TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Parallel sketch algorithms with Spark, Storm, Akka and Kafka-Streams

Author

Angelos Petheriotis

Thesis committee

Prof. Deligiannakis Antonios
Prof. Samoladas Vasilis
Prof. Garofalakis Minos

January 2021

Abstract

Efficient processing over massive data sets has taken an increasing importance in the last few decades due to the growing availability of large volumes of data in a variety of applications in computer science. Typical streaming algorithms use space at most polylogarithmic in the length of input stream. Using linear space motivates the design for summary data structures with small memory footprints, also known as synopses. Algorithms such as count min sketch use parameters error and probability of failure, which are specified by the user, in order to extract the items that exceed some threshold (support) from an unbounded data stream. Accuracy guarantees are typically expressed in terms of those parameters (error, probability of failure) meaning that the error in extracting those frequent items is within a factor of $1 + error$ of the true items' frequency with probability at least $1 - \delta$. The memory footprint depend on these parameters.

Since we make only one pass over the unbounded data stream we need to make sure we utilise the most appropriate frameworks to run our computations on. We introduce Storm, Spark, Akka and Kafka Streams frameworks which are all real-time, distributed and fault-tolerant tools. Those four frameworks have a completely different architecture to the batch processing frameworks that have been established over the years. Furthermore each of these frameworks have many differences with regards to the architectural aspect between them, aspects which we try to analyse in this diploma thesis.

We evaluate CMS, ECMS, AMS algorithms at those four frameworks in a multi node cluster the topology with regards to performance. We observe throughput, the number of processed items per second while simultaneously we make sure the error guarantees are met.

The data which were used to benchmark the different algorithms on the four frameworks, are real data from smart thermostats from 50K different units.

Περίληψη

Η αποτελεσματική επεξεργασία σε μεγάλα σύνολα δεδομένων έχει αποκτήσει μεγάλη σημασία τις τελευταίες δεκαετίες, λόγω της συνεχώς αυξανόμενης ροής των δεδομένων καθώς και της συνεχώς αυξανόμενης ανάγκης για ανάλυση αυτών των δεδομένων. Τυπικοί αλγόριθμοι ροής (χρησιμοποιούν χώρο πολυλογαριθμικό μήκος στη βέλτιση περίπτωση). Η χρήση γραμμικού χώρου παρακινεί το σχεδιασμό για συνοπτικές δομές δεδομένων με χαμηλές απαιτήσεις μνήμης, γνωστές και ως συνόψεις. Αλγόριθμοι όπως ο υπολογισμός του ελάχιστου σχήματος χρησιμοποιούν συγκεκριμένες παραμέτρους σφάλματος και πιθανότητας αποτυχίας, τα οποία καθορίζονται από το χρήστη, προκειμένου να εξαχθούν τα στοιχεία που υπερβαίνουν κάποιο όριο (υποστήριξη) από μια απεριόριστη ροή δεδομένων. Οι εγγυήσεις ακρίβειας εκφράζονται συνήθως με βάση αυτές τις παραμέτρους (σφάλμα, πιθανότητα αποτυχίας) που σημαίνει ότι το σφάλμα κατά την εξαγωγή αυτών των συχνών στοιχείων είναι εντός ενός παράγοντα $1 + error$ της συχνότητας των πραγματικών στοιχείων με πιθανότητα τουλάχιστον $1 - delta$. Το αποτύπωμα μνήμης εξαρτάται από αυτές τις παραμέτρους.

Δεδομένου ότι κάνουμε μόνο ένα πέρασμα στην ροή των δεδομένων, πρέπει να διασφαλίσουμε ότι θα χρησιμοποιήσουμε τα καταλληλότερα εργαλεία για την εκτέλεση των υπολογισμών μας. Παρουσιάζουμε τα frameworks Storm, Spark, Akka Kafka Streams τα οποία είναι εργαλεία που επεξεργάζονται μια ροή δεδομένων σε πραγματικό χρόνο, ενώ παράλληλα μπορούν εύκολα να παραλληλοποιηθούν και να ειναι ανθεκτικά σε σφάλματα (διακοπή δικτύου κλπ). Αυτά τα τέσσερα frameworks έχουν μια εντελώς διαφορετική αρχιτεκτονική σε σχέση με τα υπάρχοντα frameworks τα οποία απευθύνονται σε offline ανάλυση των δεδομένων. Επιπλέον, κάθε ένα από αυτά τα frameworks είναι διαφορετικό σε αρχιτεκτονική μεταξύ των άλλων, αρχιτεκτονικές τις οποίες προσπαθούμε να αναλύσουμε σε αυτή τη διπλωματική εργασία.

Αξιολογούμε τους αλγόριθμους CMS, ECMS, AMS σε αυτά τα τέσσερα frameworks σε ένα σύμπλεγμα πολλαπλών κόμβων. Παρατηρούμε την απόδοση, τον αριθμό επεξεργασμένων μηνυμάτων ανά δευτερόλεπτο, ενώ παράλληλα εξασφαλίζουμε ότι πληρούνται οι εγγυήσεις σφάλματος.

Τα δεδομένα που χρησιμοποιήθηκαν για τη συγκριτική αξιολόγηση των διαφορετικών αλγορίθμων στα τέσσερα πλαίσια είναι πραγματικά δεδομένα από έξυπνους θερμοστάτες σε διάστημα 6 μηνών από 500K διαφορετικές μονάδες.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Prof. Antonios Deligiannakis for the continuous support on my studies and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this dissertation. I could not have imagined having a better advisor and mentor for my studies.

Besides my advisor, I would like to thank the rest of my dissertation committee: Prof. Vasilis Samoladas and Prof. Minos Garofalakis. My sincere thanks also goes to Polyxeni Arapi and Odysseas Papapetrou for their help during this dissertation.

Nobody has been more important to me in the pursuit of this project than the members of my family. I would like to thank my parents, whose love and guidance are with me in whatever I pursue. They are the ultimate role models. Most importantly, I wish to thank my supportive partner, Ioanna, who provided and still provides unending inspiration.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Objectives

Statistical analysis and mining of huge multi-terabyte data sets is a common task nowadays, especially in the areas like web analytics and Internet advertising. Analysis of such large data sets often requires powerful distributed data stores like Hadoop and heavy data processing with techniques like MapReduce. This approach often leads to heavyweight high-latency analytical processes and poor applicability to realtime use cases. On the other hand, when one is interested only in simple additive metrics like total page views or average price of conversion, it is obvious that raw data can be efficiently summarised, for example, on a daily basis or using simple in-stream counters. Computation of more advanced metrics like a number of unique visitor or most frequent items is more challenging and requires a lot of resources if implemented straightforwardly. In the current dissertation, we provide an overview of probabilistic data structures that allow one to estimate these and many other metrics and trade precision of the estimations for the memory consumption and latency.

During this diploma thesis we are going to analyse compact structures that allow one to answer the following queries:

- How many distinct elements are in the data set (i.e. what is the cardinality of the data

set)

- What are the frequencies of the most frequent elements ?
- What are the most frequent elements (the terms “heavy hitters” and “top-k elements”) ?
- What is the estimated size of the self join?

For example the frequent items problem is one of the most heavily studied questions in data stream research, dating back to the 1980s. Finding frequent items has played an essential role in many important data mining tasks, such as outliers detection, sequential patterns, classification, clustering, etc. The main goal of a frequent items application is the analysis of vast amounts of data for discovering useful information.

The challenging part of those applications is the fact that in real life the useful information is extracted from data sources that are really huge. Web server logs, twitter logs etc. increase with a significant high rate. This renders the long term storage of data impossible, as it would cost a lot of memory and time. We thus need to come up with appropriate mechanisms, so as to avoid storing this vast amount of data. Instead, we have to make algorithms that quickly process each piece of information quickly. The data processing can, in general, be divided into batch processing and real-time processing. Batch processing is preferred in cases where data are preselected (through scripts or shell) and stored in memory. Real-time processing is preferred in cases where data are processed as they flow through various systems.

We approach this problem with sketch techniques. Sketch techniques, in general, map the input stream to a small compacted structure from which one can get responses for queries as the ones mentioned above.

- In chapter 2 we analyse each proposed algorithm individually, so the reader can have a comprehensive understanding of those sketches.
- In chapter 3 we explain the architecture of some data processing frameworks like Spark and Kafka streams.

- In chapter 4 we briefly explain some of the available techniques to create a well defined development pipeline in order to have faster iterations from development to production.

- In chapter 5 we combine the sketch algorithms with the data frameworks and we run experiments using multiple different configurations to get a good understanding of their performance characteristics.

Chapter 2

Algorithms

2.1 CMS

2.1.1 Sketch overview

The Count-Min-Sketch (CMS) is a compact summary data structure capable of representing a high-dimensional vector and answering queries on this vector, in particular point queries and dot product queries, with strong accuracy guarantees. Such queries are at the core of many computations, so the structure can be used to answer a variety of queries, such as frequent items (heavy hitters), quantile finding, join size estimation, and more.

Since the data structure can easily process updates in the form of additions or subtractions to dimensions of the vector (which may correspond to insertions or deletions, or other transactions), it is capable of working over streams of updates, at high rates. The data structure maintains the linear projection of the vector with a number of other random vectors. These vectors are defined implicitly by simple hash functions. Increasing the range of the hash functions increases the accuracy of the summary, and increasing the number of hash functions decreases the probability of a bad estimate. These tradeoffs are quantified precisely below. Because of this linearity, CM sketches can be scaled, added and subtracted, to produce summaries of the corresponding scaled and combined vectors.

2.1.2 Implementation

At first glance, the sketch looks quite like a Bloom filter [1] , as it involves the use of an array and a set of hash functions. There are significant differences in the details, however. The sketch is formed by an array of counters and a set of hash functions that map items into the array. More precisely, the array is treated as a sequence of rows, and each item is mapped by the first hash function into the first row, by the second hash function into the second row, and so on (note that this is in contrast to the Bloom filter, which allows the hash functions to map onto overlapping ranges). An item is processed by mapping it to each row in turn via the corresponding hash function and incrementing the counters to which it is mapped.

Given an item, the sketch allows its count to be estimated. This follows a similar outline to processing an update: inspect the counter in the first row where the item was mapped by the first hash function, and the counter in the second row where it was mapped by the second hash, and so on. Each row has a counter that has been incremented by every occurrence of the item. The counter was also potentially incremented by occurrences of other items that were mapped to the same location, however, since collisions are expected. Given the collection of counters containing the desired count, plus noise, the best guess at the true count of the desired item is to take the smallest of these counters as your estimate.

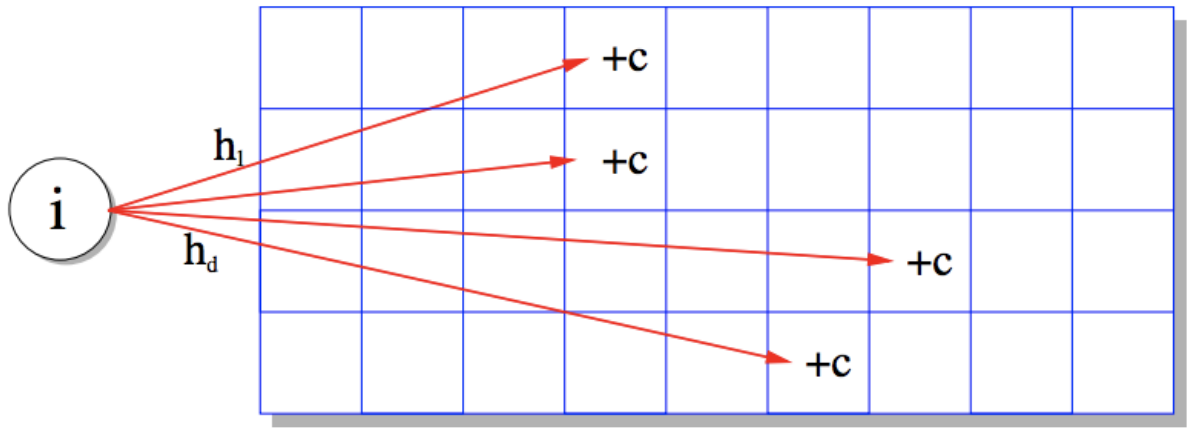

Figure 2.1: Count min example

Figure 2.1 shows the update process: an item i is mapped to one entry in each row j by the

hash function h_j, and the update of c is added to each entry. It can also be seen as modelling the query process: a query for the same item i will result in the same set of locations being probed, and the smallest value will be returned as the answer.

2.1.3 Sketch internals

With all data structures, it is important to understand the data organisation and algorithms for updating the structure, to make clear the relative merits of different choices of structure for a given task. The Count-Min Sketch data structure primarily consists of a fixed array of counters, of width w and depth d. All the counters are initialised to zeros. Each row of counters is associated with a different hash function. The hash function maps items uniformly onto the range $1, 2, ...w$.

The hash functions do not need to be particularly strong (i.e. they are not as complex as cryptographic hash functions). For items represented as integers i, the hash functions can be of the form $(a*i + b \mod p \mod w)$, where p is a prime number larger than the maximum i value (say, $p = 2^{31} - 1$ or $p = 2^{61} - 1$), and a, b are values chosen randomly in the range 1 to $p - 1$. It is important that each hash function is different, otherwise there is no benefit from the repetition.

$Update(i, c)$ operations update the data structure in a straightforward way. In each row, the corresponding hash function is applied to i to determine a corresponding counter. Then the update c is added on to that counter. Figure 2.1 shows an example of an update operation on a sketch with $w = 9$ and $d = 4$. The update of item i is mapped by the first hash function to an entry in the first row, where the update of c is added on to the current counter there. Similarly, the item is mapped to different locations in each of the other three rows. So in this example, we evaluate four different hash functions on i, and update four counters accordingly.

For $Estimate(i)$ operations, the process is quite similar. For each row, the corresponding hash function is applied to i to look up one of the counters. Across all rows, the estimate is found as the minimum of all the probed counters. In the example above, we examine each place where

i was mapped by the hash functions: in Figure 2.1, this is the fourth entry in the first row, the fourth entry in the second row, the seventh entry in the third row, and so on. Each of these entries has a counter which has added up all the updates that were mapped there, and the estimate returned is the smallest of these.

2.1.4 Discussion and Applications

As with the Bloom filter, the sketch achieves a compact representation of the input, with a tradeoff in accuracy. Both provide some probability of an unsatisfactory answer. With a Bloom filter, the answers are binary, so there is some chance of a false positive response; with a Count-Min sketch, the answers are frequencies, so there is some chance of an inflated answer.

What may be surprising at first is that the obtained estimate is very good. Mathematically, it can be shown that there is a good chance that the returned estimate is close to the correct value. The quality of the estimate depends on the number of rows in the sketch (each additional row halves the probability of a bad estimate) and on the number of columns (doubling the number of columns halves the scale of the noise in the estimate). These guarantees follow from the random selection of hash functions and do not rely on any structure or pattern in the data distribution that is being summarised. For a sketch of sizes, the error is proportional to $1/s$. This is an improvement over the case for sampling where, as noted earlier, the corresponding behaviour is proportional to $1/s$.

Just as Bloom filters are best suited for the cases where false positives can be tolerated and mitigated, Count-Min sketches are best suited for handling a slight inflation of frequency. This means, in particular, they do not apply to cases where a Bloom filter might be used: if it matters a lot whether an item has been seen or not, then the uncertainty that the Count-Min sketch introduces will obscure this level of precision. The sketches are very good for tracking which items exceed a given popularity threshold, however. In particular, while the size of a Bloom filter must remain proportional to the size of the input it is representing, a Count-Min sketch can be much more compressive: its size can be considered to be independent of the input size, depending instead on the desired accuracy guarantee only (i.e., to achieve a target

accuracy of ϵ, fix a sketch size of s proportional to $1/\epsilon$ that does not vary over the course of processing data).

2.1.5 Real word application

Since their introduction over a decade ago, Count-Min sketches have found applications in systems that track frequency statistics, such as popularity of content within different groups—say, online videos among different sets of users, or which destinations are popular for nodes within a communications network. Sketches are used in telecommunications networks where the volume of data passing along links is immense and is never stored. Summarising network traffic distribution allows hotspots to be detected, informing network-planning decisions and allowing configuration errors and floods to be detected and debugged. Since the sketch compactly encodes a frequency distribution, it can also be used to detect when a shift in popularities occurs, as a simple example of anomaly detection.

A social network such as Twitter may wish to track how often a tweet is viewed when displayed via an external website. There are billions of web pages, each of which could in principle link to one or more tweets, so allocating counters for each is infeasible and unnecessary. Instead, it is natural to look for a more compact way to encode counts of items, possibly with some tolerable loss of fidelity. Tracking the number of views that a tweet receives across each occurrence in different websites creates a large enough volume of data to be difficult to manage. Moreover, the existence of some uncertainty in this application seems acceptable: the consequences of inflating the popularity of one website for one tweet are minimal. Using a sketch for each tweet consumes only moderately more space than the tweet and associated metadata, and allows tracking which venues attract the most attention for the tweet. Hence, a kilobyte or so of space is sufficient to track the percentage of views from different locations, with an error of less than one percentage point, say.

2.1.6 A Few Observations About The Count Min Sketch

Count Min Sketch's efficiency can be demonstrated by reviewing its requirements. The space complexity of CMS is the product of w,d and the width of the counters that it uses. For example, a sketch that has 0.01 error rate at probability of 0.01 is created using 10 hash functions and 2000-counter arrays. Assuming the use of 16-bit counters, the overall memory requirement of the resulting sketch's data structure clocks in at 40KB (a couple of additional bytes are needed for storing the total number of observations and a few pointers). The sketch's other computational aspect is similarly pleasing—because hash functions are cheap to produce and compute, accessing the data structure, whether for reading or writing, is also performed in constant time.

CMS is certainly an excellent sketch, but there is at least one thing that prevent it from achieving perfection. This is that CMS can be biased, and thus overestimate the frequencies of samples with a small number of observations. CMS' bias is well known and several improvements have been suggested. The most recent is the Count Min Log Sketch [2], which essentially replaces CMS' linear registers with logarithmic ones to reduce the relative error and allow higher counts without increasing the width of counter registers.

2.1.7 Implementations of the Sketch

It is straightforward to implement the sketch in a traditional single CPU environment. Indeed, several libraries are available for the data structure, in common languages such as C, C++, Java and Python. Some skeletal pseudocode to initialise and update the sketch is shown in 2.2.

The code in Algorithm 1 initialises the array C of $w \times d$ counters to 0, and picks values for the hash functions based on the prime p.

For each $Update(i, c)$ shown in Algorithm 2, the total count N is updated with c, and the loop in Lines 2 to 4 hashes i to its counter in each row, and updates the counter there.

The procedure for $Estimate(i)$ shown in Algorithm 3 is almost identical to this loop: given i,

Algorithm 1: CountMinInit(w, d, p)

1 $C[1,1] \ldots C[d,w] \leftarrow 0$;
2 **for** $j \leftarrow 1$ **to** d **do**
3 Pick a_j, b_j uniformly from $[1 \ldots p]$;
4 $N = 0$;

Algorithm 2: CountMinUpdate(i, c)

1 $N \leftarrow N + c$;
2 **for** $j \leftarrow 1$ **to** d **do**
3 $h_j(i) = (a_j \times i + b_j \mod p) \mod w$;
4 $C[j, h_j(i)] \leftarrow C[j, h_j(i)] + c$;

Algorithm 3: CountMinEstimate(i)

1 $e \leftarrow \infty$;
2 **for** $j \leftarrow 1$ **to** d **do**
3 $h_j(i) = (a_j \times i + b_j \mod p) \mod w$;
4 $e \leftarrow \min(e, C[j, h_j(i)])$;
5 **return** e

Figure 2.2: CMS pseudo algorithm

we perform the hashing in line 3, and keep track of the smallest value of $C[j, h_j(i)]$ over the d values of j.

But again, having a single thread implementation is not a viable option in systems with large input volumes that need to respond to queries in a constant and quick manner. We have addressed the problem of parallesation of CMS and other sketch algorithms in the next chapters.

2.1.8 Scala code implementation

In this chapter we attach the Scala implementation of the CMS Sketch (the very basic components/methods of it).

```
/**
* Initialise CMS
*/
public CountMinSketchImpl(double eps, double confidence, int seed) {
    // 2/w = eps ; w = 2/eps
    // 1/2^depth <= 1-confidence ; depth >= -log2 (1-confidence)
    this.eps = eps;
    this.confidence = confidence;
    this.width = (int) Math.ceil(2 / eps);
    this.depth = (int) Math.ceil(-Math.log(1 - confidence) / Math.log(2));
    initTablesWith(depth, width, seed);
  }
private void initTablesWith(int depth, int width, int seed) {
    this.table = new long[depth][width];
    this.hashA = new long[depth];
    Random r = new Random(seed);
    // We're using a linear hash functions
    // of the form (a*x+b) mod p.
    // a,b are chosen independently for each hash function.
```

```
    // However we can set b = 0 as all it does is shift the results
    // without compromising their uniformity or independence with
    // the other hashes.
    for (int i = 0; i < depth; ++i) {
      hashA[i] = r.nextInt(Integer.MAX_VALUE);
    }
}
```

```
/**
* Increment counter of specific item (after hashing it for every line)
*/
public void addLong(long item, long count) {
    for (int i = 0; i < depth; ++i) {
      table[i][hash(item, i)] += count;
    }
    totalCount += count;
  }
```

```
/**
* Hash given item (using a fast implementation of the hashing algorithm)
*/
private int hash(long item, int count) {
    long hash = hashA[count] * item;
    // A super fast way of computing x mod 2^p-1
    // See http://www.cs.princeton.edu/courses/archive/fall09/cos521/Handouts/universalclasses.pdf
    // page 149, right after Proposition 7.
    hash += hash >> 32;
    hash &= PRIME_MODULUS;
    // Doing "%" after (int) conversion is ~2x faster than %'ing longs.
    return ((int) hash) % width;
  }
private static int[] getHashBuckets(String key, int hashCount, int max) {
        return getHashBuckets(Utils.getBytesFromUTF8String(key), hashCount, max);
}
```

```
private static int[] getHashBuckets(byte[] b, int hashCount, int max) {
    int[] result = new int[hashCount];
    int hash1 = Murmur3_x86_32.hashUnsafeBytes(b, Platform.BYTE_ARRAY_OFFSET, b.length, 0);
    int hash2 = Murmur3_x86_32.hashUnsafeBytes(b, Platform.BYTE_ARRAY_OFFSET, b.length, hash1);
    for (int i = 0; i < hashCount; i++) {
      result[i] = Math.abs((hash1 + i * hash2) % max);
    }
    return result;
}
```

```
/**
* Estimate the count by hashing the item and by getting the minimum value of all the rows
*/
private long estimateCountForLongItem(long item) {
    long res = Long.MAX_VALUE;
    for (int i = 0; i < depth; ++i) {
      res = Math.min(res, table[i][hash(item, i)]);
    }
    return res;
}
```

2.2 ECMS

2.2.1 Sketch overview

The ECM sketch was presented from the work of Papapetrou, Garofalakis and Deligiannakis [3]. It is designed to solve a similar set of problems to CMS but simultaneously being able to efficiently summarise high-dimensional streaming data over sliding windows.

The ECM-sketch combines the Count-Min sketch structure for conventional streams with state-of-the art tools for sliding window statistics. The end result is a sliding-window sketch synopsis that can provide provable, guaranteed-error performance for point and inner-product queries,

and can be employed to address a broad class of queries, such as maintaining frequency statistics, finding heavy hitters, and computing quantiles in the sliding-window model.

2.2.2 Exponential histograms

In order to succeed queries over sliding windows ECMS combines the functionalities of Count-Min sketches and exponential histograms [4]. As the CMS has been described in the previous section, we are going to describe only the histograms and how they have been used in the ECMS to provide sliding window aspect of the original CMS implementation.

Exponential histograms are a deterministic structure, proposed to address the basic counting problem, i.e., for counting the number of true bits in the last N stream arrivals. They belong to the family of methods that break the sliding window range into smaller windows, called buckets or basic windows, to enable efficient maintenance of the statistics. Each bucket contains the aggregate statistics, i.e., number of arrivals and bucket bounds, for the corresponding sub-range. Buckets that no longer overlap with the sliding window are expired and they are not considered when answering queries. To compute an aggregate over the whole (or a part of) sliding window, the statistics from all buckets overlapping with the query range are aggregated.

For example, for basic counting, aggregation is a summation of the number of true bits in the buckets. A possible estimation error can be introduced due to the oldest bucket inside the query range, which usually has only a partial overlap with the query. Therefore, the maximum possible estimation error is bounded by the size of the last bucket.

Algorithm Steps

Below we provide the high level steps of the algorithm that sustains a exponential histogram structure.

1. When a new data element arrives, calculate the new expiry time. If the timestamp of the last bucket indicates expiry, delete that bucket, and update the counter Last containing the size of the last bucket and the counter Total containing the total size of the buckets.

2. If the new data element is 0, ignore it; otherwise, create a new bucket with size 1 and the current timestamp, and increment the counter Total.

3. Traverse the list of buckets in order of increasing sizes. If there are k 2 + 2 buckets of the same size, merge the oldest two of these buckets into a single bucket of double the size. (A merger of buckets of size 2r may cause the number of buckets of size 2r+1 to exceed k 2 + 1, leading to a cascade of such mergers.)

2.2.3 Implementation

As discussed previously, ECMS uses CMS and exponential histograms in order to accomplish queries over sliding windows. In this section we present how the $insert(x)$ works for ECMS. In order to easily comprehend the structure of ECMS, it is enough to remember the following. CMS in each cell keeps a counter. ECMS on the other hand, keeps an instance of an exponential histogram for each cell.

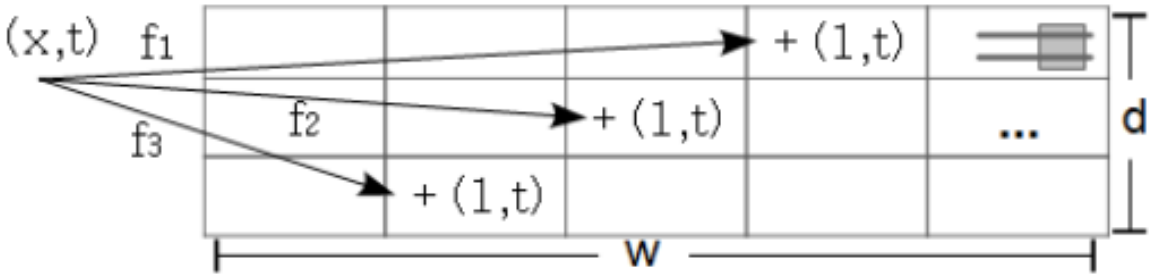

Figure 2.3: Adding an element to the ECMS

Adding an item x to the structure is similar to the case of standard Count-Min sketches. The

process for time-based sliding windows is depicted in 2.3. First, the counters $CM[j, hj(x)]$, where $j \in 1...d$, corresponding to the d hash functions are detected. For each of the counters, we register the arrival of the item at time t, and remove all expired information, i.e., the buckets of the exponential histogram that have no overlap with the sliding window range.

2.2.4 Point query estimation

A point query (x, r) is a combination of an item identifier x, and the query range r defined either as number of time units or number of arrivals. Point queries are executed as follows. The query item is hashed to the d counters $CM[j, h_j(x)]$ where $j \in 1...d$, and the estimate of each counter $E(j, h_j(x), r)$ for the query range is computed. The estimate value for the frequency of x is $min_{j=1...d} E(j, h_j(x), r)$

2.2.5 Merging ECM Sketches

Even though ECMS can handle large volumes of data in a compacted way, there are still cases where the input volume is so large that we need to parallelise/distribute the ECMS instance. Even though distributing is straightforward, it is enough to start multiple instances of the same ECMS, it is hard to answer to queries in a distributed fashion. For that reason ECMS needs to allow merging of the deployed instances so we can then answer the queries.

Suppose ECM-sketches $CM_1, CM_2, ..., CM_n$, each one corresponding to stream $S_1, S_2, ..., S_n$. The goal is to get a single ECM-sketch $CM \bigoplus$that corresponds to the logical stream $S = S1 \bigoplus S2 \bigoplus ... \bigoplus Sn$. The $\bigoplus$ operator is defined as a merging operator that preserves the ordering and arrival time of the events. Remember that standard Count-Min sketches allow merging, as long as all sketches are constructed with identical dimensions and hash functions. However, this does not trivially hold for ECM-sketches, where the counters are not simple numbers but complex sliding window structures.

So in order to succeed merging ECMS instances we need to examine merging exponential

histograms. Consider a set of exponential histograms $EH_1, EH_2, ..., EH_n$, summarising time-based sliding windows. All are configured to cover a sliding window of N time units. The merging operation is denoted with $\oplus$. To construct $EH\oplus$, ECMS algorithm considers the individual exponential histograms as logs. The basic idea is to reconstruct $EH\oplus$ by assuming that half of the elements arrive at the starting time of each bucket, and the remaining at the ending time of the bucket. We are not going to describe the detailed approach of merging ECMs but the high level idea is that a new empty histogram is created and then the entries from each EH_j get replayed in the order defined by the starting and ending timestamps of the buckets

2.2.6 Java code implementation

In this chapter we attach the Java implementation of the ECM Sketch (the very basic components/methods of it).

```
/**
* Add value to EMS. Triggers the `addOne` method of the histogram. See below for the implementation of
    `addOne` method
*
* @param value the value to add
* @param time  the time the value added
*/
public int[] add(long value, int time) {
        int[] hashes = hash(value, d, w);
    for (int depth = 0; depth < d; depth++) {
        slidingWindows[hashes[depth]][depth].addAOne(time);
    }
    lastUpdateTime = time;
    numberOfTotalProcessedEvents++;
    return hashes;
}
```

```
/**
* Add a One for a specific time
*
* @param time the time to add the One
*/
@Override
public void addAOne(int time) {
        currentRealtime = time;
    lastOneUpdate = time;
    lastSyncedTime = time;
    removeExpiredWithExpiryTime(currentRealtime - windowSize);
    bucketWallclockTimes[0].addLast(currentRealtime);  // BUCKET 0 is tmp by convention
    numberOfBuckets++;
    mergeIfNeeded(); // This is where the merging takes place
    updateTriggers();
}
```

```
/**
 * Estimate value from CMS within a time range
 *
 * @param value      the value to estimate
 * @param queryRange how many times before lastSyncTime to query for
 * @return the estimated value
 */
public double getEstimation(int value, int queryRange) {
    int[] hashes = hash(value, d, w);
    double val = Double.MAX_VALUE;
    for (int depth = 0; depth < d; depth++) {
        int w = hashes[depth];
        double est = slidingWindows[w][depth].getEstimationWithinRange(queryRange);
        val = Math.min(val, est);
    }
    return val;
}
```

```
public SlidingCMSketch mergeSlidingCMSketches(SlidingCMSketch[] dws, double delta, double epsilon, int
    windowSize) {
```

```
    // The new sliding sketch that will populated with data
    SlidingCMSketch scm = new SlidingCMSketch(delta, epsilon, windowSize);
    // Fr each cell/point of the new sliding sketch
    for (int i = 0; i < scm.w; i++) {
        for (int j = 0; j < scm.d; j++) {
            int c = 0;
            // Create a list with all the Exponential Histogram to merge
            ExponentialHistogramCircularInt[] al = new ExponentialHistogramCircularInt[dws.length];
            for (SlidingCMSketch single : dws) {
                al[c++] = (ExponentialHistogramCircularInt) single.slidingWindows[i][j];
            }
            // Merge all exponential histogramms for this point
            scm.slidingWindows[i][j] = new ExponentialHistogramCircularInt(epsilonSW,
                 windowSize).mergeEHs(al, epsilonSW, windowSize);
        }
    }
    return scm;
}
```

```
/**
 * Get the value's estimation from a specific start time
 *
 * @param startTime the point in time to start querying from
 * @return the estimated result
 */
@Override
public double getEstimationFromStartTime(int startTime) {
    double est = 0;
    int level = 0;
    int pow = 1;
    while (bucketWallclockTimes[level] != null
            && bucketWallclockTimes[level].size() > 0
            && bucketWallclockTimes[level].getFirst() >= startTime) {
        est += pow * bucketWallclockTimes[level].size();
        level++;
        pow *= 2;
    }
    // at this level, i need to find the proper element
    int originalPos = -1;
    if (bucketWallclockTimes[level] != null) {
        originalPos = bucketWallclockTimes[level].binarySearch(startTime);
    }
    int pos = originalPos;
    if (pos < 0) pos = -pos - 1;
    int levelBucketSize = pow;
    if (bucketWallclockTimes[level] == null || pos == bucketWallclockTimes[level].size()) {
        if (bucketWallclockTimes[level] != null && bucketWallclockTimes[level].size() > 0)
            est += levelBucketSize / 2;
        return est;
    } else {
        if (originalPos < 0)
            pos--;
        while (bucketWallclockTimes[level].get(pos) == startTime) pos--;
        pos++;// find the first element with this time.
        est += (bucketWallclockTimes[level].size() - pos) * levelBucketSize;
        if (levelBucketSize != 1) est += levelBucketSize / 2;
        return est;
    }
}
```

2.3 AMS

2.3.1 Sketch overview

The AMS sketch was first presented in the work of Alon, Matias and Szegedy [5]. It was proposed to solve a different problem, to estimate the value of F_2 of the frequency vector, the sum of the squares of the frequencies. Although this is straightforward if each frequency is presented in turn, it becomes more complex when the frequencies are presented implicitly, such as when the frequency of an item is the number of times it occurs within a long, unordered,

stream of items.

Frequency Moments

Consider a stream $S = a_1, a_2, ..., a_m$ with elements from a domain $D = v_1, v_2, ..., v_n$. Let m_i denote the frequency (also sometimes called multiplicity) of value $v_i D$; i.e., the number of times v_i appears in S. The $k^t h$ frequency moment of the stream is defined as:

$$F_k = \sum_{i=1}^{n} m_i^k$$

Estimating F_k may seem like a somewhat obscure goal in the context of approximate query processing. However, frequency moments have a number of applications.

F_0 represents the number of distinct elements in the streams (which the FM-sketch from last class estimates using O(log n) space. F_1 is the number of elements in the stream m. F_2 is used in database optimisation engines to estimate self join size. Consider the query, "return all pairs of individuals that are in the same location". Such a query has cardinality equal to $\sum_i m_i^2/2$, where m_i is the number of individuals at a location. Depending on the estimated size of the query, the database can decide (without actually evaluating the answer) which query answering strategy is best suited. F_2 is also used to measure the information in a stream.

In general, F_k represents the degree of skew in the data. If F_k/F_0 is large, then there are some values in the domain that repeat more frequently than the rest. Estimating the skew in the data also helps when deciding how to partition data in a distributed system.

2.3.2 Implementation

We revert to the sketch data structure of the Count-Min sketch as the basis of the AMS sketch, to emphasise the relatedness of all these sketch techniques.

The AMS summary maintains an array of counts which are updated with each arriving item. It gives an estimate of the $l2$-norm of the vector v that is induced by the sequence of updates. The estimate is formed by computing the norm of each row, and taking the median of all

rows. Given parameters ϵ and δ, the summary uses space $O(1\epsilon^2 log 1/\delta)$, and guarantees with probability at least $1-\delta$ that its estimate is within relative $\epsilon - error$ of the true $l2$-norm.

Initially, v is taken to be the zero vector. A stream of updates modifies v by specifying an index i to which an update w is applied, setting $v_i \leftarrow v_i + w$. The update weights w can be positive or negative.

The AMS summary is represented as a compact array C of $d \times t$ counters, arranged as d rows of length t. In each row j, a hash function h_j maps the input domain U uniformly to $1, 2, ...t$. A second hash function g_j maps elements from U uniformly onto $-1, +1$. For the analysis to hold, we require that g_j is fourwise independent. That is, over the random choice of g_j from the set of all possible hash functions, the probability that any four distinct items from the domain get mapped to $-1, +1$ is uniform: each of the 16 possible outcomes is equally likely. This can be achieved by using polynomial hash functions of the form $g_j(x) = 2((ax^3 + bx^2 + cx + d \mod p) \mod 2) - 1$, with parameters a, b, c, d chosen uniformly from the prime field p.

The sketch is initialised by picking the hash functions to use, and initialising the array of counters to all zeros. For each update operation to index i with weight w (which can be either positive or negative), the item is mapped to an entry in each row based on the hash functions h, and the update applied to the corresponding counter, multiplied by the corresponding value of g. That is, for each $1 \leq j \leq d$, $h_j(i)$ is computed, and the quantity $wg_j(i)$ is added to entry $C[j, h_j(i)]$ in the sketch array. Processing each update therefore takes time $O(d)$, since each hash function evaluation takes constant time.

The sketch allows an estimate of $\| v \|_2^2$, the squared Euclidean norm of v, to be obtained. This is found by taking the sum of the squares of row of the sketch in turn, and finds the median of these sums. That is, for row j, it computes $\sum_{k=1}^{t} C[j,k]^2$ as an estimate, and takes the median of the d such estimates. The query time is linear in the size of the sketch, $O(td)$, as is the time to initialise a new sketch. Meanwhile, update operations take time $O(d)$.

The analysis of the algorithm follows by considering the produced estimate as a random variable.

The random variable can be shown to be correct in expectation: its expectation is the desired quantity, $\| v \|_2^2$. This can be seen by expanding the expression of the estimator. The resulting expression has terms$\sum_i v_i^2$, but also terms of the form $v_i v_j$ for $i \neq j$. However, these "unwanted terms" are multiplied by either +1 or -1 with equal probability, depending on the choice of the hash function g. Therefore, their expectation is zero, leaving only $\| v \|_2$.

Note that since the updates to the AMS sketch can be positive or negative, it can be used to measure the Euclidean distance between two vectors v and u: we can build an AMS sketch of v and one of $-u$, and merge them together by adding the sketches. Note also that a sketch of $-u$ can be obtained from a sketch of u by negating all the counter values.

2.3.3 Notes on hash functions

Although the terminology of pairwise and four-wise independent hash functions may be unfamiliar, they should not be thought of as exotic or expensive. A family of pairwise independent hash functions is given by the functions $h(x) = ax + b \mod p$ for constants a and b chosen uniformly between 0 and $p1$, where p is a prime. Over the random choice of a and b, the probability that two items collide under the hash function is $1/p$.

Similarly, a family of four-wise independent hash functions is given by $h(x) = ax^3 + bx^2 + cx \mod p$ for a, b, c, d chosen uniformly from $[p]$ with p prime. As such, these hash functions can be computed very quickly, faster even than more familiar cryptographic hash functions such as MD5 or SHA-1.

Consequently, this sketch can be very fast to compute: each update requires only d entries in the sketch to be visited, and a constant amount of hashing work done to apply the update to each visited entry. The depth d is set as $O(log 1/\delta)$, and in practice this is of the order of 10-30, although d can be set as low as 3 or 4 without obvious problem.

2.3.4 Historical notes

Historically, the AMS Sketch was the first to be proposed as such in the literature, in 1996. The "Random Subset Sums" technique can be shown to be equivalent to the AMS sketch for estimating single frequencies. The Count-Sketch idea was presented first in 2002. Crucially, this seems to be the first work where it was shown that hashing items to w buckets could be used instead of taking the mean of w repetitions of a single estimator, and that this obtains the same accuracy. Drawing on this, the Count-Min sketch was the first to obtain a guarantee in $O(1/\epsilon)$ space, albeit for an F_1 instead of F_2 guarantee. Applying the "hashing trick" to the AMS sketch make it very fast to update seems to have been discovered in parallel by several people.

2.3.5 Join size estimation

Given two frequency distributions over the same domain, f and f' , their inner product is defined as:

$$f(i) \cdot f'(i) = \sum_{i=1}^{M} f(i) * f'(i)$$

This has a natural interpretation, as the size of the equi-join between relations where f denotes the frequency distribution of the join attribute in the first, and f' denote the corresponding distribution in the second. In SQL, this is:

SELECT COUNT(*) FROM D, D' WHERE D.id = D'.id

The inner product also has a number of other fundamental interpretations that we shall discuss below. For example, it can be used when each record has an additional "measure" value, and the query is to compute the sum of products of measure values of joining tuples (e.g. finding the total amount of sales given by number of sales of each item multiplied by price of each item). It is also possible to encode the sum of those $f(i)s$ where i meets a certain predicate as an inner product where $f'(i) = 1$ if and only if i meets the predicate, and 0 otherwise.

2.3.6 Inner product estimations

Using the AMS Sketch to estimate inner products. Given AMS sketches of f and f', C and C' respectively, that have been constructed with the same parameters (that is, the same choices of $w, d, h_j and g_j$, the estimate of the inner product is given by

$$f(i) \cdot f'(i) = \sum_{i=1}^{M} f(i) * f'(i)$$

That is, the row estimate is the inner product of the rows. The bounds on the error follow for similar reasons to the F_2. Expanding out the sum shows that the estimate gives $f \cdot f'$, with additional cross-terms due to collisions of items under h_j. The expectation of these cross terms in $f(i)f'(i')$ is zero over the choice of the hash functions, as the function g_j is equally likely to add as to subtract any given term.

2.3.7 Comparing AMS and Count-Min sketches for join size estimation

The analysis shows the worst case performance of the two sketch methods can be bounded in terms of N or F_2. To get a better understanding of their true performance, Dobra and Rusu performed a detailed study of sketch algorithms [6]. They gave a careful statistical analysis of the properties of sketches, and considered a variety of different methods to extract estimates from sketches. From their empirical evaluation of many sketch variations for the purpose of join-size estimation across a variety of data sets, they arrive at the following conclusions:

- The errors from the hashing and averaging variations of the AMS sketches are comparable for low-skew (near-uniform) data, but are dramatically lower for the hashing version (the main version presented in this chapter) when the skew is high.
- The Count-Min sketch does not perform well when the data has low-skew, due to the impact of collisions with many items on the estimation. But it has the best overall

performance when the data is skewed, since the errors in the estimation are relatively much lower.

- The sketches can be implemented to be very efficient: each update to the sketch in their experiments took between 50 and 400 nanoseconds, translating to a processing rate of millions of updates per second.

As a result, the general message seems to be that the (fast) AMS version of the sketches are to be preferred for this kind of estimation, since they exhibit fast updates and accurate estimations across the broadest range of data types.

2.3.8 Scala code implementation

In this chapter we attach the Scala implementation of the AMS Sketch (the very basic components/methods of it).

```
/**
 * return a hash of x using a and b mod (2^31 - 1) may need to do another mod
 * afterwards, or drop high bits depending on d, number of bad guys
 * 2^31 - 1 = 2147483647
 */
public static long hash31(long a, long b, long x) {
    long result = (a * x) + b;
    result = ((result >> HL) + result) & MOD;
    return result;
}
```

```
/**
 * returns values that are 4-wise independent by repeated calls to the
 * pairwise indpendent routine.
 */
public static long fourwise(long a, long b, long c, long d, long x) {
    long result = hash31(hash31(hash31(x,a,b),x,c),x,d);
    return result;
}
```

```
 public boolean add(Long item, long incrementCount) {
    int offset = 0, hash, mult;
    count += incrementCount;
    for (int j=0; j<depth; j++) {
        hash = (int) HashUtils.hash31(test[0][j], test[1][j], item); // Hash using 4-pair wise
        hash = hash % buckets;
        mult = (int) HashUtils.fourwise(test[2][j], test[3][j], test[4][j], test[5][j], item);
        if ((mult&1)==1) // increment or not counter, according to the hashing result
            counts[offset+hash] += incrementCount;
        else
            counts[offset+hash]-= incrementCount;
        offset += buckets;
    }
    return true;
}
```

```
/**
* Estimate the count of an individual item
*/
public long estimateCount(Long item) {
    int offset = 0, hash, mult, estimate;
    int[] estimates = new int[depth+1];
    for (int i=1; i<=depth; i++) {
        hash = (int) HashUtils.hash31(test[0][i-1], test[1][i-1], item);
        hash = hash % buckets; // Hash using 4-pair wise
        mult = (int) HashUtils.fourwise(test[2][i-1], test[3][i-1], test[4][i-1], test[5][i-1], item);
        if ((mult&1)==1)
            estimates[i] = counts[offset+hash];
        else
            estimates[i] = -counts[offset+hash];
```

```
            offset += buckets;
        }
        if (depth == 1)
            estimate = estimates[1];
        else if (depth == 2)
            estimate = (estimates[1]+estimates[2])/2;
        else
            estimate = ArrayUtils.medSelect(1+depth/2, depth, estimates);
        return estimate;
    }
```

```
    /**
    * estimate the F2 moment of the vector (sum of squares)
    */
    public long estimateF2() {
        int r = 0;
        long[] estimates = new long[depth+1];
        long result, z;
        for (int i=1; i<=depth; i++) {
            z=0;
            for (int j=0; j<buckets; j++) {
                z += counts[r] * counts[r]; // Estimate the 2nd frequency moment
                r++;
            }
            estimates[i]=z;
        }
        if (depth == 1)
            result = estimates[1];
        else if (depth == 2)
            result = (estimates[1]+estimates[2])/2;
        else
            result = ArrayUtils.longMedSelect(1+depth/2, depth, estimates);
        return result;
    }
```

Chapter 3

Data processing engines

In this chapter we are presenting the concepts of big data and fast data. We give some real world use cases where fast data is currently used. We describe the frameworks that have been used in fast data problems.

3.1 Big data

3.1.1 What is Big Data

Big data is a term used to describe the collection, processing and availability of huge volumes of streaming data in real-time. The three V's are volume, velocity and variety. Companies are combining marketing, sales, customer data, transactional data, social conversations and even external data like stock prices, weather and news to identify correlation and causation statistically valid models to help them make more accurate decisions.

3.1.2 Why Is Big Data Different

In the old days, a few years ago, we would utilise systems to extract, transform and load data ETL into giant data warehouses that had business intelligence solutions built over them for

reporting. Periodically, all the systems would backup and combine the data into a database where reports could be run and everyone could get insight into what was going on.

The problem was that the database technology simply couldn't handle multiple, continuous streams of data. It couldn't handle the volume of data. It couldn't modify the incoming data in real-time. And reporting tools were lacking that couldn't handle anything but a relational query on the back-end. Big Data solutions offer cloud hosting, highly indexed and optimized data structures, automatic archival and extraction capabilities, and reporting interfaces have been designed to provide more accurate analyses that enable businesses to make better decisions.

Better business decisions mean that companies can reduce the risk of their decisions, and make better decisions that reduce costs and increase marketing and sales effectiveness.

3.1.3 What kind of datasets are considered big data

The uses of big data are almost as varied as they are large. Prominent examples you're probably already familiar with including social media network analyzing their members' data to learn more about them and connect them with content and advertising relevant to their interests, or search engines looking at the relationship between queries and results to give better answers to users' questions.

But the potential uses go much further! Two of the largest sources of data in large quantities are transactional data, including everything from stock prices to bank data to individual merchants' purchase histories; and sensor data, much of it coming from what is commonly referred to as the Internet of Things IoT. This sensor data might be anything from measurements taken from robots on the manufacturing line of an automaker, to location data on a cell phone network, to instantaneous electrical usage in homes and businesses, to passenger boarding information taken on a transit system.

By analyzing this data, organizations are able to learn trends about the data they are measuring, as well as the people generating this data. The hope for this big data analysis are to provide

more customized service and increased efficiencies in whatever industry the data is collected from.

3.2 Fast data

3.2.1 What is Fast Data

Big data is a term used to describe the collection, processing and availability of huge volumes of streaming data in real-time. The three V's are volume, velocity and variety. Companies are combining marketing, sales, customer data, transactional data, social conversations and even external data like stock prices, weather and news to identify correlation and causation statistically valid models to help them make more accurate decisions.

3.2.2 Big Data versus Fast Data

A stream processing solution has to solve different challenges compared to big data:

- Processing massive amounts of streaming events (filter, aggregate, rule, automate, predict, act, monitor, alert)
- Real-time responsiveness to changing market conditions
- Performance and scalability as data volumes increase in size and complexity
- Rapid integration with existing infrastructure and data sources: Input (e.g. market data, user inputs, files, history data from a DWH) and output (e.g. trades, email alerts, dashboards, automated reactions)
- Fast time-to-market for application development and deployment due to quickly changing landscape and requirements

- Developer productivity throughout all stages of the application development lifecycle by offering good tool support and agile development
- Analytics: Live data discovery and monitoring, continuous query processing, automated alerts and reactions
- Community (component / connector exchange, education / discussion, training / certification)
- End-user ad-hoc continuous query access
- Alerting
- Push-based visualization

3.2.3 Real world use cases

Stream processing found its first uses in the finance industry, as stock exchanges moved from floor-based trading to electronic trading. Today, it makes sense in almost every industry - anywhere where you generate stream data through human activities, machine data or sensors data. Assuming it takes off, the Internet of Things will increase volume, variety and velocity of data, leading to a dramatic increase in the applications for stream processing technologies. Some use cases where stream processing can solve business problems include:

- Network monitoring
- Intelligence and surveillance
- Risk management
- E-commerce
- Fraud detection
- Smart order routing

- Transaction cost analysis
- Pricing and analytics
- Market data management
- Algorithmic trading
- Data warehouse augmentation
- Let's discuss one use case in more detail using a real world example.

3.3 Storm

3.3.1 What is storm

Apache Storm is one of the pioneering projects for real-time distributed computing. Incubated by Apache Foundation and open sourced during 2007, Storm received a lot of attraction and was quickly adopted by many organisations.

3.3.2 Key features

Storm is similar to most of the distributed tools we discuss during this dissertation, with regards to the way it manages it's own status. Storm has at least one master node, called Nimbus, and many workers nodes. Coordination for all these nodes *masters and workers* is handled with zookeeper. processing actions are handled by the workers. Processing actions called tasks *chunks of a processing pipeline* get assigned to workers through Nimbus.

Nimbus is responsible for coordinating the tasks to the workers. If a new job is submitted to the cluster, Nimbus will balance the tasks to the workers based on many factors, like load of a node, health of node and data locality. Except the occasions where everything runs smoothly on the cluster, Nimbus also tries to guarantee that the application will still run even after a

failure. For example if a node is not healthy at some point (network issues etc), Nimbus will try to rebalance the load to different nodes to avoid downtime. In order to distribute the workload of a process using Storm, we need to use Storm's API.

It's API is composed by Spouts, Bolts and the topology declaration.

To make things clear we provide an example of a Storm process. Figure 3.1 provides the key components of a Storm topology that reads from Kafka, applies a simple transformation to the input data and writes to another topic in Kafka.

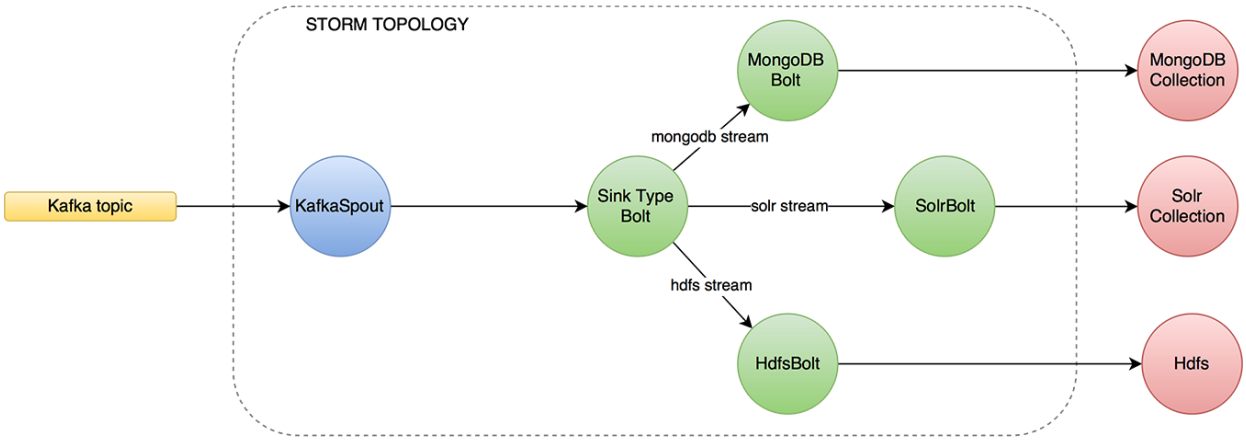

Figure 3.1: Apache storm ETL example (from [8]

As shown on the picture above, workers are assigned tasks. A worker may have one or more tasks. A worker can run on one machine but also many workers can run on multiple machines.

3.4 Akka Streams

3.4.1 What is Akka Streams

Akka-Streams provide a higher-level abstraction over Akka's existing actor model. The Actor model provides an excellent primitive for writing concurrent, scalable software, but it still is a primitive; it's not hard to find a few critiques of the model.

```
Source.single(1).map(_.toString).runWith(Sink.head)

// types:
Source[Int, Unit]
  Flow[Int, String, Unit]
       Sink[String, Future[String]]
```

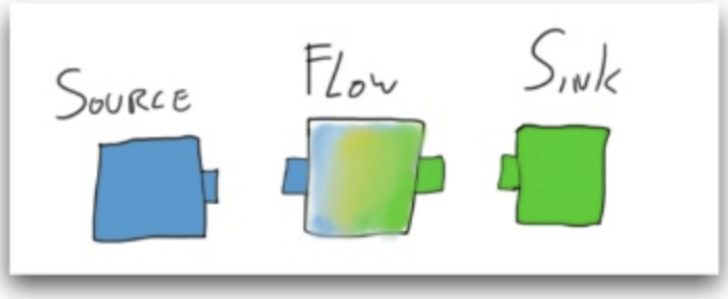

Figure 3.2: Akka Streams ETL example (from [9])

3.4.2 Key features

As mentioned earlier and as shown in 3.2, Akka streams have been designed using a higher level approach to the problem of stream processing. Thus, Akka streams allows data to *flow* within the processing engine. Even though the akka streams API might a seem a bit complex to start with, at the end it compensates the developer, since it removes low level complexity and finally results in writing very compact code blocks.

One point where Akka streams lacks in features compared to the other frameworks we are going to evaluate, is the out of the box *distributed mode*. Since it is a fairly new project, the main efforts have been focused around the API and the parallelisation on a local machine. In order to distribute the workload and the processing across a cluster we need to use some custom configuration, which we are going to go through in the next chapters.

But even though the previous point might seem a huge drawback, there is a huge benefit using Akka streams. The community has put a lot of effort in order to create an API that allows the developer to fine tune the execution of a flow. That is, one can change the parallelisation of the flow which results in taking full advantage of a machine's resources. For example compared

to Spark, the developer cannot control the cpu/thread usage of an app.

3.5 Spark

3.5.1 Overview

Apache Spark is an open source big data processing framework built around speed, ease of use, and sophisticated analytics. It was originally developed in 2009 in UC Berkeley's AMPLab, and open sourced in 2010 as an Apache project.

Spark has several advantages compared to other big data and MapReduce technologies like Hadoop and Storm. It gives a comprehensive, unified framework to manage big data processing requirements with a variety of data sets that are diverse in nature (text data, graph data etc) as well as the source of data (batch v. real-time streaming data).

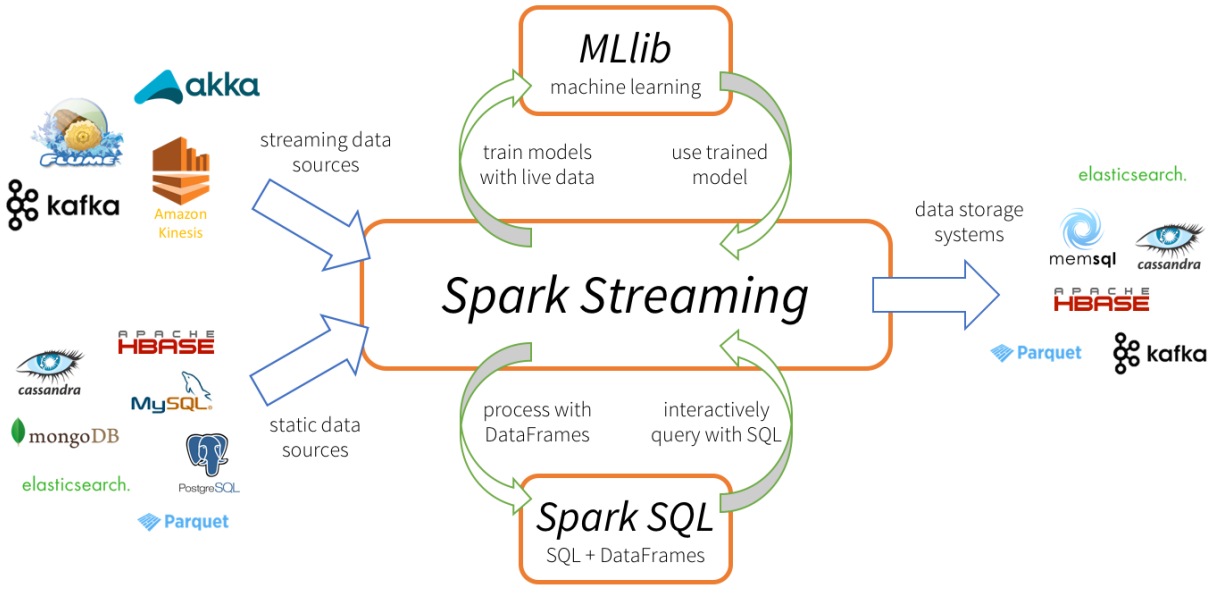

Figure 3.3: Spark ETL example (from [8])

3.5.2 Key features

It exposes APIs for Java, Python, and Scala and consists of Spark core and several related projects:

- Spark SQL - Module for working with structured data. Allows you to seamlessly mix SQL queries with Spark programs.
- Spark Streaming - API that allows you to build scalable fault-tolerant streaming applications.
- MLlib - API that implements common machine learning algorithms.
- GraphX - API for graphs and graph-parallel computation.

Furthermore, since the community has broadly adopted Spark, many submodules have been developed for it. For example, a developer can use existing modules to load data from Kafka or any database but also load data to them. Thus, starting with Spark requires almost no time and zero surprises with regards to extracting/loading data to/from it correspondingly.

Spark versus Storm

As mentioned before Storm was the very first project that allowed scalable processing of real-time input stream and it was the project that triggered the created the notion of distributed processing. Spark community having identified the drawbacks of Storm, developed Spark in a more developer friendly way. They created an easy to use API that allowed the fast adoption by the community.

We are not going to go through every single point of the Storm and Spark at this point but we have to mention that one huge benefit of Spark, is that it allowed data engineers and data scientists to start working on ML algorithms easily. That was succeeded by the creation of the Python API!

3.6 Kafka Streams

3.6.1 Overview

Kafka Streams is a client library for processing and analyzing data stored in Kafka and either write the resulting data back to Kafka or send the final output to an external system. It builds upon important stream processing concepts such as properly distinguishing between event time and processing time, windowing support, and simple yet efficient management of application state. It is based on many concepts already contained in Kafka, such as scaling by partitioning the topics. Also for this reason it comes as a lightweight library, which can be integrated into an application. The application can then be operated as desired: standalone, in an application server, as docker container or via a resource manager such as Kubernetes.

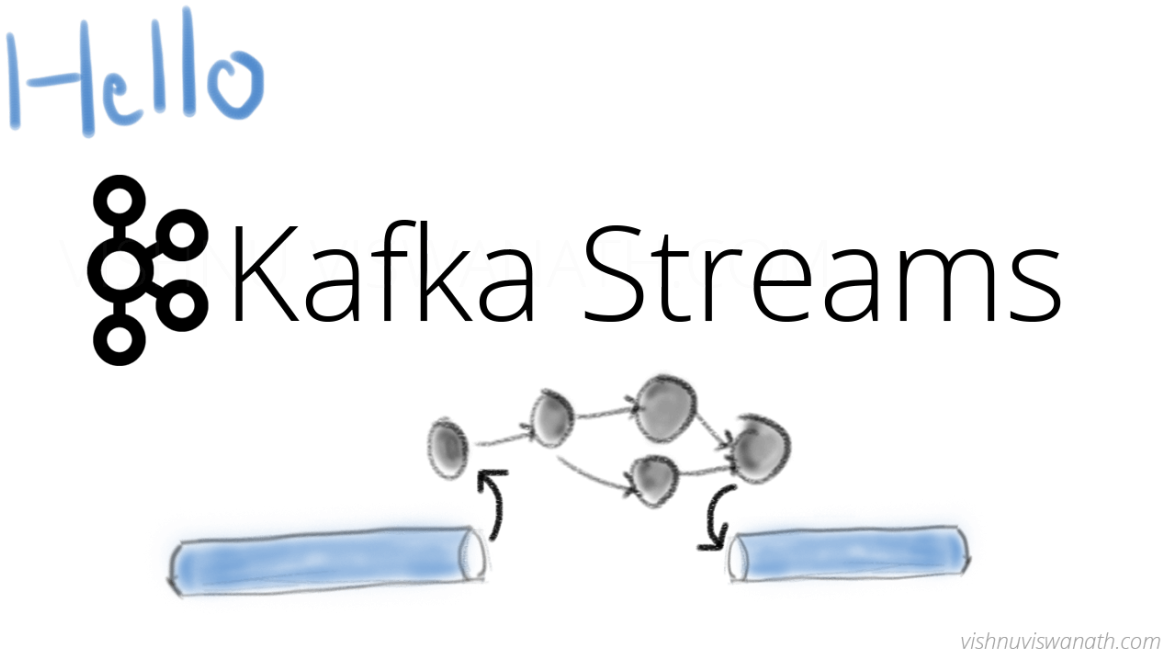

Figure 3.4: Kafka Streams ETL example

3.6.2 Key features

Kafka Streams directly addresses a lot of the difficult problems in stream processing:

- Highly scalable, elastic, distributed, and fault-tolerant application.

- Stateful and stateless processing.

- Event-time processing with windowing, joins, and aggregations.

- We can use the already-defined most common transformation operation using Kafka Streams DSL or the lower-level processor API, which allow us to define and connect custom processors.

- Low barrier to entry, which means it does not take much configuration and setup to run a small scale trial of stream processing; the rest depends on your use case.

- No separate cluster requirements for processing (integrated with Kafka).

- Employs one-record-at-a-time processing to achieve millisecond processing latency, and supports event-time based windowing operations with the late arrival of records.

- Supports Kafka Connect to connect to different applications and databases.

- Supports KTable and GlobalKTable which eases the joining of an input stream with external data.

Kafka streams versus Spark and Storm

At this point we would like to draw the attention to one very strong point of kafka streams compared to other frameworks. The lack of a master or a coordinator node. It might seem not important at the very beginning, but having not to worry about a master node is a big benefit especially when deploying production applications. That means we don't have to setup a custom framework (i.e. Spark) in our machines. It is enough for Kafka Streams to start an instance of each app using the simplest approach (i.e. java -jar myapp.jar) and that is enough for the app to start processing the input stream data.

Furthermore the fact that kafka streams can scale itself using just the consumer groups on each kafka topic, removes the dependency on applications like zookeeper etc. Again, the benefit might not be visible at the very beginning, but since anything can go wrong (i.e. network connections, disks etc) it is always preferable to have as little dependencies as possible.

3.7 Kafka

In this section we are going to provide a quick overview of the Kafka messaging framework. We haver skipped the low level details since in the next chapters it is going to become clear to the reader how kafka is used and what benefits it offers compared to old messaging systems.

3.7.1 What is kafka

Apache Kafka is a distributed publish-subscribe messaging system designed to replace traditional message brokers.

Originally created and developed by LinkedIn, then open sourced in 2011, Kafka is currently developed by the Apache Software Foundation to exploit new data infrastructures made possible by massively parallel commodity clusters.

Message brokers are a type of middleware that translates messages of one language to another, usually more commonly-accepted language. Message brokers can also be used to decouple data

streams from processing and buffer unsent messages. Apache Kafka improves on traditional message brokers through advances in throughput, built-in partitioning, replication, latency and reliability.

Kafka can be used for a number of purposes: Messaging, real time website activity tracking, monitoring operational metrics of distributed applications, log aggregation from numerous servers, event sourcing where state changes in a database are logged and ordered, commit logs where distributed systems sync data and restoring data from failed systems.

3.7.2 Architecture

Kafka stores messages which come from arbitrarily many processes called "producers". The data can thereby be partitioned in different "partitions" within different "topics". Within a partition the messages are indexed and stored together with a timestamp. Other processes called "consumers" can query messages from partitions. Kafka runs on a cluster of one or more servers and the partitions can be distributed across cluster nodes.

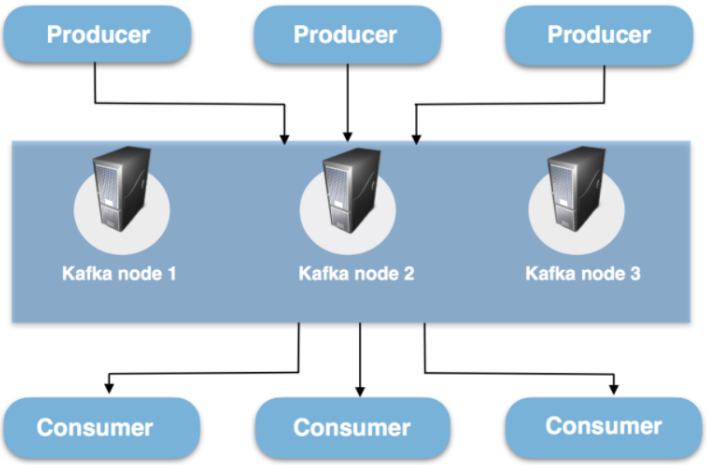

Figure 3.5: Kafka high level view

Apache Kafka efficiently processes the real-time and streaming data when used along with Apache Storm, Apache HBase and Apache Spark. Deployed as a cluster on multiple servers, Kafka handles its entire publish and subscribe messaging system with the help of four APIs, namely, producer API, consumer API, streams API and connector API. Its ability to deliver

massive streams of message in a fault-tolerant fashion has made it replace some of the conventional messaging systems like JMS, AMQP, etc.

The major terms of Kafka's architecture are topics, records, and brokers. Topics consist of stream of records holding different information. On the other hand, Brokers are responsible for replicating the messages. There are four major APIs in Kafka:

Producer API - Permits the applications to publish streams of records. Consumer API - Permits the application to subscribe to the topics and processes the stream of records. Streams API – This API converts the input streams to output and produces the result. Connector API – Executes the reusable producer and consumer APIs that can link the topics to the existing applications.

Chapter 4

Deployments

In this chapter we are going to briefly describe what is a deployment and why we needed to have a well defined pipeline for the purpose of this dissertation.

4.1 Introduction to deployments

4.1.1 How is a deployment defined

Deployment refers to getting your program to a running state on a server. It doesn't need to be the production server. You can deploy an application/module to a testing server that is running on your own workstation or on a testing machine. You might perform many deployments during the development and testing stages of a module or application.

4.1.2 Continuous integration

Continuous integration is a practice that encourages developers to integrate their code into a main branch of a shared repository early and often. Instead of building out features in isolation and integrating them at the end of a development cycle, code is integrated with the shared repository by each developer multiple times throughout the day.

The idea is to minimize the cost of integration by making it an early consideration. Developers can discover conflicts at the boundaries between new and existing code early, while conflicts are still relatively easy to reconcile. Once the conflict is resolved, work can continue with confidence that the new code honors the requirements of the existing codebase.

The end goal of continuous integration is to make integration a simple, repeatable process that is part of the everyday development workflow in order to reduce integration costs and respond to defects early. Working to make sure the system is robust, automated, and fast while cultivating a team culture that encourages frequent iteration and responsiveness to build issues is fundamental to the success of the strategy.

4.1.3 Continuous delivery

Continuous delivery is an extension of continuous integration. It focuses on automating the software delivery process so that teams can easily and confidently deploy their code to production at any time. By ensuring that the codebase is always in a deployable state, releasing software becomes an unremarkable event without complicated ritual. Teams can be confident that they can release whenever they need to without complex coordination or late-stage testing. As with continuous integration, continuous delivery is a practice that requires a mixture of technical and organisational improvements to be effective.

On the technology side, continuous delivery leans heavily on deployment pipelines to automate the testing and deployment processes. A deployment pipeline is an automated system that runs increasingly rigorous test suites against a build as a series of sequential stages. This picks up where continuous integration leaves off, so a reliable continuous integration setup is a prerequisite to implementing continuous delivery.

Continuous delivery is attractive because it automates the steps between checking code into the repository and deciding on whether to release well-tested, functional builds to your production infrastructure. The steps that help assert the quality and correctness of the code are automated, but the final decision about what to release is left in the hands of the organization for maximum

flexibility.

4.1.4 Continuous deployment

Continuous deployment is an extension of continuous delivery that automatically deploys each build that passes the full test cycle. Instead of waiting for a human gatekeeper to decide what and when to deploy to production, a continuous deployment system deploys everything that has successfully traversed the deployment pipeline. Keep in mind that while new code is automatically deployed, techniques exist to activate new features at a later time or for a subset of users.

Continuous deployment also allows organisations to benefit from consistent early feedback. Features can immediately be made available to users and defects or unhelpful implementations can be caught early before the team devotes extensive effort in an unproductive direction. Getting fast feedback that a feature isn't helpful lets the team shift focus rather than sinking more energy into an area with minimal impact.

4.1.5 Why we need the 3 C's

Someone might wonder why we need the 3 C's, even though this dissertation was developed by one individual. Did it worth the effort and the cost of the learning curve. The answer is Yes. It might not be very obvious in the very beginning, but having a well defined CI/CD pipeline means that a developer/researcher has less things to worry. For this specific case, having the CI/CD in a stable and well defined pipelined, allowed us to concentrate on the actual algorithm implementation and benchmarking instead spending time on the tools and the deployments.

4.2 Infrastructure

Since the scope of this dissertation is benchmarking sketch algorithms on top of fast data frameworks, there is no reason going into details about the infrastructure. Instead we are just going to give a high level description so the reader is aware of the underlying implementation.

4.2.1 Docker and Kubernetes

For the purpose of this dissertation we decided to implement the sketch algorithms in scala and java. That allowed us to easily run the algorithms in all the fast data frameworks we analysed in chapter 4. For the deployment of the algorithms, we did not setup anything in our local machines (i.e. no spark standalone cluster or kafka). Instead we used a cluster of virtual machines and on top of them we created a Kubernetes cluster.

Kubernetes, in short, is an open source system for managing clusters of containers. To do this, it provides tools for deploying applications, scaling those application as needed, managing changes to existing containerized applications, and helps you optimize the use of the underlying hardware beneath your containers. Kubernetes is designed to be extensible and fault-tolerant by allowing application components to restart and move across systems as needed.

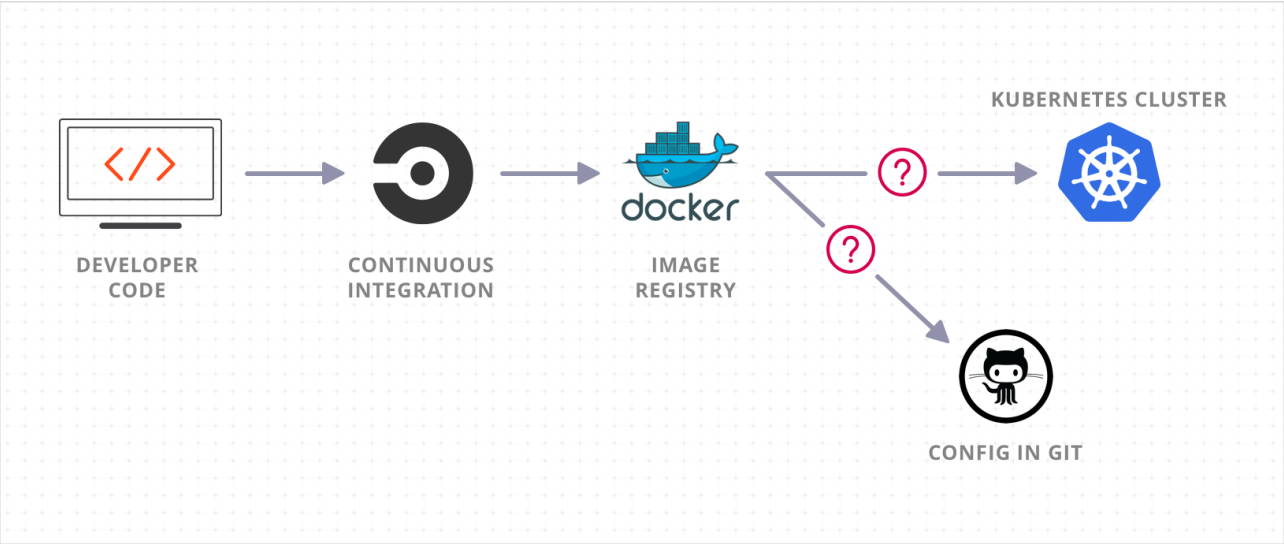

Figure 4.1: Kubernetes example

So by having a Kubernetes cluster for our tests, it allowed us to test many different frameworks with ease but we could also benchmark each algorithm under different conditions. For example we can determine how CMS behaves with less CPU, more memory etc. We are going to describe in details the different configurations in the next chapter.

Chapter 5

Benchmarking

Having described all three sketch algorithms but also all four data frameworks, in this chapter we present the results of each algorithm running on every different framework.

5.1 Background info

Before analyzing the performance of each of the algorithms it is mandatory to describe the setup of the infrastructure we used.

Figure 5.1 shows a high level approach to the setup. The setup consists of:

- Kafka cluster

  We feed the data to the Kafka cluster. For the purpose of this dissertation we have loaded 10M entries to the kafka topic and we force our apps to start reading from earliest offset (that is first entry of the topic)

  Kafka cluster uses 5 brokers in total, each one with 12GB total memory and 4 CPUs

  The input kafka topic topic consists of 10 partitions. That allowed us to run at most 10 sketches in parallel each one reading from one partition.

  The data have been hashed based on the hub_id key. So the data from one hub always end

up in the same partition. This allows us to lower the collisions while at the same time we can process more data.

- Deployment and Parallelisation

  We have created a Kubernetes cluster on which we deployed our apps. We did not have a fixed limit of instances per sketch. Instead we altered the number of running instances many times and the results are presented in the next sections.

  To simplify the analysis for every instance of every sketch we provide the same configuration with regards to memory and cpu. The limits are 4GB of memory and 2 CPUs. That means we just scale up/down the number of running instances(horizontally) instead of the allocated resources (vertically).

- Answering to point queries

  In the case we want to answer point queries (i.e. what is the frequency of $hub_id = 12$) then we just need to hash the hub_id so we get to know to what partition the hub id is mapped to. After that we can direct our query to that specific instance and ask for the frequency.

  Since spark/storm do not have an embedded HTTP Rest server that would allows us to address them, we created a wrapper HTTP service that allows us to directly query them

- Merging instances

  Since all the sketches we are studying support merging it's really easy to implement that functionality. That is from every sketch every sketch, every X seconds we take a snapshot from the sketch and we persist it to a new kafka topic (Check figure 5.1).

  Since we have the latest snapshot from each instance available, we can easily merge them (using the existing functionality of the sketch) and then reply to queries such as the total F2 estimation.

Spark configuration

Spark is the only frameworks, from the ones we examine in this dissertation that it's not completely real time. Instead it has the notion of micro batching. That is it buffer messages

every X seconds and then applies the required transformation and then again it moves to the next batch. To simplify our experiment analysis and to constrain the number of variables that affect our results, we set the micro-batching interval to 5 seconds, for every experiment.

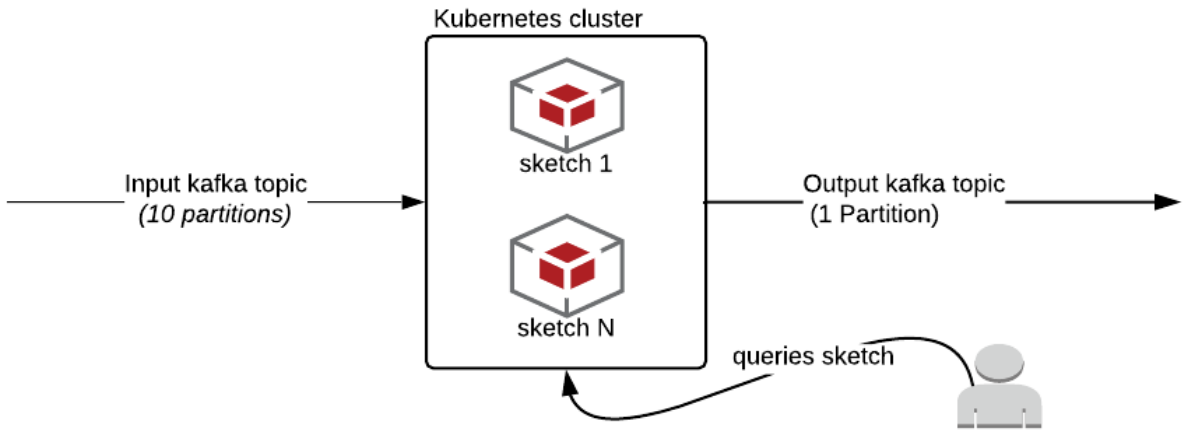

Figure 5.1: Data flow setup

5.2 CMS

Not only for CMS but for every sketch we are studying in this dissertation, there is no difference with regards to quality of the results across running the algorithm in different frameworks. The different results we get, are related to the processing time in each case.

Why there is no difference with regards to quality of the results? This is related to the fact that the sketch algorithm in every framework is exactly the same. Furthermore the input data in each framework are again the same. So it should be obvious now, that if the quality results where different that would indicate a badly written implementation of the sketches.

5.2.1 Time performance Results

In this section we are analysing the results with regards to total processing time of the input stream, for each one of the four frameworks. An aggregated view of the results is given in table 5.1.

CMS time performance results				
	Config A *(1 instance)*	**Config B** *(2 instances)*	**Config C** *(4 instances)*	**Config D** *(10 instances)*
Storm	3K mgs/s (55.5 minutes in total)	6K mgs/s (27.7 minutes in total)	11.8K mgs/s (14.1 minutes in total)	25.5K mgs/s (6.5 minutes in total)
Spark	6K mgs/s (28 minutes in total)	12K mgs/s (13.8 minutes in total)	23.5K mgs/s (7.09 minutes in total)	43K mgs/s (3.9 minutes in total)
Akka	6.5K mgs/s (25.6 minutes in total)	13K mgs/s (12.8 minutes in total)	26K mgs/s (6.41 minutes in total)	48K mgs/s (3.4 minutes in total)
KStreams	6.8K mgs/s (24.5 minutes in total)	13.6K mgs/s (12.2 minutes in total)	27.1K mgs/s (6.15 minutes in total)	51K mgs/s (3.2 minutes in total)

Table 5.1: CMS time performance results

Storm

- Config **A** (*1 instance of the CMS processing the whole input stream*)

  Time to process all data: 55 minutes. That is 3000 entries per second. Note that in this case we utilised only one CMS instance. Of course, this will result to having more collisions since all the data (10B entries) would be processed by the same ECM instance.

- Config **B** (*10 instances of the CMS processing the whole input stream*)

  Time to process all data: 6.5 minutes. Even though we provided x10 times the initial resources we did not manage to process the input stream in 1/10th of the initial time (that would be 2 minutes). This happens because there is an initial cost to i.e. setup the sketch, start consumers etc, that cannot be avoided at any cost.

Spark

- Config **A** (*1 instance/worker of the CMS processing the whole input stream*)

  Time to process all data: 28 minutes. That is 6000 entries per second (almost twice the performance of Storm for 1 instance setup)

- Config **B** (*10 instances/workers of the CMS processing the whole input stream*)

  Time to process all data: 3.9 minutes. Again, even though we provided x10 times the initial resources we did not manage to process the input stream in 1/10th of the initial time (3 minutes). This happens because there is an initial cost to i.e. setup the sketch,

start consumers, workers etc, that cannot be avoided at any cost.

As expected, the results in the case of Spark are better than the Storm implementation. That is related to how Storm has been architected. Recall that in Storm a topology is composed of spouts and bolts and has a dependency on Nimbus and zookeeper. All these dependencies add some overhead to each processed event enough to make a difference at the end. But the part that adds the most significant overhead in this experiment is that data being moved from spouts to bolts which results into intermediate data being written and then read from the disk.

Akka

- Config **A** (*1 instance/actor of the CMS processing the whole input stream*)
  Time to process all data: 25.6 minutes.

- Config **B** (*10 instances/actors of the CMS processing the whole input stream*)
  Time to process all data: 3.4 minutes.

Using this config and Akka Actors there is an interesting case we should point out. As discussed previously Spark is based on AKKA actors. Then why in this case Spark seems slower than AKKA streams provided Spark is written using Akka Actors. The answer is simple. Spark offers more guarantees while processing the stream than Akka streams. For example it offers back pressure (Add a refenrece here), exception recovery, monitoring etc. For each of these guarantees a small overhead is added enough to make a difference!

Kafka Streams

- Config **A** (*1 instance of the CMS processing the whole input stream*)
  Time to process all data: 24.5 minutes.

- Config **B** (*10 instances of the CMS processing the whole input stream*)
  Time to process all data: 3.2 minutes.

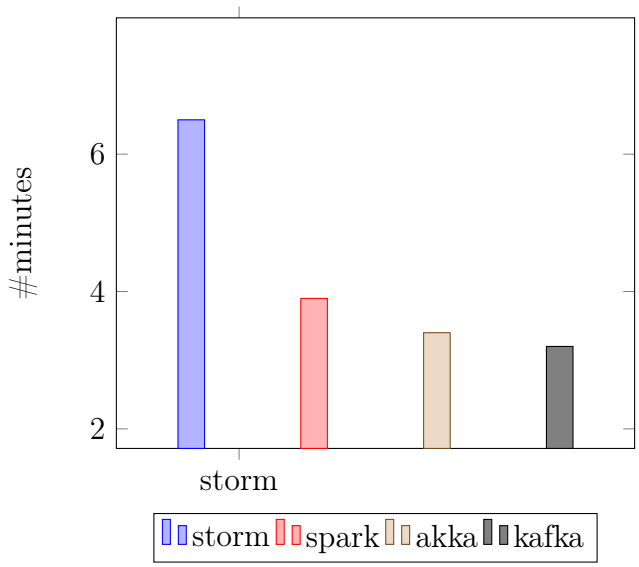

Figure 5.2: CMS time performance in different frameworks (config B)

It seems that kafka streams offers the best possible results (along with Akka Streams) compared to the other frameworks examined above. That is because kafka streams is one of the most lightweight framework. It has zero dependencies to external tools (that is no zookeeper is needed). Furthermore the fact that it is just a library you embedded in a project but also the fact that is has a simple and clean API makes itself an attractive tool to work with

5.3 ECMS

In this section we measure the performance of ECM sketch on the four frameworks. It is expected that the performance of ECMS to be worst than ECM because of the multiple operations it performs compared to CMS

5.3.1 Time performance Results

Again, we are only analysing the results with regards to total processing time of the input stream, for each one of the four frameworks. An aggregated view of the results is given in table 5.2.

ECMS time performance results				
	Config A *(1 instance)*	**Config B** *(2 instances)*	**Config C** *(4 instances)*	**Config D** *(10 instances)*
Storm	2.6K mgs/s (64.1 minutes in total)	5.2K mgs/s (32 minutes in total)	10.5K mgs/s (15.8 minutes in total)	22K mgs/s (7.5 minutes in total)
Spark	5.5K mgs/s (30.3 minutes in total)	11K mgs/s (15.1 minutes in total)	21.8K mgs/s (7.6 minutes in total)	38K mgs/s (4.4 minutes in total)
Akka	5.9K mgs/s (28.3 minutes in total)	11.9K mgs/s (14 minutes in total)	23.9K mgs/s (6.9 minutes in total)	43K mgs/s (3.8 minutes in total)
KStreams	6K mgs/s (27.5 minutes in total)	12 mgs/s (13.8 minutes in total)	24K mgs/s (6.9 minutes in total)	47K mgs/s (3.5 minutes in total)

Table 5.2: ECMS time performance results

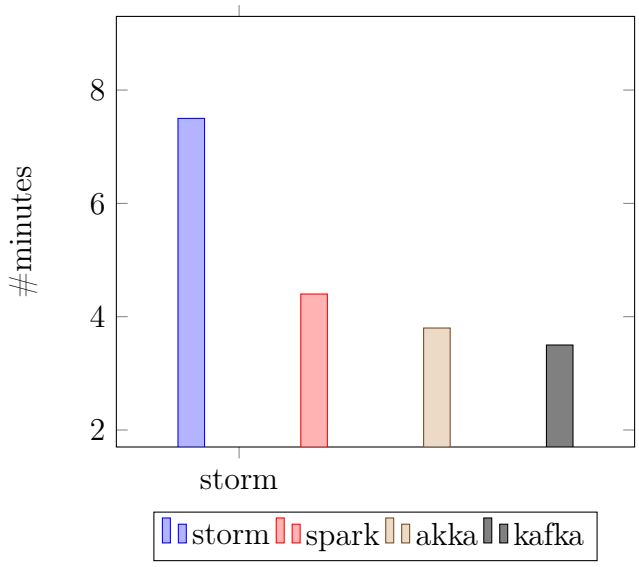

Figure 5.3: CMS time performance in different frameworks (config B)

As expected once again, it takes more time to process the stream with ECMS instead of CMS. That is due to the more operations that take place when an $update(i)$ is performed for the i_th element. Remember that ECMS adds/removes/merges buckets on every update. Of course by using ECMS we have the ability to introduce sliding windows in our queries.

Why not use the window operations that are provided by spark and kafka streams? After having analysed the ECMS algorithm it is easy to answer the question. The reason is that on every update ECMS decides whether to expire data on some specific rules. And spark and kafka streams frameworks cannot *understand* those rules (e.g. bucket size) in order to apply them internally to the CMS structure.

AMS time performance results				
	Config A *(1 instance)*	**Config B** *(2 instances)*	**Config C** *(4 instances)*	**Config D** *(10 instances)*
Storm	3K mgs/s (55.5 minutes in total)	6K mgs/s (27.7 minutes in total)	11.8K mgs/s (14.1 minutes in total)	25.5K mgs/s (6.5 minutes in total)
Spark	6K mgs/s (28 minutes in total)	12K mgs/s (13.8 minutes in total)	23.5K mgs/s (7.09 minutes in total)	43K mgs/s (3.9 minutes in total)
Akka	6.5K mgs/s (25.6 minutes in total)	13K mgs/s (12.8 minutes in total)	26K mgs/s (6.41 minutes in total)	48K mgs/s (3.4 minutes in total)
KStreams	6.8K mgs/s (24.5 minutes in total)	13.6K mgs/s (12.2 minutes in total)	27.1K mgs/s (6.15 minutes in total)	51K mgs/s (3.2 minutes in total)

Table 5.3: AMS time performance results

5.4 AMS

In this section we measure the performance of AMS sketch on the four frameworks. It is expected that the performance of AMS to similar to CMS if not the same

5.4.1 Time performance Results

Again, we are only analysing the results with regards to total processing time of the input stream, for each one of the four frameworks. An aggregated view of the results is given in table 5.3.

Based on the table 5.3 our assumptions were correct. The results are similar with the count-min sketch implementation. That is related to the fact that AMS algorithm has similar implementation to the CMS with the only major difference that AMS hashes the input element using 2 functions twice. So the overhead added is not significant and can only be observed in really large datasets after processing them for a significant amount of time.

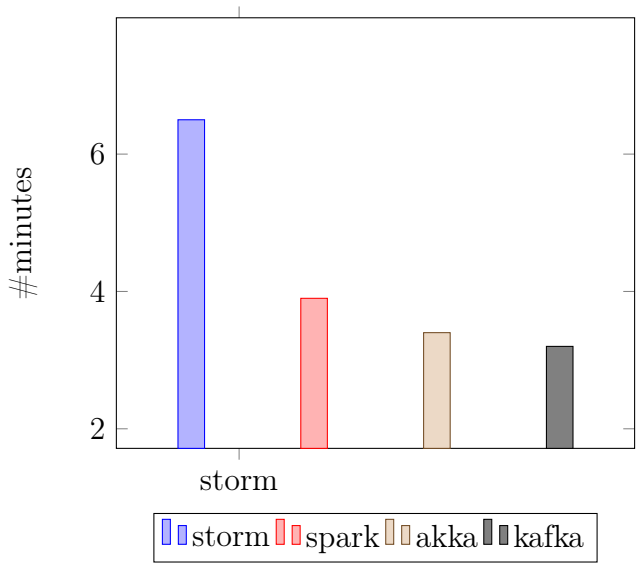

Figure 5.4: AMS time performance in different frameworks (config B)

Chapter 6

Conclusion

6.1 Summary of Dissertation Achievements

During the current dissertation we evaluated three algorithms running on four different data processing frameworks. Our analysis of the results shows that the four frameworks can have significant impact on the total processing time. As discussed previously this impact is related mostly to the guarantees each framework provides (i.e. exactly once delivery, error recovery, state recovery).

With regards to the algorithm development and portability of the algorithms, we did not face any significant problems since all the sketches supported the ‘merge‘ operator. That allowed us to easily fade out our processing phase (distribute the workload in many parallel units) and later merge the results from all the instances (scatter, gather pattern) in a seam

So, our conclusion on the choice of framework, it is up to the architectural requirements

- Portability of algos - Diff result tming accross frameworks

- because of the design of tje algos. distrubution was easy (due to ++ merge functions)

Bibliography

[1] Bloom, B. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM, 13 (7). 422-426.*

[2] Guillaume Pitel, Geoffroy Fouquier Count-Min-Log sketch: Approximately counting with approximate counters. *17 Feb 2015*

[3] O. Papapetrou, M. Garofalakis and A. Deligiannakis. Sketching distributed sliding-window data streams. *The VLDB Journal, 24(3), 2015.*

[4] Cormode, G., Yi, K.: Tracking distributed aggregates over timebased sliding windows. *In SSDBM, pp. 416–430 (2012).*

[5] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *In ACM Symposium on Theory of Computing, 1996.*

[6] A. Dobra and F. Rusu. Statistical analysis of sketch estimators. *ACM Transactions on Database Systems, 33(3), 2008.*

[7] Graham Cormode and Minos Garofalakis. Approximate Continuous Querying over Distributed Streams. *ACM Transactions on Database System, 33(2) 2008.*

[8] Tathagata Das. Spark Streaming description. *https://www.datanami.com/2015/11/30/spark-streaming-what-is-it-and-whos-using-it/.*

[9] Akka streams. *https://www.slideshare.net/ktoso/reactive-integrations-with-akka-streams.*