόψεις σε ροές δεδομένων στο Apache Flink

Γκουτζιούλη Δήμητρα

Περίληψη

Ένας μεγάλος αριθμός εφαρμογών απαιτεί αλγορίθμους και δομές δεδομένων που επεξεργάζονται αποδοτικά σύνολα δεδομένων που απαρτίζονται από gigabytes σε terabytes μέχρι petabytes. Τεράστιες ποσότητες πληροφοριών που παράγονται συνεχώς από διάφορες πηγές ονομάζονται Μεγάλα Δεδομένα. Τα Μεγάλα Δεδομένα είναι δεδομένα που περιέχουν τεράστια ποικιλία, φθάνουν σε αυξανόμενες ποσότητες και με μεγάλη ταχύτητα. Σήμερα, πολλές εφαρμογές λαμβάνουν δεδομένα σε συνεχείς ροές δεδομένων οι οποίες πρέπει να επεξεργάζονται αμέσως κατά την άφιξή τους. Έτσι, η χρήση δομών δεδομένων που ονομάζονται Συνόψεις, είναι απαραίτητη για τη διαχείριση πληθώρας δεδομένων, καθώς δεν είναι αποτελεσματική η μαζική διαχείρηση τους. Οι συνόψεις συνοψίζουν το σύνολο δεδομένων και παρέχουν προσεγγιστικές απαντήσεις σε επερωτήματα.

Μια από τις κύριες οικογένειες των συνόψεων είναι τα σκίτσα. Ένα σκίτσο μιας μεγάλης ποσότητας δεδομένων είναι μια μικρή δομή δεδομένων που μπορεί να υπολογίσει ή να προσεγγίσει ορισμένα χαρακτηριστικά των αρχικών δεδομένων. Στην παρούσα διπλωματική εργασία εστιάζουμε στους διάφορους αλγορίθμους ροών σκίτσων όπως το Bloom Filter, το Count-Min, το Flajolet-Martin και το AMS σκίτσο.

Προτείνουμε μια παράλληλη υλοποίηση της εγγραφής επερωτήσεων των παραπάνω σκίτσων τα οποία θα ενημερώνονται καθώς φτάνουν όλο και περισσότερα δεδομένα, θα εισάγονται δυναμικά νέα στιγμιότυπα των παραπάνω σκίτσων κατά την εκτέλεση σε πραγματικό χρόνο και θα υπολογίζονται διάφορες συναρτήσεις. Αυτές οι συναρτήσεις μπορούν να εκτιμήσουν το πλήθος των στοιχείων, την ποσότητα διακριτών στοιχείων ή να ενημερώσουν για την ύπαρξη ενός στοιχείου σε μία ροή. Για την υλοποίηση του παραπάνω εγχειρήματος, χρησιμοποιήσαμε την πλατφόρμα Apache Flink. Το Flink είναι ένας κατανεμημένος συνεχούς ροής μηχανισμός με υψηλό ρυθμό επεξεργασίας δεδομένων, χαμηλό χρόνο καθυστέρησης και αντοχή σε υπολογιστικά σφάλματα σε οριοθετημένες ή και μη ροές δεδομένων.

Αρχικά, παρουσιάζουμε το θεωρητικό υπόβαθρο των εφαρμοζόμενων αλγορίθμων και της κατανεμημένης πλατφόρμας. Στη συνέχεια, εξηγούμε την υλοποίηση του κώδικα καθώς χρησιμοποιούμε ως πηγή δεδομέων το Apache Kafka, τους μετασχηματισμούς της διεπαφής Datastream του Flink και τελικά, το χαρακτηριστικό Queryable State του Flink. Μέσω αυτής της μεθόδου, οι χρήστες ζητούν να μάθουν τις πιο ενημερωμένες τιμές των σκίτσων και επιπλέον αυτή η πληροφορία μπορεί να χρησιμοποιηθεί ως πηγή σε άλλες πλατφόρμες.

Acknowledgments

It's been five years of hard and exciting work until I graduate and I would like to seize the occasion to thank everyone who supported me in achieving my goals.

First and foremost, I would like to thank my advisor, Prof. Antonios Deligiannakis, who chiefly believed in me and provided me with feedback and support throughout this diploma thesis. In addition, I am grateful towards the other two committee members of my thesis, Prof. Minos Garofalakis and Prof. Vasilis Samoladas, for their useful comments and their time to evaluate this work.

Last but not least, I would like to thank my family for supporting me all these years and all my friends who tolerated me and they were there for me when I needed them most.

Contents

Chapter 1

Introduction

A growing number of applications demand algorithms and data structures that enable the efficient processing of data sets with gigabytes to terabytes to petabytes. Massive amounts of information that are generated continuously from numerous types of sources (e.g. financial transactions, sensor networks) are called Big Data. Big Data is a term that describes the large volume of either structured or unstructured data that inundates a business on daily. Big Data is data that contains greater variety arriving in increasing volumes and with ever-higher velocity. This is known as the three Vs. When Big Data combine with high-powered analytics can lead to better managing decisions and strategic business moves. Also in many applications, (e.g. network traffic monitoring, query at a search engine) data arrives in a streaming fashion way and must be processed on the fly.

The data structures used by the algorithms to represent the input data stream is merely a summary, as known as synopses. The use of synopses is essential for managing the massive data that arises daily in business as handling such large data sets it is not often efficient to work fully on them. Instead, it is much more convenient to build a synopsis, and then use this synopsis to process the data. Synopses summarize the data set and provide approximate responses to queries. A synopsis data structure may resides in main memory providing for fast processing of queries and of data structure updates by avoiding disk accesses altogether or may resides on the disks and can be swapped in and out of memory with minimal disk accesses. Moreover, a synopsis data structure leaves space in the memory for other data structures and more importantly it leaves space for other processing. So, a synopsis data structure has a minimal impact on the overall cost of the system. Due to its importance in applications there are a number of synopsis data structures in the literature and in existing systems. The four main families of synopses are random samples, histograms, wavelets, and sketches.

In this diploma thesis we focus on various streaming algorithms of sketches such as Bloom Filters, Count-Min, Flajolet-Martin and AMS sketches. We typically implement a query registration of the above sketches to be updated as more data arrives, insert new instances of these sketches in real-time execution and compute several functions. These functions may estimate the cardinality of the elements, the amount of distinct elements, or inform about the existence of an element in a stream.

In order to propose a scalable parallel implementation of synopses, we use Apache Flink framework. Apache Flink is an open source framework and distributed processing engine for large scale computations over unbounded and bounded data streams. Flink provides APIs for both Stream and Batch processing, and libraries for relational queries, complex event processing scenarios, graph processing and machine learning. In Flink, programs can be written in Java,

Scala, Python and SQL, and can be deployed in local, cluster or cloud mode.

1.1 Thesis Outline

In Chapter 1 we talk about Big Data and their semantic use in a business on a day-to-day basis. Then, we suggest Synopses, a flexible data structure, for managing the massive data that arise in modern information management scenarios.

In Chapter 2 we set the definition of streaming algorithms and their use cases. Then, in section 2.1 we present the essential use of sketches as they have been successfully applied to web data compression, approximate query processing in databases, network measurement and signal processing/acquisition. In the following sections we analyze commonly used types of sketches, specifically we set the parameters of each technique, we exhibit how each algorithm works and finally we list their advantages and the scenarios of using them. In section 2.2, 2.3, 2.4, 2.5 we make known the Count-Min, Bloom Filter, Flajolet Martin and AMS sketch respectively.

In Chapter 3 we give an introduction of Apache Flink framework. In section 3.1 we perform the dataflow programming model of streams and we emphasize on Flink's characteristics such as Time, Windows and Watermarks. In section 3.2 we describe how Flink programs are executed in the distributed runtime environment. In section 3.3 we refer to DataStream API for handling unbounded and bounded streams and in subsections 3.3.1 and 3.3.2 we elaborate the transformations that are applied in data streams and we report on stateful operators and the fault tolerance mechanism, correspondingly. In section 3.4 we expand Flink's connectors used as data sources and sinks and moreover in subsection 3.4.1 we detail the Kafka connector. Finally, in section 3.5 we focus on the features let Flink to have a wide acceptance in real-time analytics and applications.

In Chapter 4 we propose a distributed implementation of query registration in Apache Flink. For the development phase, we use the DataStream API in Java to apply transformations on unbounded data streams. In section 4.1 we analyze the use of Apache Kafka Connector as source of data that our program manages. In section 4.2 we explain thoroughly the distributed implementation of query registration. At last, in section 4.3 we point out the use of stateful function and operator to store the necessary information while in subsection 4.3.1 we exploit the Queryable State Feature of Flink to allow the user to query a job's state from outside Flink's runtime environment.

In Chapter 5 we conduct several experiments to evaluate the performance of the parallel implementation. In the first set of experiments, we discuss about the emerging results that we ran locally. In the second set of experiments, we conduct experiments with different levels of parallelism to evaluate the runtime and the throughput of our implementation.

In Chapter 6 we deduce the thesis by presenting the main conclusions, and suggest potential directions for future work.

Chapter 2

Sketches and Streaming Algorithms

In computer science, one technique to analyze Big Data is using streaming algorithms that several times they have access to limited memory. Streaming algorithms [14] are algorithms for processing data streams in only a few passes - typically just one- as the input is presented as a sequence of items. Thereat, a streaming algorithm may produces an approximate answer based on a summary or "sketch" of the data stream in memory. Streaming algorithms have several applications in networking such as monitoring network links for huge flows, counting the number of distinct values, clustering summary, estimating the distribution of flow sizes, getting insights of interest and so on.

2.1 Sketches

Sketch techniques [5] have become very popular over the past few years. A sketch of a large amount of data is a small data structure that is able to calculate or approximate certain characteristics of the original data. There are lots of types of sketches and the choice using one of them depends on what the use tries to approximate and also on the nature of the data. Sketches are mainly suitable for streaming data, in which massive quantities of data flow by and the sketch summary must continually be updated quickly and compactly. These sketches are much shorter, often exponentially, than the original data, however they retain crucial and useful information, such as the number of distinct elements in the data set, the similarity between the data elements etc.

The methods have been successfully applied to web data compression, approximate query processing in databases, network measurement and signal processing/acquisition. Commonly used data sketches include k-minimum value, hyper-log-log summaries, Bloom Filters, Count-Min, Flajolet-Martin, AMS, dp-means or k-means clusters and the t-digest. Below we expand the sketches that are used to our implementation for query registration.

2.2 Count-Min

The Count-Min sketch (CM sketch) is a sub-linear space data structure, introduced by Muthukrishnan and Cormode in 2003 [4] and since then has been used in many applications and has led to many extensions and variations. CM sketch summarizes large amounts of frequency data based on probabilistic algorithms to estimate several types of queries on streaming data. This sketch allows fundamental queries in data stream summary such as point, range, and inner

product queries to be approximately answered very quickly. In addition, it can be applied to solve several important problems in data streams such as finding quantiles, frequent items, etc.

This technique uses multiple hash functions, one for each column, to map events to frequencies, but unlike a hash table uses only sub-linear space. Although, there's always a possibility of overcounting some events due to collisions. First of all, every cell in the CM sketch is initialized to zero. When an event occurs, the event's id is hashed over every column and each hash function outputs a row value. As a result the counter at each resulting row-column combination increases by one. In order to query an event, we take the minimum value of the event's counter among all the hash functions, as it is the closest candidate to give the correct result for the query.

What is more, we model the data stream as a vector a[1 .. n] and the updates received at time t are of the form (i_t, c_t) which mean that the element a[i_t] has been incremented by c_t. The core of the data structure is a two dimensional array count[w,d] that stores the synopsis of the original vector and which is used to report approximate results of queries. Hence the total space requirement of the data structure is (w × d). Therefore, we have d pairwise-independent hash functions h_1 .. h_d that hash each of the inputs to the range (1 .. w). When an update (i_t, c_t) comes for the stream, we hash a[i_t] through each of the hash functions h_1 ... h_d and increment each of the w entries in the array that they hash to (Figure 2.1 [4]). For the purpose of getting the approximate value of an element a[i] of the vector a, it is computed the minimum of all values in each of the d cells of count where i hashes to.

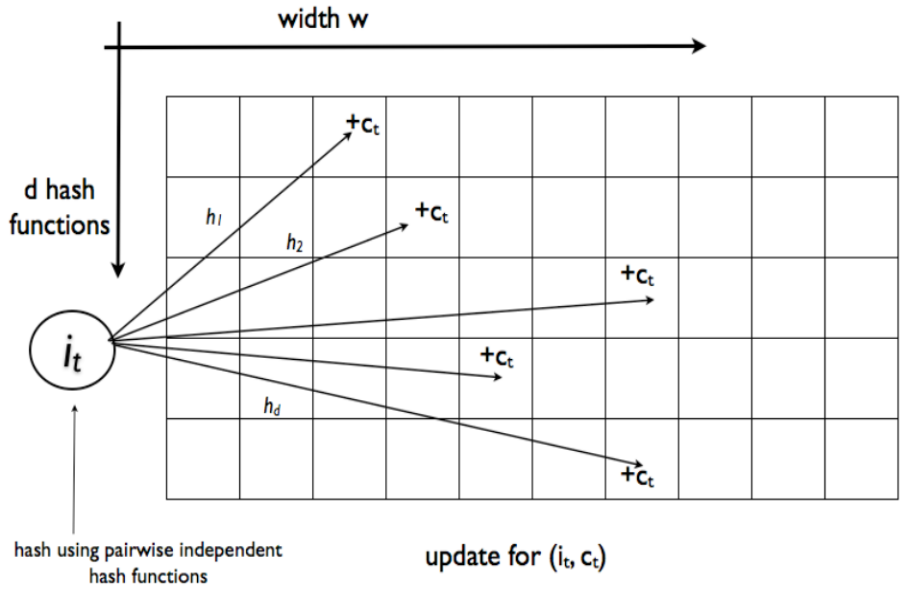


Figure 2.1: The processing of the updates in a Count-Min sketch

Working on sub-linear space implies that we do not get to store or see all data so we do not get to deliver an accurate answer to some queries. We rely on some approximation techniques that deliver an accuracy with a reasonably high probability bound. The data structure is parameterized by two factors ϵ and δ, where the error in answering the query is within a factor of ϵ with probability δ. So these parameters can be modified based on the space that is available and accordingly the accuracy of results that the data structure serves.

2.3 Bloom Filter

Bloom Filter is a space-efficient probabilistic data structure, introduced by Burton Howard Bloom in 1970 [2], that is used to test whether an element is a member of a set. The answer is that an element either definitely is not in the set or may be in the set. So, false positive matches are possible, but false negatives are not. Elements can be added to the set, but not removed and the more elements that are added to the set, the larger the probability of false positives.

The base data structure of a Bloom Filter is a Bit Vector. First of all, all the m bits of the bit array are set to zero. There are also k different hash functions, each of them maps an element to one of the m bit positions. The more hash functions are used, the slower the Bloom Filter is, and the quicker it fills up. However, if there are too few it may suffer too many false positives. In order to add an element, it must be fed to the hash functions to get k bit positions, and set the bits at these positions to 1. To test if an element is in the set, it must be fed to the hash functions to get k bit positions. If any of the bits at these positions is 0, the element definitely does not exist in the set. If all are 1, then it is possible that the element exists in the set.

In figure 2.2 [15] is illustrated a Bloom Filter with three elements 'x','y' and 'z'. It consists of 18 bits and uses 3 hash functions. In this example, the colored arrows point to the bits that the elements of the set are mapped to. In coclusion, the element 'w' definitely is not in the set, since it hashes to a bit position containing 0.

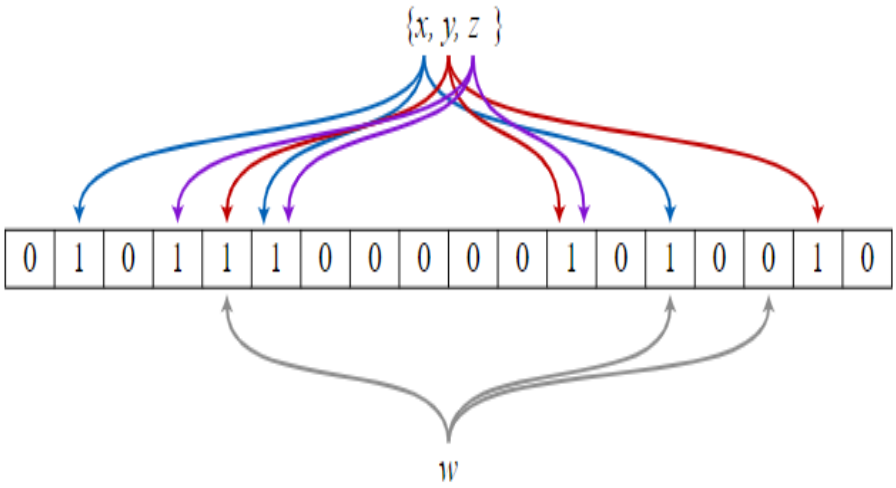


Figure 2.2: Bloom Filter testing if the element 'w' is member of the set

Wherefore a Bloom Filter is based on hash functions, requires much less space than the actual size of the items that must be stored and checked. It has constant time complexity for both adding elements and asking if they exist in the set. For these hash functions, collisions in the outputs do not really matter too much, as long as they are reasonably rare. It is more important for the outputs to be evenly and randomly distributed and, of course, it is desirable the hash functions to be fast. Content distribution networks use them to avoid caching one-hit wonders, files that are seen only once. Web browsers use them to check for potentially harmful URLs.

2.4 Flajolet Martin

The Flajolet–Martin algorithm was introduced by Philippe Flajolet and G. Nigel Martin in their 1984 article "Probabilistic Counting Algorithms for Data Base Applications" [8]. It is about an algorithm that approximates the number of distinct elements in a stream with a single pass, as known as the count-distinct problem. It uses space-consumption logarithmic in the maximal number of possible distinct elements in the stream.

Suppose that there are random hash function that act on strings and generate integers. It is expected that:

- 1/2 of them have their binary representation end in 0 (i.e. divisible by 2),
- 1/4 of them have their binary representation end in 00 (i.e. divisible by 4)
- 1/8 of them have their binary representation end in 000 (i.e. divisible by 8)
- and in general, $1/2^n$ of them have their binary representation end in 0^n.

If the hash function generated an integer ending in 0^m bits (and it also generated integers ending in 0^{m-1} bits, 0^{m-2} bits, ..., 0^1 bits), intuitively, the number of unique strings is around 2^m. This algorithm maintains 1 bit for each 0^i seen. The output of the algorithm is based on the maximum of consecutive 0^i seen.

Here is a simple explanation of how the algorithm works [1]. Firstly, is creates a bit vector (bit array) of sufficient length L, such that 2^L >n, the number of elements in the stream. Usually a 64-bit vector is sufficient since 2^{64} is quite large for most purposes. The i-th bit in this vector represents whether we have seen a hash function value whose binary representation ends in 0^i. So we initialize each bit to 0. Then, a random hash function is generated and maps input, usually strings, to natural numbers. Each word of the input, it is hashed and determined the number of trailing zeros. If the number of trailing zeros is k, the k-th bit is set to 1 in the bit vector. Once input is exhausted, the index of the first 0 in the bit array, called R. Then, the number of unique words is calculated as $2^R/\phi$, where ϕ equals to 0.77351. A proof for this can be found in the original paper listed in the reference section. The standard deviation of R is a constant: σ(R)=1.12. This implies that our count can be off by a factor of 2 for 32% of the observations, off by a factory of 4 for 5% of the observations, off by a factor of 8 for 0.3% of the observations and so on. In figure 2.3 [1] is illustrated an example of using the FM algorithm for the stream S= 1,3,2,1,2,3,4,3,1,2,3,1. The hash function that is used is $h(x) = (6x + 1) \mod 5$ and it is assumed that the absolute value of the bits equals to five. So, the results are that R = max(r(a)) = 5 and the number of distinct elements equals to N=2^R=2^5=32.

x	h(x)	Rem	Binary	r(a)
1	7	2	00010	1
3	19	4	00100	2
2	13	3	00011	0
1	7	2	00010	1
2	13	3	00011	0
3	19	4	00100	2
4	25	0	00000	5
3	19	4	00100	2
1	7	2	00010	1
2	13	3	00011	0
3	19	4	00100	2
1	7	2	00010	1

Figure 2.3: Count the number of distinct elements using the FM algorithm

To improve the accuracy of the FM algorithm, we apply the Averaging method by using multiple hash functions and using the average R instead. Another strategy is to apply the Bucketing Averages that are susceptible to large fluctuations. So using multiple buckets of hash functions from the above step and using the median of the average R, giving fairly good accuracy. In conclusion, accuracy of this algorithm can be tuned by using appropriate number of hash functions in the averaging and bucketing steps. The more hash functions are used the higher accuracy is achieved, but implies higher computation cost.

The algorithm approximates the number of unique elements, along with a standard deviation σ, which can then be used to determine bounds on the approximation with a desired maximum error ϵ, if needed. If the stream contains n elements with m of them unique, this algorithm runs in O(n) time and needs O(log(m)) memory, on the contrary with the brute-force algorithm that needs O(m) memory. It was, also, observed that this algorithm is quite sensitive to the hash function parameters. In 2007 the HyperLogLog algorithm splits the multiset into subsets and estimates their cardinalities, then it uses the harmonic mean to combine them into an estimate for the original cardinality.

2.5 AMS

The AMS sketch was introduced in 1996 by Noga Alon, Yossi Matias, and Mario Szegedy [13] and is used to approximate computation of frequency moments. It was proposed to estimate the value of F_2 of the frequency vector, the sum of the squares entries of a vector defined by a stream of updates. This quantity is naturally related to the Euclidean norm of the vector, and so has many applications in high-dimensional geometry, in data mining and machine learning or anything else that use vector representations of data. The data structure maintains a linear projection of the stream, modeled as a vector, with a number of randomly chosen vectors that are defined implicitly by simple hash functions, and so do not have to be stored explicitly. The

accuracy of the estimating result depends according to sketch's size. The summary is a linear projection and so it can be updated flexibly and also the sketches can be combined by addition or subtraction. Most directly, F_2 equates to the self-join size of the relation whose frequency distribution on the join attribute is f (for an equi-join). The algorithm works as follows:

- Pick m random hash function h_1, h_2, ..., h_m from a 4-wise independent hash family $H = \{h : [n] \to \{\frac{-1}{\sqrt{m}}, \frac{+1}{\sqrt{m}}\}\}$
- Let $A_{ij} = h_i(j)$, and compute $y_i = \sum_j A_{ij} x_j$, for all i = 1, 2, .., m.
- Output $\sum_i {y_i}^2$ which is essentially $||Ax||_2^2$.

Now compute the mean and variance of $\sum_i {y_i}^2$.

$\mathrm{E}[\sum_i {y_i}^2] = m\,\mathrm{E}[{y_i}^2] = ... = ||x||_2^2 = F_2$.

$\mathrm{Var}(\sum_i {y_i}^2) = \mathrm{Var}(||Ax||_2^2) = ... \leq 2F_2^2 = O(||x||_4^2/m)$

Using Chebyshev's inequality gives the JL guarantee

$P(||Ax||_2^2 - ||x||_2^2 \geq \epsilon ||x||_2^2) \leq \frac{Var(\sum_i {y_i}^2)}{\epsilon^2 ||x||_4^2} = O(\frac{1}{\epsilon^2 m}) \leq \frac{1}{4}$

where the last inequality holds for $m = O(\frac{1}{\epsilon^2})$. Via the Chebyshev and Chernoff bounds, we get a value that is within a factor $(1 + \epsilon)$ of F_2 with probability at least $1 - \delta$.

There are two ways to get a high-probability bound with dependence $\log \frac{1}{\delta}$. The first method is to use the "median of mean" trick and the second one is to use higher moments' bound.

The AMS sketch can also be applied to estimate the inner-product between a pair of vectors. This use of the summary to estimate the inner product of vectors was described in a follow-up work by Alon, Matias, Gibbons and Szegedy, and the analysis was similarly generalized to the fast version by Cormode and Garofalakis. The ability to capture norms and inner products in Euclidean space means that the AMS sketch turned out to be highly flexible. It is at the heart of estimation techniques for a variety of other problems which are all of direct relevance to Approximate Query Processing as well.

Chapter 3

Apache Flink

Apache Flink [10] is an open source framework and distributed processing engine for stateful computations over unbounded and bounded data streams that can perform computations at in-memory speed and at any scale. Flink was formerly known as "Stratosphere", a research project conducted by three universities in Berlin. In December 2014, it was accepted as an Apache top-level project. Flink provides multiple APIs at different levels of abstraction and offers dedicated libraries for common use cases. Apache Flink is the next generation Big Data tool also known as 4G of Big Data. Flink's programs can be written in Java, Scala, Python and SQL, and can be deployed in local, cluster or cloud mode. Flink applications can process recorded or real-time streams. Apache Flink is a distributed system and requires compute resources in order to execute applications. Flink integrates with all common cluster resource managers such as Hadoop YARN, Apache Mesos, and Kubernetes but can also be setup to run as a stand-alone cluster. Flink's architecture illustrated in Figure 3.1 [3]

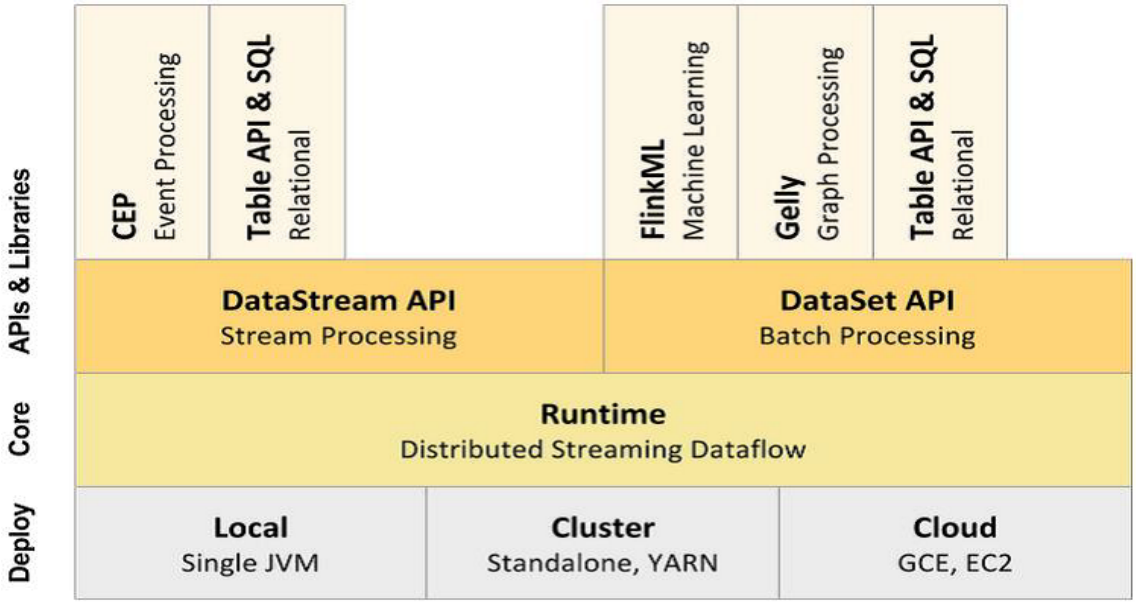


Figure 3.1: Architecture of Apache Flink

Flink provides three layered APIs (Figure 3.2 [16]). Each API offers a different trade-off between conciseness and expressiveness and targets different use cases.

ProcessFunctions are the most expressive function interfaces that Flink offers. Flink pro-

vides ProcessFunctions to process individual events from one or two input streams or events that were grouped in a window. ProcessFunctions provide fine-grained control over time and state.

The DataStream API, which is thoroughly analyzed in section 3.3, provides primitives for many common stream processing operations, such as windowing, record-at-a-time transformations, and enriching events by querying an external data store.

Flink features two relational APIs, the Table API and SQL. Both APIs are unified APIs for batch and stream processing. The Table API and SQL leverage Apache Calcite for parsing, validation, and query optimization. They can be seamlessly integrated with the DataStream and DataSet APIs and support user-defined scalar, aggregate, and table-valued functions.

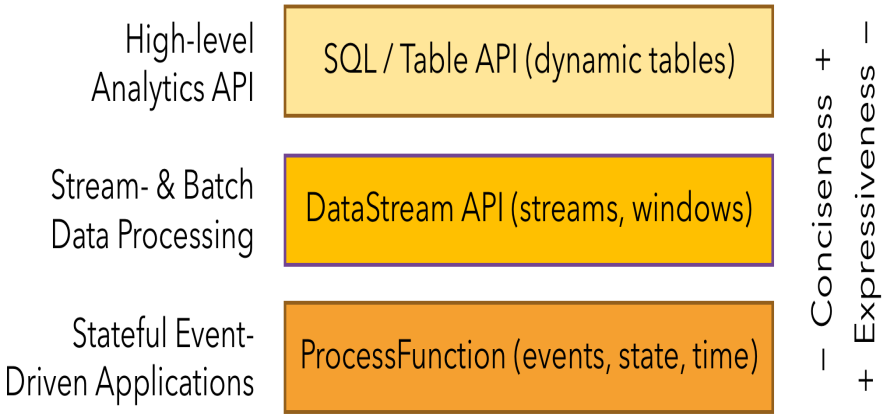


Figure 3.2: Layered APIs of Apache Flink

Moreover, Flink features several libraries for common data processing use cases. The libraries are typically embedded in an API and not fully self-contained and they are developing below:

- Complex Event Processing (CEP). Flink's CEP library provides an API to specify patterns of events and the CEP library is integrated with Flink's DataStream API. Applications for the CEP library include network intrusion detection, business process monitoring and fraud detection.

- DataSet API. The DataSet API is Flink's core API for batch processing applications. The primitives of the DataSet API include map, reduce, (outer) join, co-group, and iterate. All operations are backed by algorithms and data structures that operate on serialized data in memory and spill to disk if the data size exceed the memory budget.

- Table API & SQL. The Table API is a language-integrated query API for Scala and Java that allows the composition of queries from relational operators such as selection, filter, and join in a very intuitive way. Flink's SQL support is based on Apache Calcite which implements the SQL standard.

- Gelly. Gelly is a library for scalable graph processing and analysis. Gelly is implemented on top of and integrated with the DataSet API. Hence, it benefits from its scalable and robust operators. Gelly features built-in algorithms, such as label propagation, triangle enumeration, and page rank, but provides also a Graph API that eases the implementation of custom graph algorithms.

- FlinkML. FlinkML is the Machine Learning (ML) library for Flink. It supports many algorithms such as supervised learning and unsupervised learning, recommendation, data pre-

processing techniques and more. FlinkML has a feature called ML-pipelines, which provides the ability to chain different transformers and predictors in a type-safe manner.

Apache Flink is the powerful open source platform which can address various types of requirements efficiently such as Batch Processing, Interactive processing, Real-time stream processing, Graph Processing, Iterative Processing and In-memory processing. In the following sectors, we focus on handling streaming data, so we explicate the dataflow of Apache Flink, the distributed runtime environment and the operators of the DataStream API.

3.1 Dataflow Programming Model

The basic building blocks of Flink programs are streams and transformations. Conceptually a stream is a never-ending flow of data records and a transformation is an operation that takes one or more streams as input and produces one or more output streams as a result. When executed, Flink programs are mapped to streaming dataflows, consisting of streams and transformation operators. Each dataflow starts with one or more sources and ends in one or more sinks. The source defines where the input data comes from (e.g Apache Kafka). The sink defines where the output result is stored (e.g Apache Cassandra). An operator applies transformations into streams (e.g Map) and the resulted streams produced by data sources, sinks or operators. The dataflows resemble arbitrary directed acyclic graphs (DAGs) as this one in Figure 3.3.

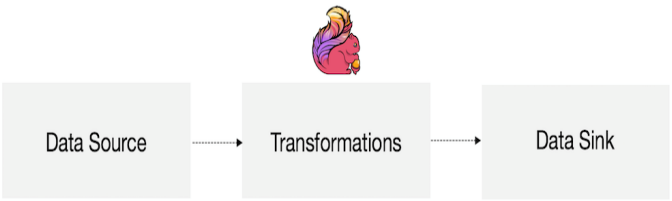


Figure 3.3: A directed acyclic graph of a dataflow

For example, in Figure 3.4 [6] we see the streaming dataflow of a Flink program written in the DataStream API. In the beginning, the program uses a data source connector to consume data from a topic of Apache Kafka. Then, a Map operator transforms the initial data stream of strings to events, by using a parse function. The next operator groups by the data stream according to the key "id," and then applies every 10 seconds an aggregation function to the events with the same key. Finally, a data sink is used to store the results of the aggregation function to rolling files in the system.

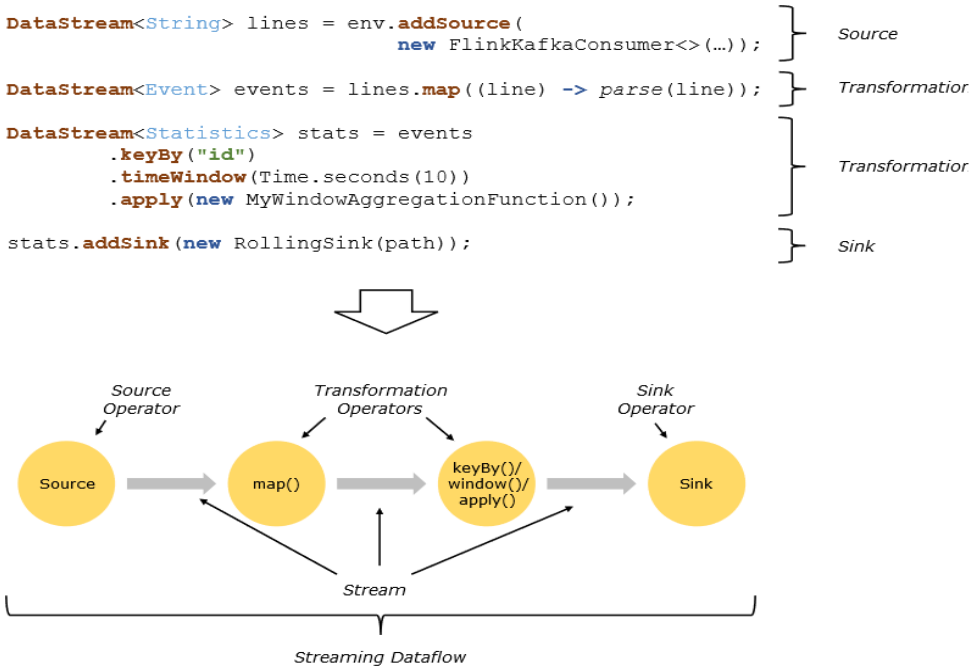


Figure 3.4: Example of a streaming dataflow

Programs in Flink are inherently parallel and distributed. During execution, a stream has one or more stream partitions and each operator has one or more operator subtasks. The operator subtasks are independent of one another and execute in different threads and possibly on different machines or containers. The number of operator subtasks is the parallelism of that particular operator. The parallelism of a stream is always that of its producing operator. Different operators of the same program may have different levels of parallelism. In Figure 3.5 [6] it is shown the parallelism view of the previous example.

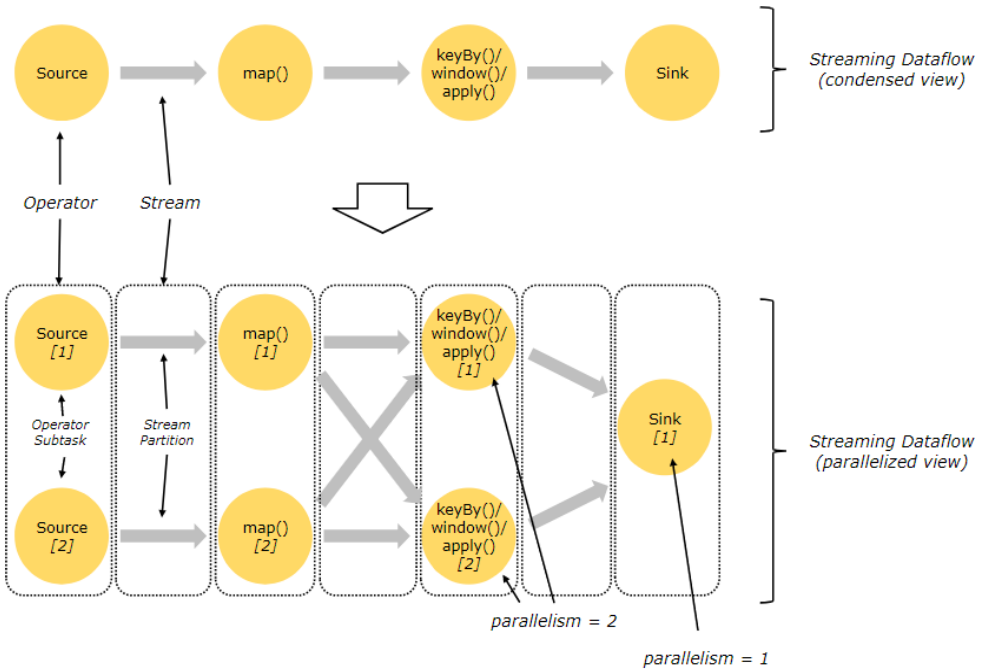


Figure 3.5: Example of a parallel streaming dataflow

Notion of Time

An important aspect of streaming applications is the measurement of time. Flink supports three different notions of time:

i Event time is the time when an event was created and it is usually described by an event timestamp. Event time programs must specify how to generate Event Time Watermarks, which is the mechanism that signals progress in event time.

ii Processing time refers to the system time of the machine that is executing the respective time-based operation. When a streaming program runs on processing time, all time-based operations (like time windows) will use the system clock of the machines that run the respective operator.

iii Ingestion time is the time when an event enters the Flink. At the source operator each record gets the source's current time as a timestamp, and time-based operations (like time windows) refer to that timestamp. Ingestion time sits conceptually in between event time and processing time. Compared to processing time, it is slightly more expensive, but gives more predictable results.

Windows

Windows split the stream into "buckets" of finite size, over which we can apply computations. There are the Keyed Windows for the keyed streams using the keyBy operator and the Non-Keyed Windows for the non-keyed ones. A window is created as soon as the first element that should belong to this window arrives, and the window is completely removed when the time (event or processing time) passes its end timestamp plus the user-specified allowed lateness. There is, also, the window assigner which defines how elements are assigned to windows such as tumbling windows, sliding windows, session windows and global windows.

Watermarks

The mechanism of Flink to measure progress in time-based operations is called watermarks. Watermarks flow as part of the parallel streaming dataflow along with stream events and carry a timestamp t. A Watermark(t) declares that event time has reached time t in that stream, meaning that there should be no more elements from the stream with a timestamp t' $\leq$ t. When a subtask of an operator receives a watermark, it advances its internal clock according to the watermark's timestamp. Watermarks are crucial for out-of-order streams where the events are not ordered by their timestamps. In general a watermark is a declaration that by that point in the stream, all events up to a certain timestamp should have arrived.

3.2 Distributed Runtime Environment

For distributed execution, Flink chains operator subtasks together into tasks. Each task is executed by one thread. Chaining operators together into tasks is a useful optimization as it reduces the overhead of thread-to-thread handover and buffering, and increases overall throughput while decreasing latency. The sample dataflow in the Figure 3.5 is executed with five subtasks, and hence with five parallel threads.

There are two types of processes that consists of the Flink's runtime illustrated in Figure 3.6 [7]. Firstly, there are the **JobManagers**, also called masters, which coordinate the distributed execution. They schedule tasks, coordinate checkpoints, coordinate recovery on failures, etc. There is always at least one Job Manager. A high-availability setup will have multiple JobManagers, one of which is always the leader, and the others are standby. Secondly, there are the **TaskManagers**, also called workers, which execute the tasks (or more specifically, the subtasks) of a dataflow, and buffer and exchange the data streams. Same again there must always be at least one TaskManager. Each Task Manager is a separate JVM process and it is composed by a number of task slots.

They either can be started directly on the machines as a standalone cluster, in containers, or managed by resource frameworks like YARN or Mesos. TaskManagers connect to JobManagers, announcing themselves as available and are assigned work.

There is also the **Job Client** which is the starting point of the program execution and it is not a part of the runtime. The job client creates and sends the streaming dataflow to the Job Manager for the execution in the distributed environment. During the execution, the client receives statistics and results from the Job Manager.

3.3 DataStream API

DataStream is the core API for handling unbounded and bounded streams. This API provides many common stream processing operators which we will elaborate on the subsection 3.3.1. DataStream API [9] supports two types of broadcast streams, streams that are broadcasted to the downstream parallel subtasks of an operator and streams that are available among the parallel subtasks. All of the above operators can be stateful and fault tolerant with the appropriate use of state data structures. Flink has a feature called Queryable State that allows the user to query the state from outside of the distributed environment. Furthermore, DataStream API is compatible with Apache Storm and therefore allows the reuse of Storm code (e.g Storm topologies, Spouts & Bolts).

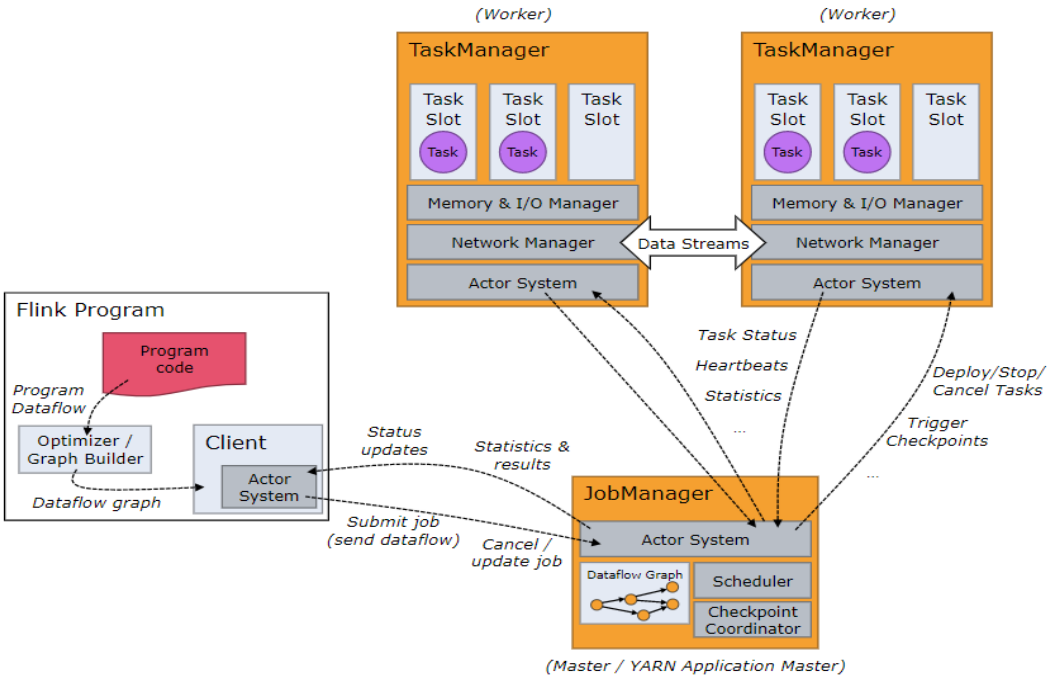


Figure 3.6: Distributed Architecture of the runtime environment

3.3.1 DataStream Transformations

In this subsection we describe the anatomy of DataStream API programs along with specific operators that we used in the implementation of the code. Data streams are represented by special classes (e.g DataStream<T>, KeyedStream<T, KEY>), which are immutable collections of data.

The first step in a Flink program is to create a StreamExecutionEnvironment. It is able to obtain automatically obtain the execution environment from the getExecutionEnvironment() function.

```
1        StreamExecutionEnvironment env = StreamExecutionEnvironment.
             getExecutionEnvironment();
```

Next a source is created that reads from a Kafka topic or a text file:

```
1        DataStream<T> data = env.addSource(new FlinkKafkaConsumer08<>(...));
2        DataStream<String> text = env.readTextFile("file:///path/to/file");
```

Then, there are plenty of transformations which programs can combine into sophisticated dataflow topologies. We present those we use in our implementation.

Map Transformation: Takes one element (Datastream) and produces one element (Datastream). Here is an example of a map function that doubles the values of the input stream.

```
1        DataStream<Integer> dataStream = //...
2        dataStream.map(new MapFunction<Integer, Integer>() {
3              @Override
4              public Integer map(Integer value) throws Exception {
5                  return 2 * value;
```

```
        }
    });
```

FlatMap Transformation: Takes one element (Datastream) and produces zero, one, or more elements (Datastream). Here is an example of flatmap function that splits sentences to words

```
    dataStream.flatMap(new FlatMapFunction<String, String>() {
        @Override
        public void flatMap(String value, Collector<String> out)
            throws Exception {
            for(String word: value.split(" ")){
                out.collect(word);
            }
        }
    });
```

KeyBy Transformation: Logically partitions a stream into disjoint partitions. All records with the same key are assigned to the same partition. Internally, keyBy() is implemented with hash partitioning. There are different ways to specify keys. This transformation returns a KeyedStream, which is, among other things, required to use keyed state.

```
    dataStream.keyBy("someKey") // Key by field "someKey"
    dataStream.keyBy(0) // Key by the first element of a Tuple
```

Connect Transformation: "Connects" two data streams retaining their types. Connect allowing for shared state between the two streams.

```
    DataStream<Integer> someStream = //...
    DataStream<String> otherStream = //...
    ConnectedStreams<Integer, String> connectedStreams = someStream.connect(
        otherStream);
```

CoFlatMap Transformation: Similar to flatMap on a connected data stream.

```
    connectedStreams.flatMap(new CoFlatMapFunction<Integer, String, String
        >() {

        @Override
        public void flatMap1(Integer value, Collector<String> out) {
            out.collect(value.toString());
        }

        @Override
        public void flatMap2(String value, Collector<String> out) {
            for (String word: value.split(" ")) {
              out.collect(word);
            }
        }
    });
```

ProcessFunction: Transforms a DataStream or a KeyedStream given a ProccessFunction. The ProcessFunction can be thought of as a FlatMapFunction with access to keyed state and timers.

```
1        stream.keyBy(...).process(new MyProcessFunction())
```

Next, the output results must be stored to an outside system by creating a sink. DataStream API has a variety of data sink functions.

```
1        writeAsText(String path)
2        print()
```

Finally, when the program is completed must trigger the program execution by calling execute() on the StreamExecutionEnvironment.

```
1        env.execute(Job name);
```

3.3.2 State & Fault Tolerance

Streaming applications often require data structures to store intermediate results of their computations. Stateful functions and operators store data across the processing of individual elements, and the information that each operator remembers is called state. Flink provides in-core data structures for stateful operations, that are scoped per parallel subtask (e.g figure 3.5 Map[1]) or per key attributes from the data records (e.g 3.4 keyBy("id")). Flink provides different state backends that specify how and where state is stored. State can be located on Java's heap or off-heap. Depending on the state backend, Flink can also manage the state for the application.

There are two basic kinds of state in Flink: **Keyed State** and **Operator State**. Keyed State is always relative to keys and can only be used in functions and operators on a KeyedStream. With Operator State (or non-keyed state), each operator state is bound to one parallel operator instance. Keyed State and Operator State exist in two forms: **managed** and **raw**. Managed State is represented in data structures controlled by the Flink runtime, in contrast Raw State is state that operators keep in their own data structures and when checkpointed, they only write a sequence of bytes into the checkpoint so Flink knows nothing about the state's data structures and sees only the raw bytes.

We used managed keyed state in our implementation and more specifically we used the ValueState<T>which is a type of state that keeps a value that can be updated and retrieved. To get a state handle, we had to create a StateDescriptor. This holds the name of the state the type of the values that the state holds, and possibly a user-specified function. In our case we used the ValueStateDescriptor. Although, state is accessed using the RuntimeContext, so it is only possible in rich functions in order to have access in the four methods: open, close, getRuntimeContext, and setRuntimeContext. The RuntimeContext that is available in a RichFunction use the method **getState(ValueStateDescriptor<T>)** for accessing state.

The queryable state feature of Flink allows you to access state from outside of Flink during runtime. The Queryable State feature consists of three main entities:

1 The QueryableStateClient, which runs outside the Flink cluster and submits the user queries.

2 The QueryableStateClientProxy, which runs on each TaskManager and is responsible for receiving the client's queries, fetching the requested state from the responsible Task Manager on his behalf, and returning it to the client.

3 The QueryableStateServer which runs on each TaskManager and is responsible for serving the locally stored state.

The client connects to one of the proxies and sends a request for the state associated with a specific key, k. The only requirement to initialize the client is to provide a valid TaskManager hostname and the port where the proxy listens. In order for a state to be visible to the outside world, it needs to be explicitly made queryable by using either a QueryableStateStream or the stateDescriptor.setQueryable(String queryableStateName) method.

Apache Flink offers a fault tolerance mechanism to consistently recover the state of data streaming applications. The mechanism ensures that even in the presence of failures, the program's state will eventually reflect every record from the data stream exactly once. The fault tolerance mechanism continuously draws consistent snapshots of the distributed data stream and operator state. These snapshots act as consistent checkpoints to which the system can fall back in case of a failure. For streaming applications with small state, these snapshots are very light-weight and can be drawn frequently without much impact on performance.

In case of a program failure (due to machine or network or software failure), Flink stops the distributed streaming dataflow. The system then restarts the operators and resets them to the latest successful checkpoint. The input streams are reset to the point of the state snapshot. Any records that are processed as part of the restarted parallel dataflow are guaranteed to not have been part of the previously checkpointed state.

3.4 Connectors

Connectors provide code for interfacing with various third-party systems. Some of the supporting connectors of Flink are: Apache Kafka (source/sink), Apache Cassandra (sink), Amazon Kinesis Streams (source/sink), Elasticsearch (sink), Hadoop FileSystem (sink). In our implementation we use a Kafka Connector in order to read from Kafka topics the data inputs and the requests. Apache Kafka is a distributed, high-throughput message queuing system designed for making streaming data available to multiple data consumers. The Flink Kafka Consumer integrates with Flink's checkpointing mechanism to provide exactly-once processing semantics.

3.4.1 Apache Kafka Connector

Apache Kafka [11] makes the streaming data durable by persisting incoming messages on disk using a log data structure. Typical installations of Flink and Kafka start with event streams being pushed to Kafka, which are then consumed by Flink jobs. Flink integrates with Kafka in a way that guarantees exactly-once delivery of events, does not create problems due to backpressure, has high throughput, and is easy to use for application developers.

What is more, in Kafka there are topics. A topic is a handle to a logical stream of data, consisting of many partitions. Partitions are subsets of the data served by the topic that reside in different physical nodes. Services that put data into a topic are called producers and a service that reads data from a topic is called a consumer. Moreover, there is the Kafka broker, a service that is installed on the node that contains the partition and allows consumers and producers to access the data of a topic. Each message within a partition is assigned with a unique id, called "message offset", which represents a unique, increasing logical timestamp within a partition. This offset allows consumers to request messages from a certain offset.

When the user creates a Kafka topic, has to specify the number of partitions in order to be assigned to Flink's parallel task instances. When there are more Flink tasks than Kafka partitions, some of the Flink consumers will just idle and they won't read any data. Although when there are more Kafka partitions than Flink tasks, Flink consumer instances will subscribe to multiple partitions at the same time (Figure 3.7 [12])

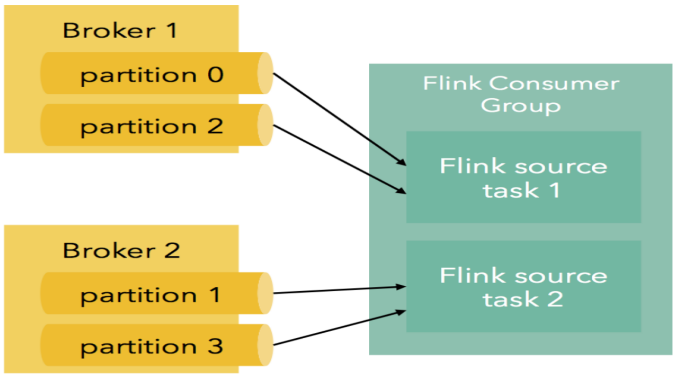


Figure 3.7: Kafka partitions assigned to Flink workers

The case that Kafka partitions equal to Flink parallelism is ideal since each consumer takes care of one partition. If the messages are balanced between partitions, the work will be evenly spread across Flink operators. Flink's Kafka consumer is called FlinkKafkaConsumer and provides access to one or more Kafka topics.

```
1        Properties properties = new Properties();
2        properties.setProperty("bootstrap.servers", "localhost:9092");
3        properties.setProperty("zookeeper.connect", "localhost:2181");
4        properties.setProperty("group.id", "test");
5        DataStream<String> stream = env
6       .addSource(new FlinkKafkaConsumer<>("topic", new SimpleStringSchema(),
           properties));
```

We use the JsonDeserializationSchema (and JSONKeyValueDeserializationSchema) which turns the serialized JSON into an ObjectNode object, from which fields can be accessed using objectNode.get("field").as(Int/String/...)(). The KeyValue objectNode contains a "key" and "value" field which contain all fields, as well as an optional "metadata" field that exposes the offset/partition/topic for this message.

Flink's Kafka Producer is called FlinkKafkaProducer and allows writing a stream of records to one or more Kafka topics.

```
1        DataStream<String> stream = ...;
2        FlinkKafkaProducer<String> myProducer = new FlinkKafkaProducer<String>(
3        "localhost:9092",            // broker list
4        "my-topic",                  // target topic
5        new SimpleStringSchema());   // serialization schema
6        stream.addSink(myProducer);
```

Flink's Kafka connectors provide some metrics through Flink's metrics system to analyze the behavior of the connector. The producers export Kafka's internal metrics through Flink's metric system for all supported versions. The consumers export all metrics starting from Kafka version 0.9.

3.5 The Importance of Apache Flink

Subsequently, we briefly describe some of the features that let Flink to have a wide acceptance in real-time analytics and applications:

1 Flink provides continuous streaming processing at event-driven applications and offers stream and batch analytics. Flink's engine, process data streams as true streams because each record is processed immediately and independently as soon as it arrives. Furthermore, Flink's expressive APIs and specific performance guarantees allow applications to run 24/7 and processes data at lightning fast speed (hence also called as 4G of Big Data).

2 Benchmarks have proven that Flink can compete with other well-known distributed Big Data platforms and that it can process millions of records per second. Users of Flink have reported impressive performance numbers, such as applications running on thousands of nodes that process multiple trillions of events per day. It has excellent performance with low latency, high throughput and in-Memory computing.

3 Streaming applications often require some custom state to maintain intermediate results of their computations. Flink uses a sophisticated late data handling and an asynchronous lightweight incremental checkpoint mechanism that guaranties exactly-once state consistency in case of a failure. It has fault tolerance as failure of hardware, node, software or a process does not affect the cluster.

4 Flink scales to any use case and is able to support very large state and incremental checkpointing. Finally, flexible deployment and the use of savepoints, make Flink the appropriate tool to manage Big Data.

Chapter 4

Implementation

In this chapter we describe the distributed implementation of query registration in Apache Flink. For the development phase, we use the DataStream API in Java to apply transformations on unbounded data streams. For data sources and sinks, we use Apache Kafka. During our implementation, we took into consideration the original source code of the Count-Min, Bloom Filter, AMS and FM algorithm. Each class that implements one of the previous algorithms, among other methods, it includes an add(Integer value) function and an estimate(Integer value) function, while the AMS class includes the estimateF2() function as well. However we use an abstract class called Sketch that contains all the above functions of every kind of sketch by inheritance. As it is easy to understand by the name of each method, the add function inserts a value in a sketch structure and the estimate function computes the desirable result. In case that sketch is an instance of AMS class, then the estimateF2() function computes the value F_2 of the frequency vector otherwise the function returns 0. In Figure 4.1 is illustrated the dataflow of the inheritance of the classes. Each class that extends the abstract class Sketch has a private field which references to an instance of one of the sub-classes.

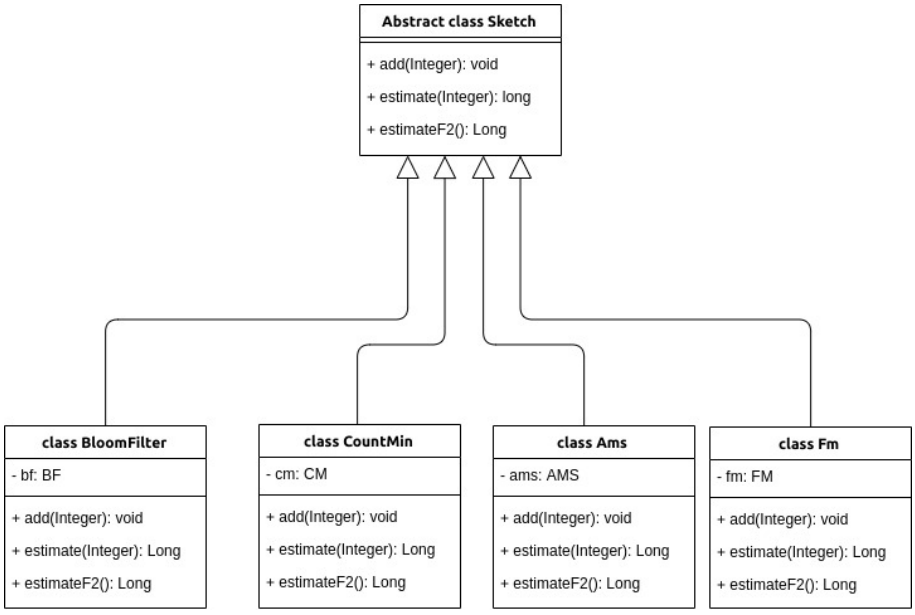


Figure 4.1: Diagram of class inheritance

Consequently, in the first section we talk about the implementation of Kafka connector and

how it interacts with Flink when is used as a data source. In the second section we introduce a parallel implementation for inserting values in the sketches after creating them. Moreover we request either to create new sketches or to estimate the result each sketch has computed. In the third section we feature the use of stateful operations and in the subsection we propose the idea of requesting queries in real-time.

4.1 Evaluation Methodology using Kafka Connector

First of all, it's worth mentioning that we use two different sources. The first one is used as the data input namely a <key, value>pair. We may have lots of streams in our architecture, so "key" refers to the stream id and "value" refers to the element that we want to enter in a sketch in order to compute some functions about it. The second one is used as a stream of requests of type <key, value>pair again. As we mentioned before it is able to request for a new instance of any sketch or for the estimated result of the wished value. So, "key" refers once again to the stream id that we want to deal with and "value" refers both to the element and the action we wish. This is accomplished as we split the String "value" information. Summarizing, we have a data input with the stream id and the according value and then we have a stream of requests which must first create some sketches in order to insert the values into them and then estimate the result.

To begin with, we built two Kafka Consumer and two Kafka Producer as it is pointed out in subsection 3.4.1. They all belong to the same group-id.

```
1 kafkaConsumer input = new kafkaConsumer("localhost:9092", "topicInput");
2 kafkaConsumer requests = new kafkaConsumer("localhost:9092", "topicRequests");
```

Then, we set two variables with type of DataStream<ObjectNode>and we use the addSource operator in order to attach a new source function and read the data input and the data requests from Apache Kafka.

```
1 DataStream<ObjectNode> InputStream = env.addSource(input.getFc());
2 DataStream<ObjectNode> ReqStream = env.addSource(requests.getFc());
```

4.2 Distributed implementation of query registration

The first two variables that anyone has to set in a Flink program is the StreamExecutionEnvironment and the setStreamTimeCharacteristic. In the first case we use the getExecutionEnvironment() in order to create an execution environment that represents the context in which the program is currently executed i.e. local or remote environment. In the second case we use TimeCharacteristic.Ingestion for further details see the section 3.1. Thus, we set the level of parallelism, although this parameter can be changed by the command line.

```
1 StreamExecutionEnvironment env = StreamExecutionEnvironment.
      getExecutionEnvironment();
2 env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime);
3 env.setParallelism(4);
```

As we have already set the two Kafka sources, we call the map operation for each of them. Hence, each line of the stream within the data input is transformed to Tuple2 <Integer,Integer>, while each line of the stream within the requests is transformed to Tuple2 <Integer,String>. After all, the "keyBy" operator is called in order to sort the tuples by the stream id. The operation "name" terms the above transformations to "DataInputMap" and "RequestMap" respectively when they appear at the streaming dataflow in Figure 4.2. Apache Flink Web Dashboard creates the dataflow execution of the Flink program.

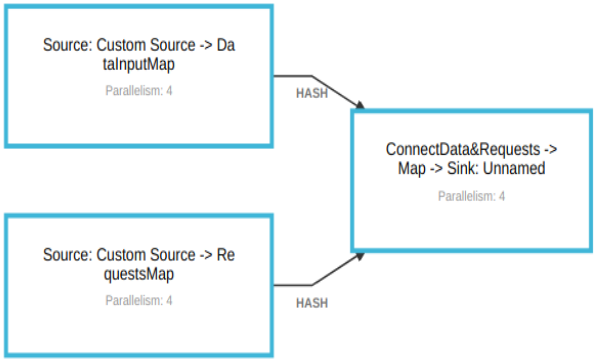


Figure 4.2: Streaming dataflow of the two Kafka Sources to map operators

In the sequel, we use the operator "connect" with the results of the above DataStreams as arguments. Then, we use a coflatMap transformation, named as "ConnectData&Requests" (Figure 4.2), which is worth to be described in detail. The flatMap transformation extends the RichCoFlatMapFunction as we manage ConnectedStreams now and we want to have access to the RuntimeContext in order to know the subtask id when we use a parallel implementation. So the coFlatMap contains two flatMap that execute in parallel for every of the above DataStream.

In the RichCoFlatMapFunction, initially, we set a variable type of ArrayList<Sketch>which stores a list with the available sketches. Furthermore, we set a variable type of HashMap<Integer, ArrayList<Sketch>>wherein we store the stream id and the aforementioned list of sketches.

In particular at the first flatMap, we get the list of sketches of each stream id and for each element we call the add(Integer value) function in the direction of entering the flowing values across all kind of sketches from the list.

At the second flatMap the second node of the Tuple2, splits its information to two variables by the comma delimiter. Now, there is the value and the act, which may be "create a CM sketch", "create a BF sketch", "create an AMS sketch", "create a FM sketch" or "estimate value". For example if the second node includes the value (240,1) that means that "240" refers to the requested value that we are interested of and "1" refers to the creation of a new instance of Count-Min sketch. The number "2","3","4" refers to the creation of a new instance of Bloom Filter, AMS and FM sketch correspondingly. Any other number refers to the estimation of the relevant value. If it is requested to create a new sketch, we make an instance of it, we get the list of sketches of the specific value (i.e. "240") and we add the new sketch to the list. What is more we use a variable to count the instances of each sketch that has been created. If it is requested to estimate a value for a stream id, we get the list of the sketches for the specific stream id and for each element we call the estimate(Integer value) function of class Sketch. The result of each sketch is printed on the screen while the collector of the RichCoFlatMapFunction collects a record Tuple2 with the information of the stream id and the list of the sketches which relates to. So, this operator produces a DataStream<Tuple2 <Integer, ArrayList<Sketch>>.

4.3 Distributed Solution using State

Afterward, we wish to use a stateful function and operator to store the list of the sketches that every stream id owns. At the beginning we use the KeyBy operator to group the output of the "ConnectData&Requests" operator according to the stream id. By using the KeyBy operator, we construct a new KeyedStream that each key represents the different number of the streams. For each one of the keys, we apply the KeyedProcessFunc() function. We decided to use a ValueStateDescriptor and to set it as Queryable. So the ValueState stores the String information. In this way, we store the most up-to-date list of available sketches for each different stream id in the Keyed State of Flink. Finally data sinks consume the DataStream<String >and forward it to files or we could use the addSink() function to write to a Kafka topic. In Figure 4.3 is illustrated the streaming dataflow of the total implementation of the program.

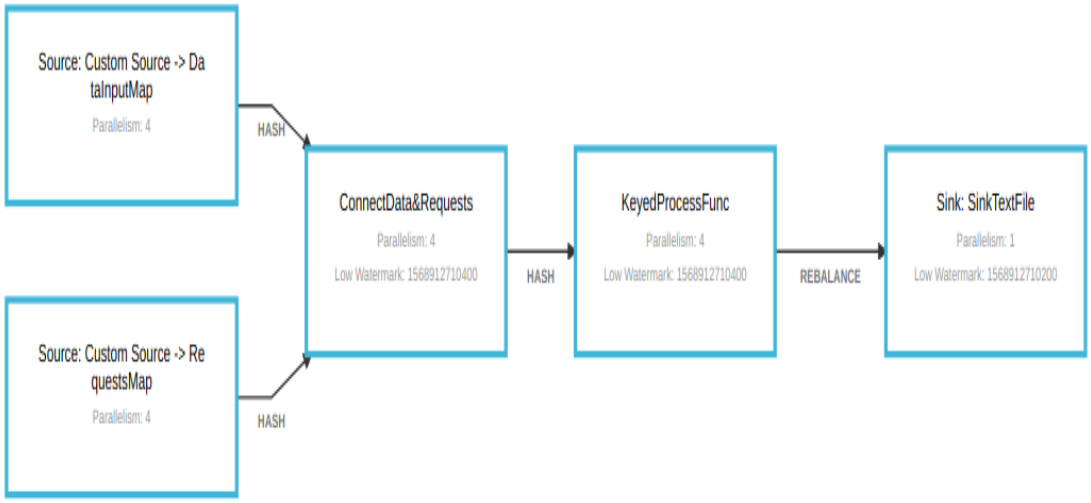


Figure 4.3: Streaming dataflow of Flink program

4.3.1 Real-Time Querable State

In this subsection, we exploit the Queryable State Feature of Flink to allow the user to query the aforementioned keys from outside Flink's runtime environment. The Queryable State of Flink, exposes the Keyed State to the outside world and allows the user to query the available keys of a specific operator. The Queryable State feature is capable of performing queries only in states maintained by KeyedStreams. In our use case, the keyed state is stored inside the KeyedProcessFunc() function, to the ValueState data structure. In order to query the keys of the ValueState, we have to specify the hostname and the port of the TaskManager. Then, we have to specify the name of the data structure that holds the state (e.g "request"). Afterwards, we submit the query to the Task Manager with a specific key.

```
1  CompletableFuture<ValueState<String>> getKvState =
2                      client.getKvState(JobID.fromHexString("JobId"),"request",
                           key, TypeInformation.of(new TypeHint<String>() {}),
                           mydescriptor);
```

Internally, the QueryableStateClientProxy of the Task Manager receives the request, and then asks the Job Manager which one of Task Managers holds the value of the queryed key. Based on that answer, the proxy will retrieve the value from the QueryableStateServer of the corresponding Task Manager. This value is then returned from the proxy to the client.

Chapter 5

Experimental Evaluation

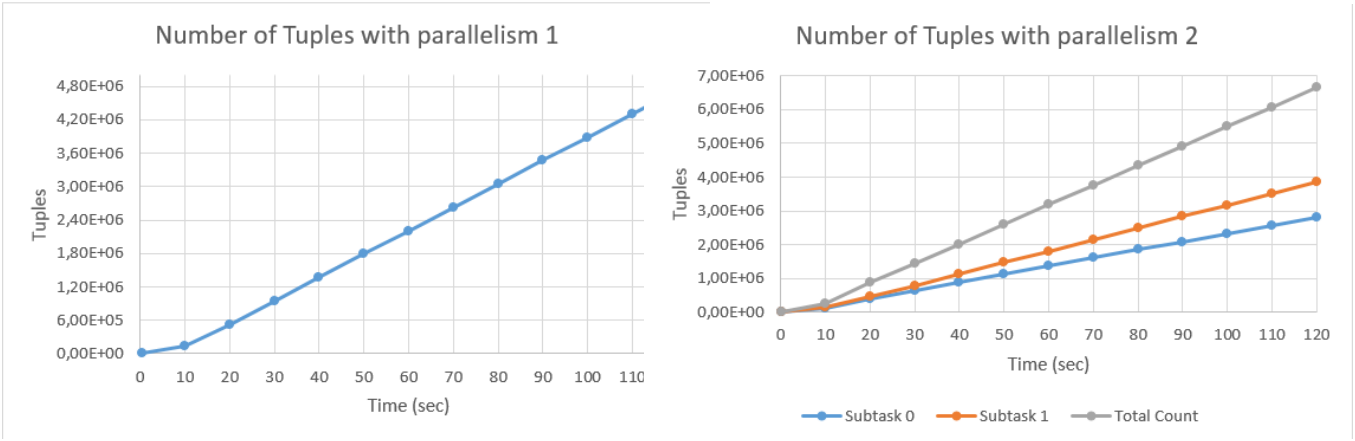
We conducted several experiments locally and remotely using the multi-node cluster of our university, to evaluate the performance of the distributed implementation of query registration over synopses.

5.1 Locally experiments

To begin with we ran locally experiments to a portable computer equipped with Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz. During our experiments we used unbounded data streams using Apache Kafka. The main source of the input dataset is a csv file that contains information about a population. Moreover there are eight columns "Year", "District.Code", "District.Name", "Neighborhood.Code", "Neighborhood.Name", "Gender", "Age", "Number". We used the fourth and the eighth column in order to have 73 distinct keys and 14017 values. So we read topics from Kafka continuously from the beginning of the csv file in order to create an unbounded stream. The source of the request dataset is a txt file with key,value pairs that we created. We chose randomly to create 17 count-min, 15 bloom filter, 7 ams and 5 FM sketches with 40 distinct keys while the remotely experiments are more specified.

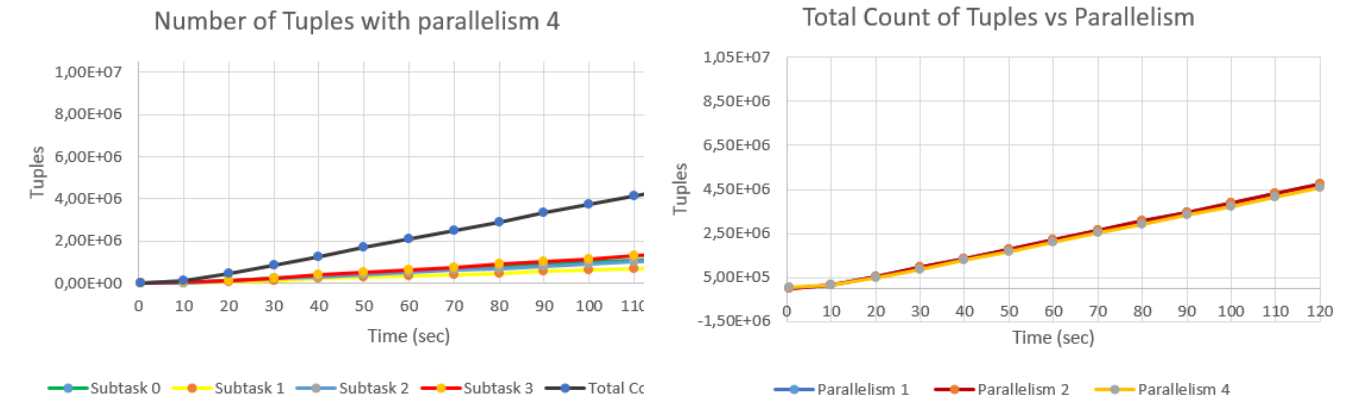
In figure 5.1 is illustrated the number of tuples that are inserted into various sketches and different stream ids while the input dataset increases. The subfigure (a), (b) and (c) exhibits the number of tuples per parallelism 1,2 and 4 respectively. In addition each subfigure presents both the number of tuples per subtask, when level of parallelism is greater than 1, and the total cumulative number of tuples per parallelism level.

In subfigure (d) of figure 5.1 we ascertain that the total number of tuples that are inserted into sketches remains about the same irregardless the level of parallelism that has been increased. This was expected as we ran the experiments locally and so there were not any more available resources in order to improve the performance of the system. Finally we note that the number of tuples inserted into sketches increases linearly and with constant ratio in time. Furthermore in local mode the total number of tuples is much smaller than in remote mode one.



(a) Insert value to sketches vs parallelism 1

(b) Insert value to sketches vs parallelism 2



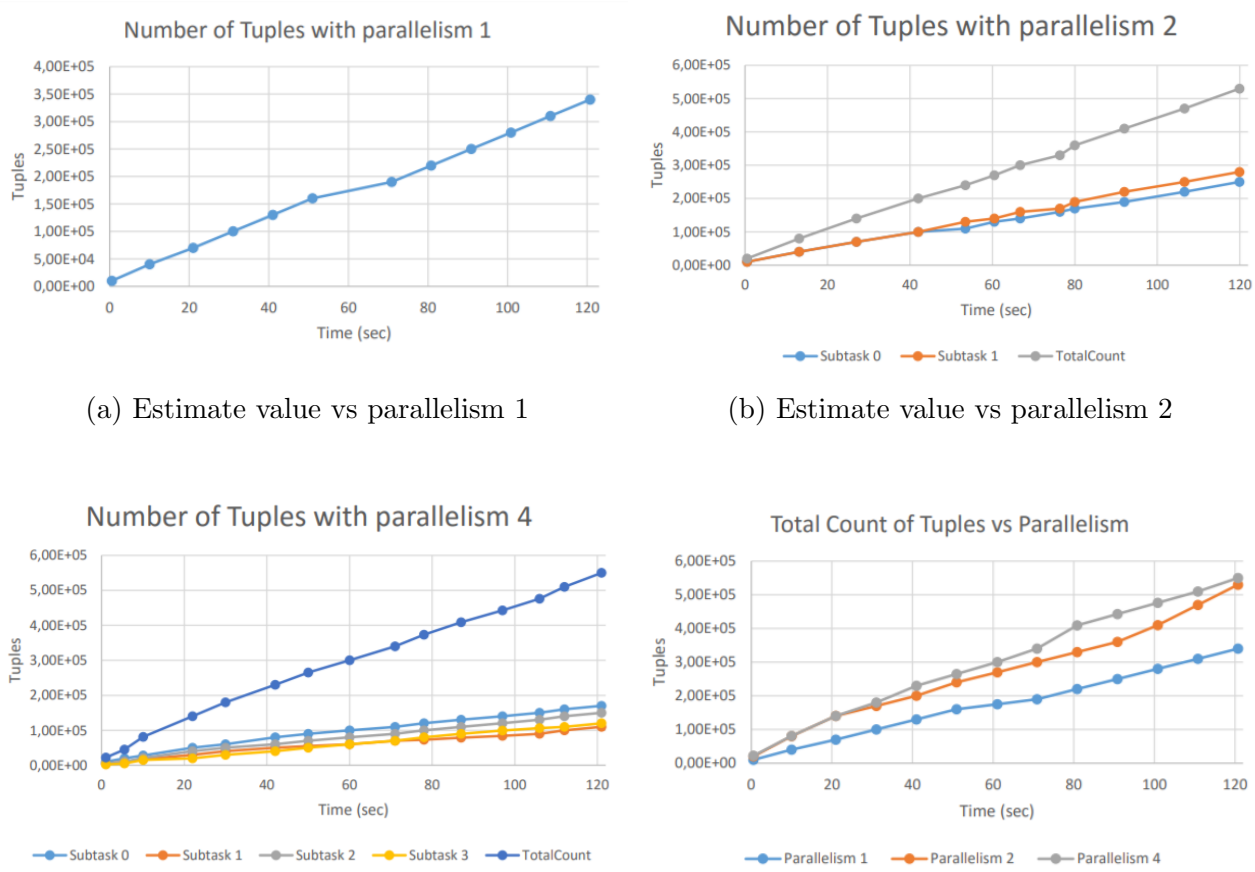
(c) Insert value to sketches vs parallelism 4

(d) Total count of tuples vs parallelism

Figure 5.1: Illustration of inserting values to various sketches

In figure 5.2 is illustrated the number of tuples that are estimated from various sketches to 42 distinct stream ids while the request dataset increases. The subfigure (a), (b) and (c) exhibits the number of tuples per parallelism 1,2 and 4 respectively. In addition, each subfigure presents both the number of tuples per subtask, when level of parallelism is greater than 1, and the total cumulative number of tuples per parallelism level.

In subfigure (d) of figure 5.2, we ascertain that the total number of tuples that are estimated from sketches remains about the same irregardless the level of parallelism that has been increased. As mentioned before there were not any more available resources in order to improve the performance of the system. Finally we note that the number of tuples estimated from sketches increases almost linearly. Furthermore in local mode the total number of tuples is much smaller than in remote mode one.



(a) Estimate value vs parallelism 1

(b) Estimate value vs parallelism 2

(c) Estimate value vs parallelism 4

(d) Total count of tuples vs parallelism

Figure 5.2: Illustration of estimate values from various sketches

5.2 Flink Cluster Setup

In order to run our experiments to the multi-node cluster of our university, we deployed Flink by using the Standalone Cluster setup. This setup includes a single Job Manager (master node) and at least one Task Manager (worker nodes). In our setup, we used 12 Task Managers with maximum number of parallel task slots 36 (i.e. 36 physical cores). During our experiments, the maximum Job parallelism that we used was 32, so we let Flink's runtime to make the choice of the specific task slots. The table below presents the system specifications of the Job and Task managers.

Flink Cluster Setup			
Node	CPU	Cores	Ram GB
1 Job Manager	Intel Xeon E5-2430 v2	6	32
12 Task Managers	Intel Xeon X3323	4	8

5.3 Remotely Experiments

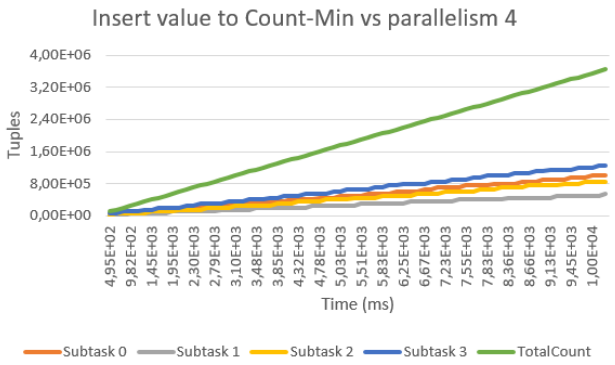
In remote mode we ran several experiments about inserting a value to a specific kind of sketch each time. The characteristics of a new sketch when it is dynamically created are presented below.

- Count-Min: epsilon=0.0002, confidence=0.99 and seed=4
- Bloom Filter: falsePositivePropability=0.02 and expectedNumOfElements=1000000
- AMS: depth=5 and numOfBuckets=512
- FM: bitmapSize=32, numHashGroups=64 and numHashFunctionsInEachGroup=32

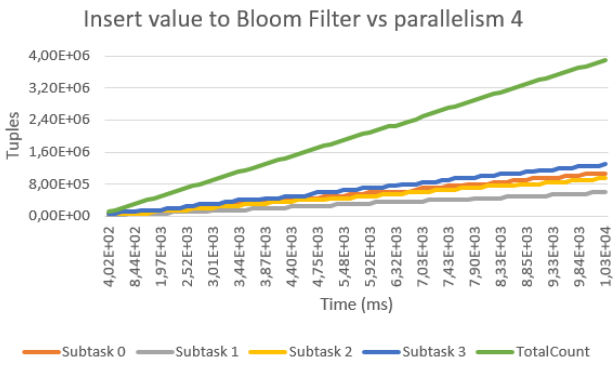
5.3.1 Insert value into sketches

In this subsection we discuss about the experiments of inserting a value into a sketch. In subfigure (a), (b), (c), (d) of figure 5.3 we present the number of tuples that are inserted into Count-Min, Bloom Filter, AMS and FM sketch respectively. In every experiment we used 45 instances of each sketch to 40 distinct stream ids. We note that the number of tuples that are inserted in every sketch is almost equal (figure 5.4). The total number of tuples of every subfigure is much higher than the results of the local experiments. Tuples increase linearly for each kind of sketch.

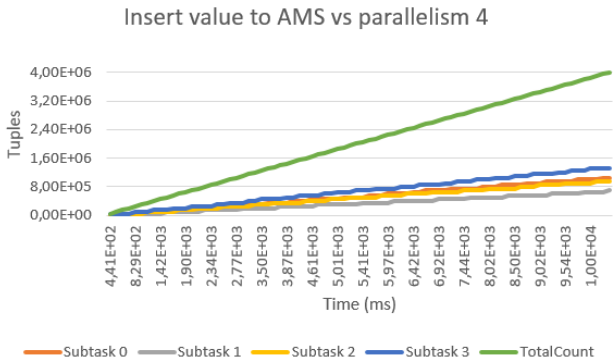
As we increase the parallelism level, Flink Cluster provides more resources and thus the number of tuples increases as well. Indicatively in figure 5.5 we present the result of inserting values to Count-Min sketch with parallelism 4 and 8.



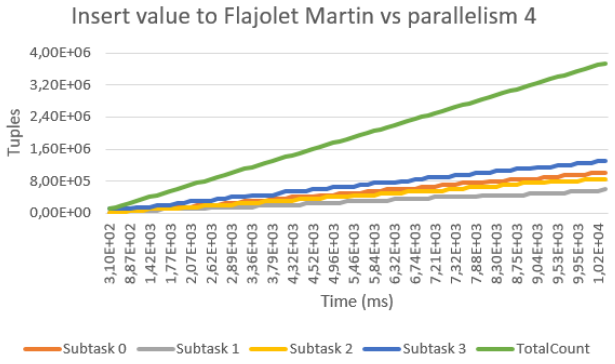
(a) Insert value to Count-Min vs parallelism 4



(b) Insert value to Bloom Filter vs parallelism 4



(c) Insert value to AMS vs parallelism 4



(d) Insert value to FM vs parallelism 4

Figure 5.3: Illustration of insert values in every sketch

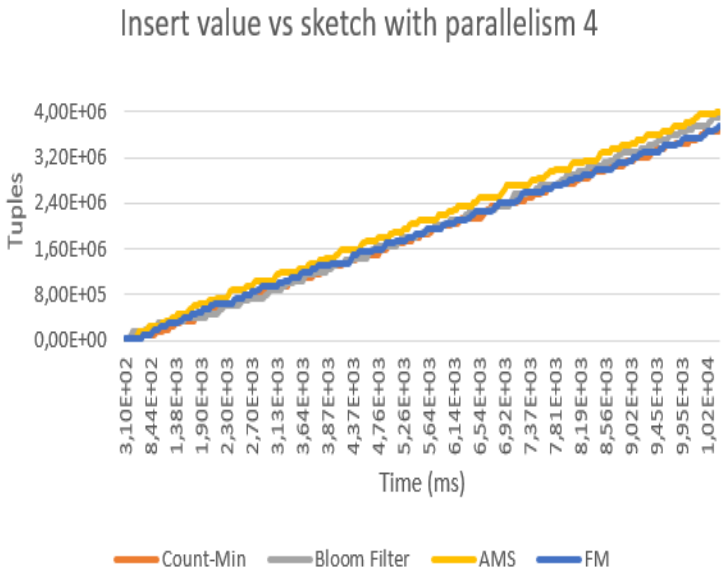
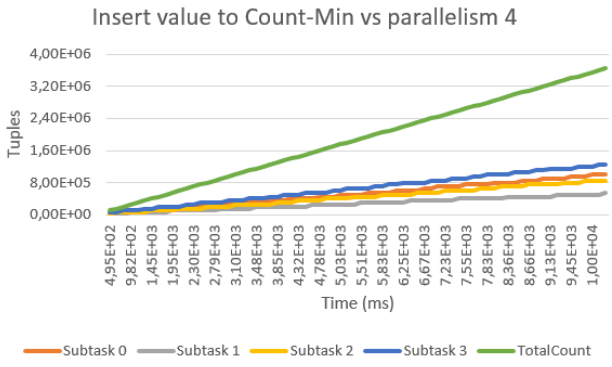
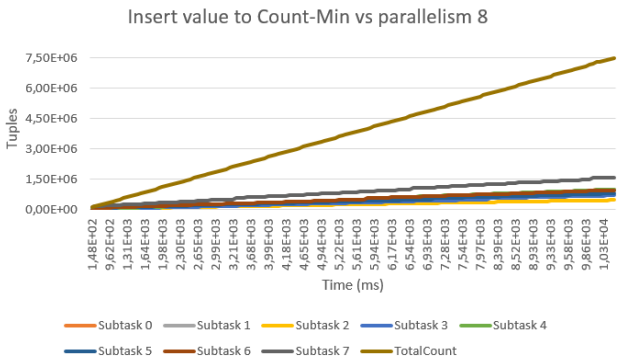


Figure 5.4: Insert value vs sketch with parallelism 4



(a) Insert value to CM vs parallelism 4



(b) Insert value to CM vs parallelism 8

Figure 5.5: Illustration of insert values into sketches vs parallelism level

In figure 5.6 is illustrated the number of tuples of inserting values into each sketch per parallelism level for 10000ms. When we double the parallelism level the number of tuples almost doubles as well.

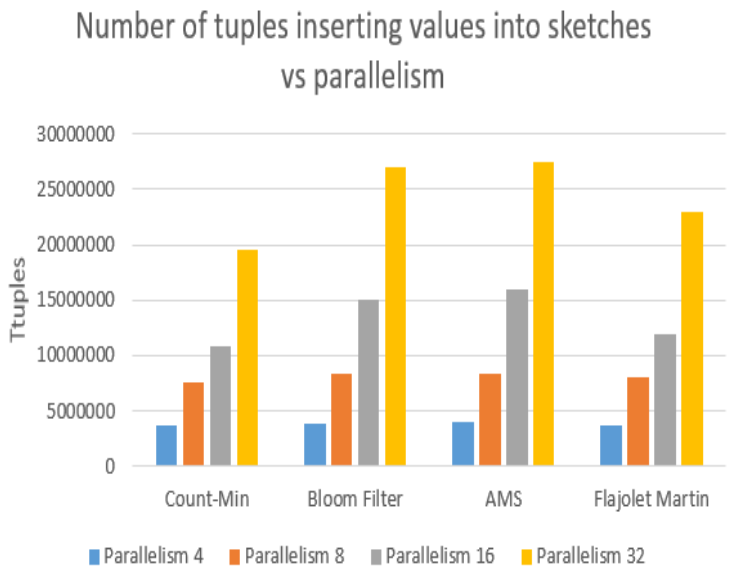
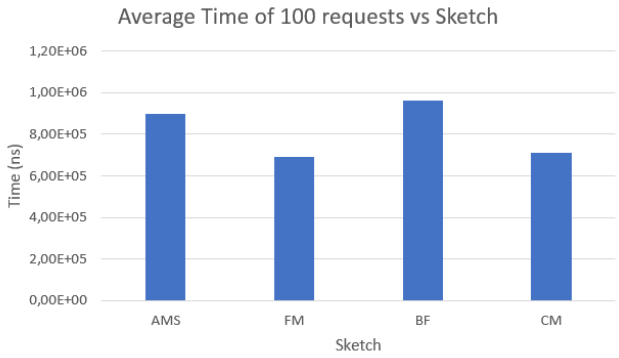


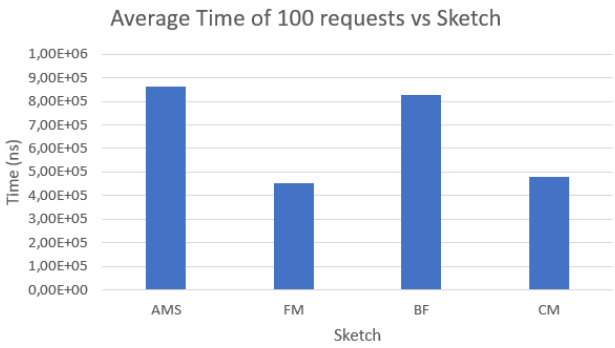
Figure 5.6: Insert values into sketches for 10000ms time period vs parallelism level

5.3.2 Estimate value

In this subsection we discuss about the experiments of estimating a value of each sketch. Indicatively, in subfigure (a) and (b) of figure 5.7 we present the average time needed for each subtask of every sketch to compute 100 requests for estimating a value for parallelism level 4 and 8 respectively. In every experiment we used 35 instances of each sketch with distinct streams ids. In order to obtain the results of the above experiments we recorded 10 samples of estimated time for each subtask and we computed the mean for each subtask. Finally, the largest value of time of all the subtasks of each sketch was chosen as the time that each sketch needed. It is worth mentioned that time was measured in nanoseconds.



(a) Average time to estimate values vs sketch and parallelism 4



(b) Average time to estimate values vs sketch and parallelism 8

Figure 5.7: Average time to compute 100 requests vs sketch vs parallelism

5.3.3 Throughput

Whereupon we exhibit the results of throughput of the above experiments. In figure 5.8 is illustrated the average number of tuples per second for inserting a value into each kind of sketches while using different levels of parallelism. It is expected that as we double the parallelism level, throughput doubles as well.

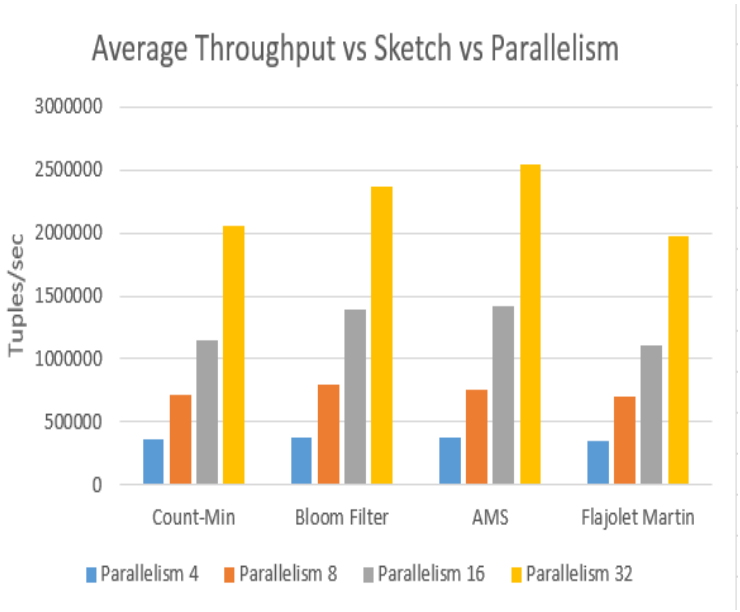


Figure 5.8: Average Throughput inserting value to sketches vs parallelism level

Concerning the throughput of estimating a value of each sketch we present indicatively the throughput of each subtask of every sketch per 100 requests and parallelism 4 in figure 5.9 for different stream ids.

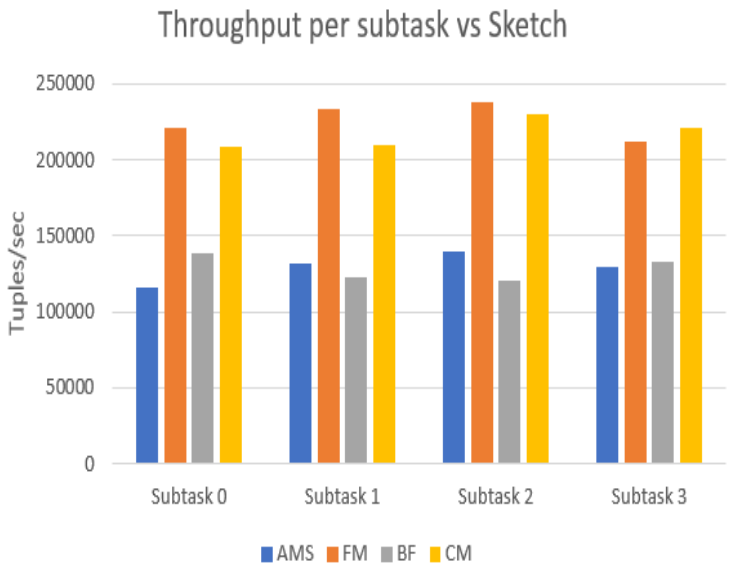


Figure 5.9: Throughput estimate per subtask vs sketch for parallelism level 4

In figure 5.10 is illustrated the total throughput of estimating at each subtask 100 requests for each sketch for different parallelism levels. The Flajolet Martin sketch seems to responded better at estimating values while the Count-Min sketch comes next. As we increased the parallelism level, throughput enlarged as well.

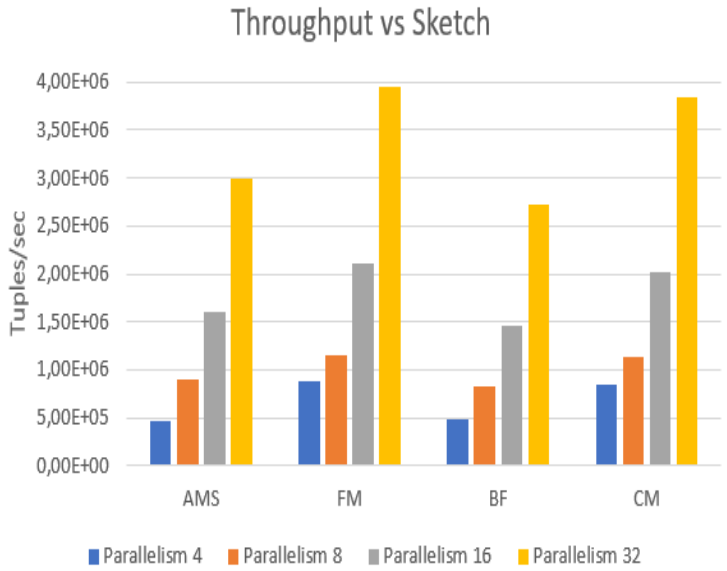


Figure 5.10: Throughput vs sketch vs parallelism level

Chapter 6

Conclusions & Future Work

In this diploma thesis, we proposed a distributed implementation of query registration over synopses. We focused on various streaming algorithms of sketches such as Bloom Filter, Count-Min, Flajolet-Martin and AMS sketches. The requirements included the dynamical creation of new sketches in real-time execution, update the sketches and computation of several functions such as the cardinality of the elements, the amount of distinct elements, or inform about the existence of an element in a stream. Furthermore, we developed a program that exploits the Queryable State feature of Flink, in order to allow the user to query the most up-to-date values of the sketches.

For the development phase, we used Apache Flink framework which is a distributed processing engine for large scale computations over unbounded and bounded data streams. It has excellent performance with low latency, high throughput and in-Memory computing.

We conducted several experiments locally and remotely to evaluate the performance of our implementation. The experimental results of the cluster proved that by increasing the job parallelism, the running time drops significantly and at the same time the throughput gets better. Insertion and estimation increases linearly and constantly. Throughput experiments showed that our implementation handles efficiently the growth in the size of the input and requests.

In future work more sketches and more structures of synopses could be added to this project. In addition, the execution result (about the state) could be used as a source to another framework i.e. Apache Spark in order to compute new methods.

References

[1] Ravi Bhideh. *Flajolet-Martin algorithm*. URL: http://ravi-bhide.blogspot.com/2011/04/flajolet-martin-algorithm.html.

[2] Burton H. Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Computer Usage Company, Newton Upper Falls, MA* 13 (1970), Pages 422–426.

[3] Paris Carbone et al. "Apache flink: Stream and batch processing in a single engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[4] Graham Cormode. "Count-min sketch". In: *Encyclopedia of Database Systems* (2009), pp. 511–516.

[5] Graham Cormode et al. "Synopses for massive data: Samples, histograms, wavelets, sketches". In: *Foundations and Trends® in Databases* 4.1–3 (2011), pp. 1–294.

[6] *Dataflow Programming Model*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.9/concepts/programming-model.html#programs-and-dataflows.

[7] *Distributed Runtime Environment*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.9/concepts/runtime.html#job-managers-task-managers-clients.

[8] Philippe Flajolet and G Nigel Martin. "Probabilistic counting algorithms for data base applications". In: *Journal of computer and system sciences* 31.2 (1985), pp. 182–209.

[9] *Flink DataStream API Programming Guide*. URL: https://ci.apache.org/projects/flink/flink-docs-stable/dev/datastream_api.html.

[10] Ellen Friedman and Kostas Tzoumas. *Introduction to Apache Flink: Stream Processing for Real Time and Beyond.* " O'Reilly Media, Inc.", 2016.

[11] Nishant Garg. *Apache Kafka*. Packt Publishing Ltd, 2013.

[12] Robert Metzger. *Kafka + Flink: A Practical, How-To Guide*. URL: https://www.ververica.com/blog/kafka-flink-a-practical-how-to.

[13] Y. Matias N. Alon and M. Szegedy. "The space complexity of approximating the frequency moments". In: *In Proc. of the 1996 Annual ACM Symp. on Theory of Computing* (1996), pp. 20–29.

[14] Jelani Nelson. "Sketching and streaming algorithms for processing massive data". In: (2012).

[15] Stefan Nilsson. *Bloom filters explained*. URL: https://yourbasic.org/algorithms/bloom-filter/.

[16] *What is Apache Flink? — Applications*. URL: https://flink.apache.org/flink-applications.html.