Independent degree project –
second cycle

*Master's thesis MA, 30 credits*

Master of Science in Computer Engineering

**TupleSearch**
A scalable framework based on sketches to process and store streaming temporal data for real time analytics

Henrik Karlsson
TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics
Henrik Karlsson

MID SWEDEN UNIVERSITY
The Department of Information and Communication Systems (IKS)

Examiner: Prof. Tingting Zhang, <Tingting.Zhang@miun.se>
Supervisor: Magnus Eriksson, <Magnus.Eriksson@miun.se>
Author: Henrik Karlsson, <Heka1203@student.miun.se>
Degree programme: Master of Science in Computer Engineering, 300 credits
Main field of study: Computer Engineering
Semester, year: Spring, 2017
Abstract

In many fields, there is a need for quick analysis of data. As the number of devices connected to the Internet grows, so does the amounts of data generated. The traditional way of analyzing large amounts of data has been by using batch processing, where the already collected data is processed. This process is time consuming, resulting in another trend emerging: stream processing. Stream processing is when data is processed and stored as it arrives. Because of the velocity, volume and variations in data, Stream processing is best carried out in the main memory, and means processing and storing data as it arrives, which makes it a big challenge. This thesis focuses on developing a framework for the processing and storing of streaming temporal data enabling the data to be analyzed in real time. For this purpose, a server application was created consisting of approximate in-memory data synopsizes, called sketches, to process and store the input data. Furthermore, a client web application was created to query and analyze the data. The results show that the framework can support simple aggregate queries with constant query time regardless to the volume of data. Also, it can process data 6.8 times faster than a traditional database system. All this implies that the system is scalable, at the same time it with a query error vs. memory trade-off. For a distribution of ~3000000 unique items it was concluded that the framework can provide very accurate answers, with an error rate less than 1.1%, for the trendiest data using about 100 times less space than the actual size of the data set.

Keywords: Streaming Data, Stream Processing, Count-Min Sketch, Time Adaptive Sketches
Acknowledgements

I would like to thank Björn Lindström along with others from Knowit for providing me with an authentic data set and for showing me interest throughout my work on the thesis. I would also like to thank my supervisor Magnus Eriksson for his input on the report. Finally, I would like to thank my examiner Tingting Zhang for her guidance within this field of research.
Table of contents

1 Introduction ........................................................................................................1
  1.1 Background and problem motivation .........................................................1
  1.2 Overall aim .....................................................................................................2
  1.3 Scope .............................................................................................................2
  1.4 Concrete and verifiable goals .........................................................................3
  1.5 Outline ...........................................................................................................4

2 Batch processing and Stream processing .........................................................5

3 Sketches overview ..............................................................................................8
  3.1 Introduction to sketches ................................................................................8
  3.2 Registers .......................................................................................................8
  3.3 Hash functions ...............................................................................................9
  3.4 Hash collisions .............................................................................................9
  3.5 Universal hash functions ..............................................................................9
  3.6 The Bloom filter .........................................................................................10
  3.7 Count-Min Sketch and its different versions ...............................................11
    3.7.1 Count-Min Sketch ................................................................................11
    3.7.2 Count-Min Sketch with conservative updates .....................................15
    3.7.3 Time adaptive Count-Min Sketch .......................................................15
  3.8 Clarification ..................................................................................................16

4 Web standards, formats and frameworks .........................................................17
  4.1 Ajax application ............................................................................................17
  4.2 JSON ...........................................................................................................17
  4.3 REST ............................................................................................................17
  4.4 Document Object Model .............................................................................19
  4.5 JavaScript ....................................................................................................20
  4.6 ECMAScript ................................................................................................20
  4.7 TypeScript ....................................................................................................21
  4.8 React ............................................................................................................21
  4.9 WebSocket Protocol .....................................................................................22
  4.10 MongoDB ..................................................................................................23
5 Methodology.................................................................24
  5.1 Motivation for data streaming algorithms .........................24
  5.2 Development of proof-of-concept application .......................26
  5.3 Test cases ..................................................................27
  5.4 Measurements ............................................................27
    5.4.1 Experimental setup .................................................27
    5.4.2 Query error ............................................................28
    5.4.3 Query latency ..........................................................30
    5.4.4 Insertion time ..........................................................31
6 Implementation...................................................................32
  6.1 Application flow .........................................................32
  6.2 REST API ....................................................................33
  6.3 In-memory processing, storage and querying .......................37
    6.3.1 Point and range update .............................................37
    6.3.2 Point and range query ..............................................42
    6.3.3 Top-k update ............................................................43
    6.3.4 Top-k query ..............................................................47
    6.3.5 Subscribe service .....................................................48
  6.4 Client application .........................................................49
7 Results..............................................................................53
  7.1 Query error ..................................................................53
  7.2 Query latency ...............................................................56
  7.3 Insertion time ...............................................................57
8 Conclusion.........................................................................59
1 Introduction

1.1 Background and problem motivation

There exists a class of systems and devices that generates and communicates data in a continuous manner, collectively called streaming data. Streaming data [1] includes financial data generated by customers via ecommerce purchases as well as sensor data from connected devices. Due to the continuous flow of data, it grows quickly in volume, but also changes in structure. As time passes it becomes more and more complex to analyze the data and we eventually end up with what is called big data.

The traditional method for dealing with a large volume of data is using batch processing. Batch processing [1] is when arbitrary queries are performed on large batches of data. This typically means a latency of several minutes up to several hours and the process is performed offline at a data warehouse. Consequently, a more complex data analysis can be conducted.

Telia is a mobile operator with several systems generating transactional data, including purchases made by customers online as well as at retailers. The data volume is about 100 million transactions per year and the current method for analyzing the data is using batch processing. However, in many fields of applications simple response functions and simple aggregates queries on the data are requested, such as frequency estimations or other statistical information. Furthermore, the query latencies are expected to be in the order of milliseconds so that they can be used for real-time analytics. Providing such solutions over a continuous stream of data is a big challenge. In fact, "Scaling Up for High Dimensional Data and High Speed Data Streams" is considered one of two major challenges of data mining research [2]. In the mining of data streams, there are three major challenges [3]: volume, velocity and volatility. Volume and velocity requires the data to be processed sequentially and incrementally over sliding time windows and with limited space, whereas volatility corresponds to an environment with a drift in the data patterns over time thus old data is of limited use.

The aim of this thesis is to develop a system that support real-time query latencies for simple aggregate queries on a continuous input of temporal data. For this purpose, traditional database solutions are not
TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics
Henrik Karlsson 2017-05-24

suitable. Another way to go about it is to use efficient in-memory storage, which is what is used here, alongside techniques that are referred to as sketches [4]. A sketch is a synopsis data structure which is a summarized version of the whole data set and resides in the main memory.

An application of this type can be beneficial in any area where data is continuously generated and where real-time analytics are of interest. As an example, it can be used to: discover anomalies or similarities or detect trends and examine the outcome of recent activities.

1.2 Overall aim
The aim of this thesis is to examine existing methods for dealing with streaming data. Furthermore, the aim is to apply these methods and develop a general solution that can process and store a continuous flow of data, in a way, so that simple queries on the data can be performed with latencies in the order of milliseconds. Finally, the aim is to provide a way for the end user to visualize the data in a meaningful way.

The thesis aims to provide information regarding data structures and algorithms within the field of streaming data, and to contribute with knowledge on how such data structures and algorithms can be extended and applied to new fields.

1.3 Scope
The scope of the proposed solution is limited to support simple aggregate queries for temporal data, as opposed to a traditional database system that can support more complex queries.

The practical part aims to apply and extend the sketching techniques to handle the problem with the steadily growing data and to provide a solution that can preprocess the data and store it in such a way that simple queries can be performed. As a result, query latencies in the order of milliseconds is expected.

The theoretical part aims at comparing the proposed solution with a traditional relational database system. The comparison is from a performance perspective, where the parameters of interest are: insertion time, query error and query latency.
1.4 Concrete and verifiable goals

The concrete and verifiable goals are:

1. Give an introduction and overview of sketching techniques for streaming data.

2. Apply and extend sketching techniques, to the problem of streaming temporal data and show its use on a real-world dataset.

3. A server application that can process a continuous flow of temporal data and store it in-memory to support real-time queries on the stream of data with approximate statistical answers.

4. A general solution that is applicable on temporal data with various fields that are user-defined.

5. Client web application for querying and visualizing the data.

6. Examine the performance of the proposed solution compared to a traditional database systems in terms of query latency and insertion time.

7. Examine how large estimation errors that can be expected in practice with the approximate data structures, i.e. to what extent it deviates from the true value.
1.5 Outline

The report is organized as follows.

Chapter 2 reviews the traditional way of dealing with large amounts of data as well as in what direction the trend is moving.

Chapter 3 provides an overview of sketching techniques, what they are used for and how they differ from each other.

Chapter 4 reviews key terms related to web applications.

Chapter 5 presents the evaluation method used to meet the verifiable and concrete goals.

Chapter 6 describes the implementation of the framework in detail.

Chapter 7 presents the findings of the work.

Chapter 8 provides conclusions and discusses possible extensions for future work.
2 Batch processing and Stream processing

According to Cisco [5], the Internet is expected to reach 50 billion devices in 2020. Many of the devices will be smart devices that will be generating a vast amount of data. The term big data has been around for many years now and refers to datasets that are too large to process and manage using traditional relational database systems; hence, we need other solutions to cope with the problems of big data. Up until a few years back, the proposed solution for the processing of big data was distributed batch processing [6].

A commonly used tool to speed up the batch processing process, which could take anything from hours up to days, is the MapReduce framework [6]. MapReduce was first announced by Google, but there are open source implementations such as Hadoop from Apache. MapReduce is at its core a programming model that can provide high throughput batch processing for time-consuming jobs. This is done by distributing the job in parallel on several machines. It has support for two computational functions called map and reduce. MapReduce takes a set of key-value pairs as input and the same is the output. Map is the name of the high-order function that applies a given function to an input pair, which results in a set of immediate pairs. Reduce is the name of the high-order function that takes a key and a list of associated values to that key as input and outputs a set of immediate key-value pairs. The execution of a MapReduce follows the same procedure. First, each of the input pairs is given to the map function, which applies the function to transform the input to another set of pairs. Then, in the second phase, each value with a common key is compiled into a list. The keys and their corresponding list are fed into the reduce function, which transforms the input to another set of key-value pairs. A straightforward MapReduce job is counting the words of a given document. The procedure can be described using the pseudocode in Figure 1. The distribution of work among the machines is usually implemented as a master and slave architecture, where the master machine assigns tasks to each slave machine. This is possible because the input is stored as a distributed file system, that is, the input can be accessed by each machine in the system.
TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics
Henrik Karlsson

Algorithm 1 Count each word occurrence

Require: key: document name
Require: value: content
1: procedure MAP(key, value)
2: for each word w ∈ value do
3: EmitINTERMEDIATE(w, 1)
4: end for
5: end procedure

Algorithm 2 Reduce to count per word

Require: key: a word
Require: values: a list of counts for that word
1: procedure REDUCE(key, values)
2: c ← 0
3: for each value v ∈ values do
4: c ← c + v
5: end for
6: Emit(c)
7: end procedure

Figure 1: MapReduce procedure for counting the words of a document. Inspired by [7].

Although MapReduce reduces the computational time significantly for batch processing, it was neither created for nor suitable for stream processing [6]. This because all input must be present before any computation can begin. Also, the architecture requires time for job startup and coordination between the machines. Thus, MapReduce cannot process the input data as if it arrived in a stream, which is typically the case. The flaws of the MapReduce implementation mentioned have led to another trend called stream processing.

A data stream [8] can be thought of as a continuous stream of data tuples, where typical examples are: click streams, message streams and event streams. The idea is to process each tuple of data immediately as it arrives in the stream, where the tradeoff becomes high per tuple processing overhead. The result is that since the data is already processed, we can expect low latencies since no extra processing is needed when data is queried for. An alternative to the architecture of MapReduce, is memory computing [6], where the main memory of one or multiple machines are used to store and process the data in real-time. The main memory can deliver faster write speeds and lower access times, than writing and reading to the file system on a hard-drive. For instance, the main memory can provide access times in the order of nanoseconds whereas a hard drive can provide
it in the order of milliseconds. Imagine combining in-memory computing with efficient data structures in a streaming fashion - we would not only be able to process the data in real-time, but also to access the data with low latencies. It is these types of techniques that are focused on in this thesis. As previously mentioned, there are several data structures within the research, collectively called sketches, that are suitable for this purpose. Some of these structures are explained in the following chapter.
3 Sketches overview

In this chapter, the theory behind the data streaming algorithms applied in this thesis is described. The primary building blocks of such algorithms are described in section 3.1—3.6, followed by a description of the Count-Min Sketch and its versions in section 3.7.

3.1 Introduction to sketches

Frequency estimation are common in data analytics. A frequency estimation basically means: "How many times has x appeared?" For a few number of specific observations, it can be easily computed using for example, a key-value store. But when the number of distinct elements increases, it becomes an expensive task [5]. The memory required to store the data, using for instance a hash map, will grow linearly with the number of unique observations. Consequently, a machine will run out of main memory fast. Efforts has been made to solve these problems, where several algorithms, collectively called sketches, have been developed. The main characteristics of sketches are [9]:

- Constant time updates
- The storage space needed is independent of the amount of data
- Querying runs at worst linear time
- Approximate answers

A sketch [10] creates a compact summary of the data that is being observed that is significantly smaller than the data as a whole. Each new observation in the stream of data means a modification of the sketch, resulting in that queries can be asked with approximate answers. Therefore, a sketch can be thought of as an approximate frequency histogram for many different observations. This thesis investigates linear sketches, that is, sketches that can be represented as a linear transform of the input and preserve important operations although the subspace is minimized.

3.2 Registers

Sketches are built upon registers. A register [9] is an array of indices that holds the data for the sketch. Its storage space does not depend on the
number of different elements and takes up constant space. The register consists of a number of indexes equal to the size of the sketch and where the indexes can be altered. For instance, each index could be a counter that is incremented or decremented, when an event occurs.

### 3.3 Hash functions

A hash function [9] is a function that takes an input of arbitrary length and tries to output a uniformly distributed value in a finite space of output values. For instance, a 16-bit hash function takes a string and produces an output of length 16-bit, meaning there are $2^{16}$ unique outputs. An important characteristic of a hash function is that it is deterministic, that is, the same input always yields the same output. Since streaming applications require quick computations, hash functions with outputs between 32-bit and 128-bit are commonly used. A few of the hash functions used by sketching algorithms are the FNV hash, the Jenkins hash and the MurmurHash. The hash functions are not used in a cryptographic way, but rather to compress information.

### 3.4 Hash collisions

Hash collisions [9] occur when a hash function produces the same output value for two different inputs. Consider an 8-bit hash function which maps every input to one of $2^8$ possible output values. What happens when the number of unique inputs exceeds $2^8$? There will be at least two inputs with the same output value, in other words, there will be at least one hash collision.

### 3.5 Universal hash functions

Most sketching algorithms require several hash functions, because hash collisions can be reduced by using several independent or pairwise independent hash functions. Hash functions with these properties are difficult to produce and it is more common to use a universal hash function [9]. A universal hash function is selected from the same family of hash functions at random and can be created in two ways.

One way to generate $d$ universal hash functions is to select $d$ values uniformly from the possible space of initial values [9]. In this case, the $d^{th}$ hash function uses the $d^{th}$ value as input to generate the hash. Another way to generate $d$ universal hash functions is to use an initial seed value to initialize the hash function. Once, the first hash value is produced, it can be used to produce the next hash function and so on.
### 3.6 The Bloom filter

The Bloom filter [11] was invented in the 1970’s and has been frequently used in database applications. A Bloom filter is a space-efficient data structure used to answer membership queries like: "Is x included in the set?" The tradeoff for being space-efficient is that false positives are introduced due to hash collisions. To represent a set \( S = \{i_1, i_2, \ldots, i_n\} \) of \( n \) items, the Bloom filter is represented as a 1-bit register of length \( w \), see Figure 2. For each insertion of a new item \( i \) to the Bloom filter, the sequence of independent hash functions: \( h_1, \ldots, h_{d-1}, h_d \) are calculated, which outputs a value within the range \( 1 \ldots w \). For each output, which is an index in the register, the corresponding bit in the register is set to 1 as in Figure 3. To check for the existence of item \( i \) in the Bloom filter, \( i \) is hashed \( d \) times. If all corresponding bits are set to 1, the item is a member of the Bloom filter. To identify an item as missing from the Bloom filter, at least one of the bits must be zero, as in Figure 4. Consequently, the memory space used by a Bloom filter is \( O(w) \).

The probability that a hash collision occurs, is directly related to \( n, w \) and \( d \). If \( n \) and \( w \) are given fixed bounds, an optimal choice of \( d \) can be made. This is under the assumption that the hash function produces uniformly distributed hashes and that the hash functions are independent of each other. The probability that any entry within the sketch is zero, after \( n \) distinct observations has been made, is given by:

\[
p = (1 - \frac{1}{w})^{dn}
\]  

(1)

This comes from the fact that each of the \( d \cdot n \) hash outputs has probability \( 1 - \frac{1}{w} \) of leaving an index unset. A false positive occurs when an item that is missing from the Bloom filter, hashes to indices in the bloom filter that were all set to 1. This has a probability of: \( (1 - p)^d \), where \( p \) is the fraction of indices in the filter that is currently set to 0. The idea is that small values of \( d \) will keep the number of 1s lower, but increase the probability of hash collisions, whereas large values of \( d \) will increase the number of 1s, thus, reducing the probability of hash collisions.

![Initial state of Bloom filter](image)

**Figure 2: Initial state of Bloom filter.**
TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics
Henrik Karlsson
2017-05-24

Figure 3: Insertion of an item into the Bloom filter.

Figure 4: Membership estimation of item i1 which can be found in the Bloom filter, whereas i2 is not.

3.7 Count-Min Sketch and its different versions
In this section, the baseline Count-Min Sketch (CMS) is described. This is followed by a description of the Count-Min Sketch with conservative updates and the time adaptive Count-Min Sketch.

3.7.1 Count-Min Sketch
The key idea of the CMS [12] is to summarize frequency information in a stream of data, where there is a larger number of distinct. The updates are stored in a vector $a$ of numbers. In comparison to other sketching algorithms, the CMS allows several queries using the same data structure. The CMS, as it is, has support for one query, the point query, but can be extended to support other queries, such as the range query and the inner product query. Only the point query and range query are used in this study and can be defined as follows:

- A point query $Q(i)$ is the estimated value of the $i$th component in $a$
- A range query $Q(j, k)$ is the estimated value of $\sum_{i=j}^{k} a_i$

Essentially, the CMS is a two-dimensional array with depth $d$ and width $w$ as shown in Figure 5. Each row is associated with a pairwise independent hash function, hence there are $d$ pairwise independent hash functions such that $h_1 \ldots h_d : \{1 \ldots n\} \rightarrow \{1 \ldots w\}$. 

\begin{align*}
\text{Figure 5: Two-dimensional array with depth } d \text{ and width } w.
\end{align*}
TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics
Henrik Karlsson 2017-05-24

Figure 5: The Count-Min Sketch structure.

For simplicity, let the update operation be denoted as \( U(i, c) \), which means an update of the \( i \)th component with value \( c \). When a new item \( i \) arrives in the stream, \( h_{m\in\{1...d\}}(i) \) is calculated for each row of the CMS, and for each output within the range \( 1 \ldots w \) the corresponding index is incremented by \( c \) as in Figure 6. Since the width of each row is expected to be small in comparison with the number of unique elements in the stream, hash collisions will occur at some point and as with the Bloom filter each row in the sketch has a one-sided error bound.

Figure 6: Update for an item in the Count-Min Sketch.

Since the CMS is an approximate data structure, each query to it will include an estimation error. The estimation error is commonly denoted with the two parameters: \( \varepsilon \) and \( \delta \). Where \( \varepsilon \) is the relative error and \( \delta \) is the margin of error. The error in answering a query can be expected to be within a factor of \( \varepsilon \) with probability \( 1 - \delta \) hence the width and depth can be derived as follows:

\[
 w = \left\lceil \frac{\varepsilon}{\varepsilon} \right\rceil \tag{2}
\]
\[
 d = \left\lceil \log \left( \frac{1}{\delta} \right) \right\rceil \tag{3}
\]

This implies that if the width and/or depth increases, the error bound decreases.
A point query, is the frequency estimation of element $i$ and is done by hashing $i$ with the corresponding hash function for each row. Subsequently, the corresponding value on each row is looked up and the minimum of these values is returned as an estimate $\hat{a}_i$, see Figure 7. Now, suppose we want to estimate the number of times an item has occurred in the stream so far. With probability $1 - \delta$, the estimate $a_i$ has the following guarantees:

$$\hat{a}_i \leq a_i + \varepsilon \|a\|_1,$$

where $a$ is the total number of unique items that have been observed so far. Observe that the error bound is always an overestimate thus taking the minimum yields in the best estimate.

![Figure 7: Point query of an item within the Count-Min Sketch. The estimate is given by $\hat{a}_i = 8$.](image)

As shown in Figure 5, the storage space required for one sketch is $O(w \cdot d)$. Throughout this thesis, it is assumed that each index of the CMS is a 32-bit counter thus the memory in bytes, taken per sketch, can be calculated as follows:

$$M = \frac{32 \cdot w \cdot d}{8}$$

For instance, let’s assume that $w = 10$ and $d = 4$. Then, the memory required to store the sketch is: $\frac{32 \cdot 10 \cdot 4}{8} = 160$ bytes.

Based on the definition of a point query, a range query can be calculated as the sum of point queries for the components $a_j, ..., a_{k-1} + a_k$. The error bound for a range query is:

$$\hat{a}[j, k] \leq a[j, k] + n\varepsilon \|a\|_1,$$

where $n$ is the number of point queries. This approach clearly does not result in a good approximate, since the error grows linearly with the number of point queries. Therefore, a less naive approach is to use dyadic ranges. A dyadic range, is an interval with a start index of 1 and where its
length \( l \) is a power-of-two, i.e. \( l = 2^p \). If the desired range is \( 1 \ldots n \), then \( p \) lies within the range \( 0 \ldots \lg(n) \), that is, for each point in \( 1 \ldots n \), a specific point is a member of exactly one dyadic range of length \( 2^0, 2^1, \ldots, 2^\lg(n) \). As it happens, any range within \( 0 \ldots n \), can be divided into at most \( 2\lg(n) \) non-overlapping intervals. The procedure is depicted in Figure 8 and works greedily, starting by finding the largest non-overlapping interval followed by the next largest possible non-overlapping interval.

![Figure 8](image)

Figure 8: When \( n = 16 \), the range \([3-16]\) can be estimated by the dyadic ranges \([9-16], [5-8], [3-4]\). That is, instead of performing 14 point queries, only 3 are required.

Embracing the concept of dyadic ranges, we can store \( \lg(n) \) sketches, meaning that the first sketch has a range of the point query, while the next one combines the first and the second, and so on. The last sketch will have a length of \( 2^{\lg(n)} \) and contain only two intervals. Since the number of queries required is reduced from \( n \) to at most \( 2\lg(n) \), the error bound tightens and instead becomes:

\[
\hat{a}[j, k] \leq a[j, k] + 2\lg(n)\epsilon\|a\|1
\]  

(7)

It follows that the overall storage space required to support range queries is \( O(w \cdot d \cdot \lg(n)) \).
TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics
Henrik Karlsson 2017-05-24

3.7.2 Count-Min Sketch with conservative updates
As mentioned in the previous section, the point query is always an over estimate due to hash collisions. Estan et al. [13] introduces the concept of conservative updates. A conservative update is when a counter is updated only if it is necessary. This is applicable to the CMS since each index is a counter. To update a counter for a specific item, we first calculate the minimum frequency $\hat{a}_i$. Then, an update is carried out:

$$CMS[m, h_m(i)] \leftarrow \max\{CMS[m, h_m(i)], \hat{a}_i + c\} \text{ for } m \in \{1 \ldots d\} \quad (8)$$

This reduces the over counts and thus reduces the overall estimation error.

3.7.3 Time adaptive Count-Min Sketch
The time adaptive CMS is an extension to the previously described CMS which supports temporal queries. Shrivastava et al. [14] explains that when dealing with temporal data, the most recent data tends to be most informative. Therefore, it would be useful to be able to estimate the frequency of an element at any time. A key observation made in [13] is that we can distinguish the same element at different time instances $t_s$, by treating them as different elements within the CMS. Let $t$ be the time interval we are interested in storing. Instead of storing $a$ items as before, we store $a \cdot t$ elements hence the hash functions are:

$$h_1 \ldots h_d: \{1 \ldots n\} \times \{1 \ldots t\} \rightarrow \{1 \ldots w\} \quad (9)$$

This small modification is sufficient to support the point query for any element at any time instance $t_s \in \{1 \ldots t\}$. Additionally, Shrivastava et al. [14] suggests the use of dyadic intervals to support the range query for an item $i$ between time interval $t_{s\text{start}}$ and $t_{s\text{end}}$. The procedure is identical to the one in Figure 8, but with the time instances governing the dyadic intervals. Consequently, storing data over any time interval $t$ requires $\lg(t)$ sketches.

Furthermore, Shrivastava et al. [14] take advantage of the fact that the CMS has better accuracy for high frequent items, also called heavy-hitters. They suggest the use of a function that is multiplied by on update and divided by on querying. The function can be any monotonically increasing function of time. The idea is inspired by Dolby noise reduction, which uses pre-emphasis to boost certain parts of the input signal when recording, whereas de-emphasis is applied to revert the signal into its original
form. This has shown strong theoretical and experimental error guarantees and Shrivastava et al. [13] encourage the use of pre-emphasis and de-emphasis in practice to achieve time adaptivity.

3.8 Clarification

As perhaps obvious, the data structures presented in this chapter are good for one thing: to provide counts for observed events in a data stream over time. It should be remembered that most statistics comes from counts, or more formally, sums. With little effort from the end user, it can be transformed into more “complex” statistics, such as the sample mean or correlations, both of which are based on sums [15]:

\[
\hat{\mu} = \frac{\sum_{j=1}^{n} f_j x_j}{\sum_{j=1}^{n} f_j} \tag{10}
\]

\[
r = \frac{\sum_{j=1}^{n} x_j y_j - n \bar{x} \bar{y}}{\sqrt{\sum_{j=1}^{n} x_j y_j - n \bar{x} \bar{y} - \sum_{j=1}^{n} x_j^2 - n \bar{x}^2}} \tag{11}
\]
4 Web standards, formats and frameworks

4.1 Ajax application
Asynchronous JavaScript and XML (Ajax) [16] is when a web request to a server is issued by the client-side and executed in the web browser. The asynchronous part means that the script that sent the request will continue to execute without waiting for a reply, therefore a page reload is not required. Ajax applications are built on several Ajax requests; the purpose of which is to gather parts of the information for the web page. Instead of requesting the whole page, only the information necessary is requested incrementally, thus, reducing the network load. XML has been the data interchanging standard format, but as the concept of Ajax applications emerged, JSON has become a well-used alternative. Ajax applications have become popular since the technology can make web applications more similar to desktop applications in a responsive manner.

4.2 JSON
JavaScript Object Notation (JSON) [17] is a lightweight textual representation of simple data structures that are interchanged on the web. JSON has support for the primitive types: string, number, Boolean and null, but also for the two structured types: objects and arrays. To represent the JSON structure of a person with the name Henrik who is 23 years old, the notation is as follows:

```
{
    "Person": {
        "name": "Henrik",
        "age": 23
    }
}
```

4.3 REST
Representational State Transfer (REST) [18] is a web architectural style that is commonly used in web services. A web service that adopts this architecture is called a REST API. A REST API consists of multiple inter-
TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics
Henrik Karlsson 2017-05-24

linked resources that are accessible via the use of Uniform Resource Identifiers (URIs). It also inherits the requests, methods, response codes and message headers that are included in the HyperTextTransferProtocol (HTTP). Each HTTP request issued must contain an interaction method in the request message header. The interaction methods and corresponding actions are presented in Table 1.

Table 1: HTTP interaction methods

<table>
<thead>
<tr>
<th>Interaction method</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Retrieve representation of a resource's state</td>
</tr>
<tr>
<td>HEAD</td>
<td>Retrieve the metadata associated with a resource's state</td>
</tr>
<tr>
<td>PUT</td>
<td>Add or update a resource</td>
</tr>
<tr>
<td>POST</td>
<td>Create new resource</td>
</tr>
<tr>
<td>DELETE</td>
<td>Remove resource</td>
</tr>
</tbody>
</table>

For further clarification, let’s extend the JSON object from the previous section, with an identifier field:

```json
{
    "Person": {
        "id": 1234,
        "name": "Henrik",
        "age": 23
    }
}
```

Let’s now assume that the "id" field of a person is what identifies the person as a resource in the URI context. The interaction methods in Table 1 can be combined with the URIs in Table 2 to operate on the person resource.
Table 2: REST resource manipulation

<table>
<thead>
<tr>
<th>Interaction method</th>
<th>URI</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/person/1234</td>
<td>Retrieves the person with id 1234</td>
</tr>
<tr>
<td>HEAD</td>
<td>/person/1234</td>
<td>Retrieves only the metadata</td>
</tr>
<tr>
<td>PUT</td>
<td>/person/1234</td>
<td>Updates the person with id 1234</td>
</tr>
<tr>
<td>POST</td>
<td>/person/1234</td>
<td>Creates a new person with id 12345</td>
</tr>
<tr>
<td>DELETE</td>
<td>/person/1234</td>
<td>Removes the person with id 1234</td>
</tr>
</tbody>
</table>

4.4 Document Object Model

The Document Object Model (DOM) [19] is an application programming interface (API) used to create documents consisting of valid HTML and well-formed XML. A document’s logical structure is defined by the DOM; it also defines how the document is accessed and updated. From a developer perspective, the DOM is used to build documents, navigate their structure and to dynamically add, update or delete content and elements. In other words, everything in a HTML document can be modified using the DOM and it is designed to be used with any programming language. The structure of the DOM can be represented as a tree structure, as in Figure 9, where its corresponding HTML document is as follows.

![Figure 9: DOM tree structure](image)
4.5 JavaScript

JavaScript [20] is an object-oriented scripting language that was released in the 90s. The script part of the name means that a JavaScript runs by an interpreter instead of being read by a compiler and translated to an executable file, as with compiled languages. This makes JavaScript portable since it may run in any interpreter built for the language.

Functions in JavaScript are first-class, that is, the function itself is an object consisting of its own properties and methods. Furthermore, a function may be passed as a parameter into another function or returned from other functions. The language is both dynamically and weakly typed; data types do not have to be defined nor checked at interpreter time. Its syntax and structure is inherited from the C programming language.

4.6 ECMAScript

ECMAScript (ES) [21] is a script language that together with the DOM defines the current JavaScript implementation standard used by the web browser. It is commonly used in web applications to affect how they look and feel for the user. ECMAScript is object-oriented and version ES6 came with several important features such as classes, arrow functions, modules and so on.
4.7 TypeScript

TypeScript is a programming language that is a strict superset of JavaScript. It adds optional types and class-based object-oriented programming [22]. TypeScript compiles to JavaScript, where each expression in TypeScript translates into a JavaScript expression [23]. This makes it possible to mix TypeScript and JavaScript syntax, meaning backwards compatibility with existing JavaScript code is of no concern. Since TypeScript compiles, the benefits of using it includes basic type and syntax checking. It should be pointed out that TypeScript is not type safe. Variables at run time can have a type that does not match the annotation.

The following TypeScript excerpt illustrates a function that outputs a person’s name and age. The last row will not compile to JavaScript, since the arguments when invoking the method does not match the expected types: string and number.

```typescript
interface Person{
    name: string,
    age: number
}

function printPerson(person: Person){
    const person = person.name + " " + person.age + " old";
    console.log(person);
}

printPerson("Henrik", 23);
printPerson(1234); //won’t compile
```

4.8 React

React [24] is a JavaScript library developed by Facebook which can be used to build user-friendly front ends. The idea behind react was that building front ends with JavaScript usually requires that the developer interacts with the DOM. In the interaction with the DOM it is necessary to add, update or delete information or objects dynamically, and, in many
cases it can be cumbersome. Also, one of the biggest problems when developing web applications is keeping the user interface in sync with the business logic and the state of the application. React simplifies this process, by abstracting away the interaction with the DOM, which makes it easier for the developer and has also shown to increase the performance of web applications.

The building block in react is the component, that is, a JavaScript function or class [25]. A component has a number of key characteristics. It implements a render method that will take arbitrary input data and returns what to display. The input data can be stored as an internal state or as a property by the component. If the data is stored as a state and the component's state changes, the render method will be called and the component will be re-rendered.

A simple example of a react component in ECMAScript and TypeScript syntax is shown below:

```javascript
interface MessageInterface{message: string}

class Message extends React.Component<MessageInterface>{

    render(){

        return(

            <div>The current message is: {this.props.message}</div>

        )

    }

}
```

The component will display a div with the message provided to the component, if the argument meets the string requirement.

### 4.9 WebSocket Protocol

Many applications require bidirectional communication between a client and a server. Take for instance stock trading, where a user is interested in several stocks and therefore would like instant notices if any of them changes in value. The traditional way of handling this, is for the user to poll the remote server, where the data is stored, in a frequent manner to detect new updates. Doing so, introduces several problems [25]:
• The server must keep multiple connections open for each client.

• Due to headers in each request, the communication overhead is high.

• Client-side implementation must maintain mappings between incoming and outgoing connections to keep track of replies.

The WebSocket Protocol [25] provides a single TCP connection for bidirectional traffic, that is, the server can send the updates to the client immediately as they are present. In combination with the WebSocket API, which is used on the client side, it can be used for two-way communication between a web application and a remote server. This is made possible by allowing the exchange of messages between the application and server, while keeping the connection open. This technology is suitable in a variety of applications, where real-time updates are needed.

4.10 MongoDB

MongoDB [27] is an open source non-relational database that stores the data in what is called documents. A document follows a JSON-like structure that may vary from simple key value pairs, to objects with nested structures. The representation of each document is called Binary JSON, which is, as the name suggests, a binary representation of JSON. Documents that follow a similar structure are organized into collections, where each collection can be thought of as a table in a relational database, that is, the documents are rows and the fields within the documents are columns. The document model maps directly to the application objects, and the MongoDB driver is supported by over 10 programming languages, which makes it an interesting substitute for other database solutions.

The following code excerpt shows a document for a tweet and how it can be represented within the MongoDB database:

```json
{   _id: ObjectId('1ab23cd4567e'),
    author: 'heka1203',
    content: 'My first tweet!',
    tags: ['#first', '#testing']
}
```
5 Methodology

This chapter describes the idea behind the data streaming algorithms used in this work. It also describes the tools, work process and how the evaluation of the results was carried out.

5.1 Motivation for data streaming algorithms

Goal 1 is to give an overview of existing data sketching techniques. This was done by reviewing the problem of streaming data in previous research, where sketches were found as a commonly used data structure for summarizing streaming data. Because there are several sketching techniques with different functionality and complexity it was necessary to examine concepts inherited by the most basic sketches to the more advanced ones.

Provided with a real-world data set, sales data for a mobile operator in the month of January, the first step was to examine the data set. Each record is called a transaction and contains several fields. Therefore, the data set was cleansed by deciding which fields should be considered for each record. The fields that defines a sale were chosen as:

- date - the date of the transaction
- retailer - the retailer where the transaction was registered
- item - the item sold
- kommando - the type of transaction (identifies the sale category)
Next, the dataset was converted into JSON-format, which was also selected as the interchanging format for the server application. By plotting and visualizing the distribution of unique transactions in the data set, it was concluded that the data set contains very few unique transactions that occurring frequently, but many which occur rarely (note the rough exponential decrease in Figure 10). When a distribution appears as a line on a log-log plot which is the case in in Figure 11, the distribution is usually referred to as a Zipf, a power-law or a Pareto distribution [28]. In fact, most distributions involve some amount of skew [29] hence a solution should take this into consideration.

Due to the skewness of the distribution, it was assumed that the data of interest are in fact the very frequent transactions because remaining transactions are rare by comparison, which motivates the choice of using CMS as a primary building block for the whole application. A skewed distribution has proven to help the CMS in the frequency estimations [10]. This because that it is less likely that the high frequent items will collide. It is
somewhat unlikely that one of the high frequent transactions will collide with another high frequent one. Now, it is even less likely that collision will occur for each row, and, since we use the minimum frequency of the $d$ rows in the sketch, this is not a problem. However, a collision with one of the less frequent ones is more likely, but this won’t have a significant impact on the net frequency. Also, with simple means, the CMS can be extended to provide a list of the top $k$ most frequent items within the stream of data. For instance, consider a user that would like to investigate current trends in the stream of data, the top list could be useful as a hint of which items are of interest to further investigate.

5.2 Development of proof-of-concept application

To verify goal 2 and 3, a proof-of-concept application was developed, where the sketching techniques from the theory were modified and applied. The proof-of-concept application was divided into two parts: a server part and a client part. Different tools and frameworks were used for their development. The following paragraphs describe the methodology applied.

The server application was developed in the Java programming language where the code was written in the open source NetBeans IDE [30]. Preferably, the solution should be able to receive different forms of data, but also respond to requests performed by clients, hence the application required network communication. There were two reasons for choosing Java as the server language. The first is that the language is type safe, which makes the application easier to test and debug, in addition to which it works with the data structures. The second reason is that it was easier to find material and libraries related to network programming in Java in comparison to for example C++.

For the development of the client web application ECMAScript, TypeScript, HTML and CSS were used. Primarily, the JavaScript library React was used, to create all the components in the DOM. Each component contains data produced from requests sent to the server and visualized in the DOM objects. WebSocket communication is used to subscribe and/or unsubscribe to data and to receive real-time updates from the server. CSS was used to style the DOM and to make the web application look and feel more user-friendly. The design principles are based on Material Design, which is a design language for unified user experiences, created by Google [31].
5.3 Test cases

Several test cases were performed to verify goal 5. The data set mentioned in section 4.1, which includes sales data for the month of January, was uploaded to the server application, where several Ajax requests were sent that triggers the actions available on the server. The requests were sent from the client web application where the response was visualized, to ensure that goal 5 was met. The results from the test cases are shown in section 6.3.

5.4 Measurements

Measurements were done to verify goal 6 and 7. First, the query error was measured to evaluate the error within the queries, but also to identify the sketch size required to achieve an acceptable error. Remaining measurements were based on the sketch size found. The query latency and the insertion time were measured to determine how well the proposed solution works in comparison with a traditional Sequential Query Language (SQL) database solution.

5.4.1 Experimental setup

The experimental setup consisted of a single computer where both the server application and the client web application was running locally on a 2015 MacBook Pro hence factors as the network delay are not considered within the measurements of the query latency nor the insertion time. The MacBook Pro has an Intel Core i5 2.7 GHz CPU and 8 GB 1866 MHz RAM.

The necessary information about the sample data set was extracted from the original data set and is presented in Table 3.
Table 3: The dataset

<table>
<thead>
<tr>
<th>Sample size</th>
<th>1,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique transactions</td>
<td>9,500</td>
</tr>
<tr>
<td>Effective unique transactions (including the 31 time instances)</td>
<td>294,500</td>
</tr>
<tr>
<td>Disk space (MB)</td>
<td>80</td>
</tr>
</tbody>
</table>

Varying sample sizes were used for the time measurements, where each sample size was generated by reducing or creating duplicates of the existing records in the original data set. This was done by dividing and/or multiplying the underlying distribution of the data set by a factor. Consequently, the data set followed the same distribution regardless of sample size. All measurements were performed in Java and averaged over 10 runs to reduce the error. The time measurements were performed using the nanotime function which provides nanosecond precision [32]. The SQL database used for comparison was PostgreSQL, which is an open source relational database system [33].

5.4.2 Query error

The error introduced in the queries, was calculated by comparing the estimations with the true value retrieved from the sample data set. Since the error arises from hash collisions, the width of the sketch was varied in the measurements. The depth of each sketch was fixed at $d = 4$, where the expected margin of error was calculated using Equation (3). It follows from the properties of the ceiling function that [34]:

$$\lceil x \rceil = n \iff n - 1 < x \leq n \text{ for } n \in \mathbb{Z}, x \in \mathbb{R}$$  \hspace{1cm} (12)

Using this rule, for $d = 4$, Equation (3) can be rewritten as:

$$4 = \lceil \log \left( \frac{1}{\delta} \right) \rceil \Rightarrow 4 - 1 < \log \left( \frac{1}{\delta} \right) \leq 4 \text{ for } \delta > 0$$  \hspace{1cm} (13)

Solving for the margin of error $\delta$, we get:
TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics
Henrik Karlsson

\[
3 < \frac{\log(\frac{1}{\delta})}{\log(2)} \leq 4 \\
3\log(2) < \log\left(\frac{1}{\delta}\right) \leq 4\log(2) \\
\log(2^3) < \log\left(\frac{1}{\delta}\right) \leq \log(2^4) \\
\log(8) < \log\left(\frac{1}{\delta}\right) \leq \log(16) \\
8 < \frac{1}{\delta} \leq 16 \\
\frac{1}{16} \leq \delta < \frac{1}{8} 
\] (14)

Hence, the margin of error for queries to the sketches was expected to lie within this interval. It was found in section 3.7.1 that the width of a sketch is directly proportional to the memory space taken, but also to the error bound. Since a range query consists of several point queries, with the exception of when the time range is of length 1, which indeed is one point query, the error was only examined for the point query, since it gives an intuition for both. For the 31 time instances of data, \(\log[31] = 5\) sketches were used to support range and point queries. To see how the temporal information affects the error, point queries were issued for each of the 31 days of data. The error estimation can be described as follows. For each day, the top 10 most frequent transactions in the data set were extracted. Then, for each time instance, a point query was issued for all the 10 transactions, where its estimated value was compared with the true value in the sample data set. The Mean Absolute Percentage Error (MAPE) was calculated for each time instance using the equation [13]:

\[
\text{MAPE} = \frac{100}{n} \sum_{j=1}^{n} \frac{|y_j - \hat{y}_j|}{|y_j|},
\] (15)

where \(n\) is the number of observations, \(y_j\) is the true value and \(\hat{y}_j\) is the estimated value. This was done for varying sketch widths \(w = \{512, 1024, 2048, 4096\}\), to see how the error was affected by the memory space used. Based on the results from the point queries, a sketch width of \(w = 4096\) was found to maintain a MAPE of less than 1.1% for the whole-time interval, which was determined acceptable. For a fixed width per sketch \(w = 4096\), the significance of the proposed extensions to the baseline CMS was examined by comparing the MAPE between the baseline CMS, the CMS with conservative updates and the proposed CMS.

For the top-k query, the error estimation is based on the number of \(k\) unique frequent transactions that can be provided. For testing purposes, the measurements were averaged over each time interval within the last 7 days. To make a fair comparison in terms of storage space for the top-
list, its width per sketch was also set to $w = 4096$. To see how the error was affected, measurements were done where the number of top transactions was set to $k = \{8, 10, 12, 14, 16\}$ respectively. A correct estimate is defined as where each of the top transactions is present and in the right order in the result set. The error is therefore how many percent it deviates from the correct estimate. To calculate how much the result deviates from the true value in percent, a value not included in the result set was set to 1, and a value included in the result set, but in the wrong position was set to 0.5. The false positive rate (FP) was then calculated as:

$$FP = \frac{100}{k} \sum_{i}^{1} 1 - \sum_{i}^{0.5},$$

(16)

where $i$ is the number of transactions not in the top-list and $j$ is the number of transactions in the top-list, but in the wrong position.

Finally, for all error measurements, the memory space required was calculated using Equation (5), to provide the reader with a more concrete value.

**5.4.3 Query latency**

For a fair comparison, it should be mentioned that the SQL database can answer many more queries, compared to the proof-of-concept application in this thesis. The focus is, however, low query latency for simple aggregate queries. There are three queries provided by the server application: the point query, the range query and the top query; only these queries were compared to the SQL database. For additional context, the available queries are translated into SQL syntax in Table 4.
TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics
Henrik Karlsson 2017-05-24

Table 4: Queries translated to SQL syntax.

<table>
<thead>
<tr>
<th>Query type</th>
<th>SQL syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point</td>
<td>SELECT COUNT(*) AS c FROM table_name WHERE DATE=(date);</td>
</tr>
<tr>
<td>Range</td>
<td>SELECT COUNT(*) AS c FROM table_name WHERE date BETWEEN date1 AND date2;</td>
</tr>
<tr>
<td>Top-k over recent time windows (last n days)</td>
<td>SELECT COUNT(*) AS c FROM table_name WHERE date &lt;= DATE_SUB(CURRENT_DATE(), INTERVAL, n DAY) ORDER BY c DESC LIMIT k;</td>
</tr>
</tbody>
</table>

Each query was executed at both the server application and the SQL database, where sample sizes \(s = \{2^{17}, 2^{18}, 2^{19}, 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}\}\), were used. The mean query latency for each individual query was calculated and then summed up for the three queries, resulting in an overall mean query latency. A comparison plot was made for each sample size with the query latency in log scale, since the query latency differed a lot in magnitude between the SQL database and the in-memory grid.

5.4.4 Insertion time

The insertion time is the time it takes to insert a new transaction into the server application and/or the SQL database. The time required to insert data into the persistent store was not considered, since all comparisons are between the SQL database and the in-memory grid, although, the MongoDB insertion is an asynchronous operation. Batch insertions were done for different batch sizes \(b = \{2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16}, 2^{17}\}\). For each batch insertion, the number of seconds required to process the whole insertion \(t_{\text{insert}}\) was measured. The mean number of insertions per second was then calculated as:

\[
t_{\text{insert}}/s = \frac{1}{n-m} \sum_{i=m}^{n} \frac{2^i}{t_{\text{insert}}}, \tag{17}
\]

where \(m\) is the first exponent and \(n\) is the last exponent in the sequence of batch sizes.
6 Implementation

This chapter describes how the proof-of-concept application was implemented, starting with an overview of the application flow, followed by a detailed explanation of the implementation concerning the server application and the client application.

6.1 Application flow

Figure 12 depicts the application flow and the system architecture consists of two main components: a client web application and a server. The server application takes input data in JSON-format, processes it and stores it in-memory to answer queries issued from either the client web application or via any other client application that supports HTTP requests. A question that may arise when talking about in-memory storage is: "What if the machine crashes?" Therefore, the server is also connected to a MongoDB database. MongoDB was an obvious choice since the input data to the server is in JSON-format which maps well to the BSON document model in MongoDB. Each insertion is first written to the in-memory grid and then through to the database. This means that the solution has persistent storage and can support recovery from a crash. If a crash occurs, the data will be read from the database and into the in-memory grid, which is built up the same way as before the crash occurred.

Goal 4 of this thesis was to provide a solution that can be widely adopted, in the fields of streaming data. This means that data with different fields is acceptable as input to the server application. Input data is represented as key-value pairs in JSON-format, where each independent data point sent from any source is defined as a tuple. For each unique set of key-value pairs, the server application stores it in a separate in-memory grid. In the following, the in-memory grid will be defined as a cache. The idea of the cache is to store recent data, which can be queried with real-time latency. It is of course not tenable to keep the data in memory forever, however, all data is retained in the persistent store.
Figure 12 shows that data is generated by a collection of devices, which is sent as a stream of tuples to the server application. Each tuple passes through a filter, which decides if it is valid based on the definition of the cache it was sent to, and if it should be further processed and stored or not. Therefore, at the top level of the application, there is a filter. The idea is that only the tuples containing the valid fields for that specific cache are further processed. Since the expected input is supposed to be in JSON-format which consists of multiple key value pairs, the process can be described as follows. The keys of a tuple are extracted as a set, this is also done for the keys field in the cache configuration. Let’s call the set of keys from an input tuple $A$ and the set of acceptable keys $B$. Based on the properties of sets, if $A \subseteq B$ and $B \subseteq A$, then $A = B$, that is, only tuples which fulfill this requirement are further processed. If the data has been processed and stored, it can be served to the clients. The following sections explain each piece in depth.

6.2 REST API
The REST API endpoints define all possible actions accessible via the server application and are therefore explained first, to give the reader more context. All the procedures described in section 6.3 are triggered by a HTTP request to the REST API on the server application. Table 5 describes the possible REST URIs along with their corresponding action.

<table>
<thead>
<tr>
<th>Method</th>
<th>URI</th>
<th>Body format</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>/api/cache</td>
<td>JSON</td>
<td>1</td>
</tr>
<tr>
<td>PUT</td>
<td>/api/cache</td>
<td>JSON</td>
<td>2</td>
</tr>
<tr>
<td>GET</td>
<td>/api/cache/:name</td>
<td>None</td>
<td>3</td>
</tr>
<tr>
<td>DELETE</td>
<td>/api/cache/:name</td>
<td>None</td>
<td>4</td>
</tr>
<tr>
<td>POST</td>
<td>/api/cache/:name</td>
<td>JSON</td>
<td>5</td>
</tr>
<tr>
<td>GET</td>
<td>/api/cache/:name/filter/point/date/:date/tuple/:tuple</td>
<td>None</td>
<td>6</td>
</tr>
</tbody>
</table>
TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics
Henrik Karlsson 2017-05-24

| GET | /api/cache/:name/filter/point/startdate/:startdate/enddate/:enddate/tuple/:tuple | None | 7 |
| GET | /api/cache/:name/filter/range/startdate/:startdate/enddate/:enddate/tuple/:tuple | None | 8 |
| GET | /api/cache/:name/filter/top/days/:days | None | 9 |

The following examples illustrate how each route is used under the assumption that we are working against the same cache.

Action 1 creates a new cache with the provided cache configuration. The body is expected to be in JSON format as follows:

```
{
    "name": "sales",
    "timeToLive": 31,
    "keys": [
        "item",
        "retailer",
        "kommando"
    ],
    "levels": [
        
        "item",
        "item, retailer",
        "retailer"
    ]
}
```
The "name" field is the name that will be referred to when performing insertions or retrievals to/from the cache. The "timeToLive" field determines for how many days the cache will exist in memory before it is expired, in this case 31 days. The "keys" field decides which tuples should be acceptable for each tuple sent to the cache. For instance, a valid input to this cache would be any tuple that contains the keys: item, retailer and kommando. The "levels" field is a list of keys, governing which subset of keys that can be queried for, like the WHERE clause in SQL. Hence, each subset of keys that matches the ones in the levels list, along with their values, are extracted and treated as a unique tuple in the cache. In this case, the cache can be queried per item, per item and retailer, and per retailer.

Action 2 accepts the same input format as in action 1. The difference is that it will modify the fields that are changed in the cache configuration.

Action 3 returns the current representation of the cache configuration.

Action 4 removes the cache from the server application, but also from the persistent store.

Action 5 inserts a new tuple into the cache. For this scenario, an acceptable body would be as follows:

```json
{
    "item": "halebop",
    "retailer": 1234,
    "kommando": "gsm_reg_tjanst"
}
```

For the following URIs, the "tuple" parameter is what identifies the resource and is expected to be in JSON format, such as: {"item": "halebop"}. 

35
TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics
Henrik Karlsson 2017-05-24

Action 6 is a point query and returns a single point estimation for a specific tuple in a specific cache at the given date, where a response may look as follows:

[{
  "tuple": {"item": "halebop"},
  "value": 40,
  "date": "2017-05-12"
}]

Action 7 is multiple point queries and returns every single point estimation for a specific tuple in a specific cache between start date and end date. The response is as in action 6, but the array contains one point estimation for each date in the range.

Action 8 is a range query which returns the sum of point estimations for a specific tuple in a specific cache between start date and end date, where a response may be as follows:

[{
  "tuple": {"item": "halebop"},
  "value": 1000,
  "startdate": "2017-05-12"
  "startdate": "2017-05-30"
}]

]
Action 9 is a top query which returns a list of the top tuples per frequency from the current date and the provided number of days back. The top-list is sorted in descending order, where a response may be as follows:

```
[
    {
        "tuple": {"item": "halebop"},
        "value": 1000
    },
    {
        "tuple": {"item": "spotify"},
        "value": 700
    },
    {
        "tuple": {"item": "telia refill"},
        "value": 500
    }
]
```

6.3 In-memory processing, storage and querying

This chapter describes the proposed implementation, the processing, storing and querying of the received input data in each cache.

6.3.1 Point and range update

For valid tuples, the processing and storage can begin. In Chapter 3 the concept of sketches, hash functions and dyadic intervals were introduced, all of which are required components for the storing of the tuples. The main component that is used is the Count-Min Sketch. There are some assumptions made to simplify the process of explaining:
Each tuple is assumed to be stored at a daily resolution. In practice, it is a timestamp of arrival (but the concept is the same for any integer value), the time always starts at the first day of 2017, i.e. 2017/01/01.

- Tuples are kept for 31 days.
- Each query reflects a frequency estimation of a tuple based on key-value pairs that are a subset of all its key-value pairs.

The following notations are used throughout the chapter:

- \( T \): A tuple containing several key-value pairs (keys are denoted with subscripts).
- \( ts \): A timestamp at daily resolution for when the tuple was observed.
- \( t \): Time interval.
- \( h_m(x) \): A universal hash function within the range 1 \( \ldots \) \( d \)
- \( CMS_n \): The \( n \)th Count-Min Sketch within the range 1\( \ldots \)\( \lg(t) \)

As observed in section 3.7 the idea is to keep \( \lg(t) \) sketches, to be able to answer range queries with minimal error bounds. Hence, the number of sketches are \( \lceil \lg(31) \rceil = 5 \) sketches.

Figure 13 depicts the initial state of the sketches. Given that the time range is 1 \( \ldots \) 31 days, it can be noted that sketch 1 stores point estimations at a resolution per day, sketch 2 stores point estimations for every two days and so on.

![Figure 13: Initial state of the 5 sketches.](image)

When a new tuple \( T \) arrives in the stream, it is timestamped. This means that \( ts \) will fall within the range 1 \( \ldots \)\( \lg(t) \). The bin interval to which \( ts \) belongs, of sketch \( n \), is defined as:
\[\text{tupleSearch} \rightarrow \text{A scalable framework based on sketches to process and store streaming temporal data for real time analytics}\]

Henrik Karlsson 2017-05-24

\[bin^ts_n = \left\lfloor \frac{ts}{2^{n-1}} \right\rfloor \text{ for } 0 < n \leq \lfloor \log(t) \rfloor (18)\]

For instance, let’s assume that \( T \) has arrived at 2017-01-16, that is: \( ts = 16 \). The calculations, to which bin interval \( ts \) belongs in each sketch, are as follows:

\[bin^1_{16} = \left\lfloor \frac{16}{2^0} \right\rfloor = 16 \]
\[bin^2_{16} = \left\lfloor \frac{16}{2^1} \right\rfloor = 8 \]
\[bin^5_{16} = \left\lfloor \frac{16}{2^5} \right\rfloor = 1 \quad (19)\]

Figure 14 illustrates the scenario stated above.

![Figure 14: Finding the bin intervals for \( ts = 16 \) within each sketch.](image)

The value of the bin interval will act as a tuple index in each sketch. Important to note, is that the wider the sketch is, the more timestamps will fall into a specific bin interval. Let’s suppose three tuples are observed. The first at 2017-01-01, the second at 2017-01-04 and the third at 2017-01-16. Hence, we have: \( ts = 1, ts = 4, ts = 16 \) accordingly. As an example, the calculations for finding the bin intervals of sketch 5, are as follows:

\[bin^1_5 = \left\lfloor \frac{1}{2^4} \right\rfloor = 1 \]
\[bin^4_5 = \left\lfloor \frac{4}{2^4} \right\rfloor = 1 \]
\[bin^5_5 = \left\lfloor \frac{16}{2^4} \right\rfloor = 1 \quad (20)\]

It can be noted that all timestamps within 1...16, fall within the first bin interval of sketch 5. This means that all of the above tuples are inserted into the same index in sketch 5.
An update of a newly arrived tuple, can be described algorithmically as in Figure 15. It can be noted that there are four procedures which are indirectly dependent of each other. When a tuple arrives in the stream, the range update method is invoked, which, for each of the $\log(t)$ sketches, invokes the point update method. The point update method finds the correct bin interval, as in Equation (18). Then, a point query is issued which returns the current minimum frequency $\hat{a}_T$ divided by a monotonic function of time. The suggested implementation uses both conservative updates and a modified version of de-emphasis. It can be noted, based on the error bound of the CMS, that the error increases when the number of unique tuples are increased in the distribution, i.e. with time. Also, the smaller the width of a sketch, the higher the error. Consequently, the monotonic function $f$ is exponential, where the base $b$ is dependent of the sketch width and the exponent of the bin interval such as:

$$f = b^{bin} \text{ for } b = 1 + \frac{1}{w}$$

The tuple index for each row is calculated by hashing the tuple string and shifting the hash the number of bits of the bin interval, hence the bin interval distinguishes the different dyadic ranges. Finally, each corresponding tuple index is conservatively updated.
TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics

Henrik Karlsson

2017-05-24

Algorithm 3 Find the bin interval at time $ts$ of the $n$th sketch

Require:
- $ts$: timestamp
- $n$: sketch sequence index
1: procedure FINDBIN($ts, n$)
2: return $\text{cell}(ts/2^{n-1})$
3: end procedure

Algorithm 4 Increment the frequency for tuple $T$ of the $n$th sketch.

Require:
- $T$: tuple in JSON-format
- $ts$: timestamp
- $n$: sketch sequence index
- CMS: a sequence of $w \times d$ registers
1: procedure POINTUPDATE($T, ts, n$)
2: $\text{bin} \leftarrow \text{FINDBIN}(ts, n)$
3: $\hat{a}_T \leftarrow \text{POINTQUERY}(T, \text{bin})$
4: for $m = 1 : d$ do
5:   $CMS[n][\hat{b}_m(T) \ll \text{bin}] \leftarrow \max(CMS[n][\hat{b}_m(T) \ll \text{bin}], \hat{a}_T + 1)$
6: end for
7: end procedure

Algorithm 5 Increment the frequency for tuple $T$ of each sketch within the dyadic interval.

Require:
- $T$: tuple in JSON-format
- $ts$: timestamp
1: procedure RANGEUPDATE($T, ts$)
2: for $n = 1 : \text{cell}(\log_2(t))$ do
3:   POINTUPDATE($T, ts, n$)
4: end for
5: end procedure

Algorithm 6 Estimate the frequency for tuple $T$ of the $n$th sketch.

Require:
- $T$: tuple in JSON-format
- $ts$: timestamp
- $n$: sketch sequence index
- CMS: a sequence of $w \times d$ registers
- $f()$: a monotonically increasing function of time
1: procedure POINTQUERY($T, ts, n$)
2: $\text{bin} \leftarrow \text{FINDBIN}(ts, n)$
3: $\hat{a}_T \leftarrow \min_{m \in [1, d]}(CMS[n][\hat{b}_m(T) \ll \text{bin}])$
4: return $\frac{\hat{a}_T}{\text{bin}}$
5: end procedure

Figure 15: Update and query procedure for a tuple at time $ts$.

To provide multiple levels of granularity, the procedure is not only done for all the key-value pairs within $T$, but also for the subset of key-value pairs that are specified in the "levels" field of the cache configuration. As a result, subsets of key-value pairs can be queried for, since the provided combinations of key-value pairs are stored.
6.3.2 Point and range query

Unlike the point update, the point query is a frequency estimation of a specific tuple \( T \) with timestamp \( t_s \). Its procedure is very similar to the point update, but there are two discrepancies. First, instead of incrementing the value at the calculated indices in the sketch, we look at their current values. The estimate is given by the minimum value \( \hat{a}_R \), of the values at the corresponding indices in each row. The procedure is also depicted in Figure 15, observe that if \( n = 1 \), it will return the frequency estimation at that exact timestamp. In fact, to provide the point estimation for a specific timestamp \( t_s \), we must have \( n = 1 \). Other values for \( n \) are only useful for the range query.

The range query is more complex. Here the concept of dyadic intervals is used; see section 3.7 for a reminder. A range query sums the frequency estimations between \( t_{s\text{start}} \) and \( t_{s\text{end}} \) in at most \( 2\lg(t) \) point queries. First, we find the dyadic intervals in the range, which is a procedure identical to the procedure in Figure 8, but with the timestamps governing the dyadic intervals. Let’s assume that tuple \( T \) has arrived multiple times for timestamps \( t_s \) in \{1,2,3,4,5,6,7,8\} and has been inserted into the sketches as previously described. To estimate the range sum of frequencies for \( T \) between \( t_{s\text{start}} = 1 \) and \( t_{s\text{end}} = 8 \), we start by finding the dyadic intervals as demonstrated in Figure 16.

![Figure 16](image)

**Figure 16:** When \( t = 31 \), the time range between \( t_{s\text{start}} = 1 \) and \( t_{s\text{end}} = 8 \) is covered by the dyadic ranges [1-2], [3-4], [5-8].

The binary logarithm of the length for each dyadic interval corresponds to the sketch associated with it, hence for every dyadic interval we can perform a point query. Furthermore, \( t_s \) is set to any value in its corresponding dyadic interval, since the bin interval is always calculated afterwards when the point query is carried out, that is, \( t_s \) will fall into the same bin interval for that whole dyadic interval. Finally, each of the individual point queries are summed up, see Figure 17. In this example, we would perform three point queries for: \( n = 1, t_s = 1; n = 1, t_s = 3; n = 2, t_s = 5 \) accordingly.
6.3.3 Top-k update

The idea of keeping top k frequency estimations over recent time windows can give an indication of recent trends. There is one top-list provided in the current implementation, consisting of the top-level tuples, i.e. the set of key-value pairs that matches the attributes field in the cache configuration, but more levels can easily be added. To keep the top-list updated, three components are used. The first is the Count-Min Sketch (with the extensions described in the previous section), the second is a fixed size heap and the third is a fixed size FIFO queue. In the case of multiple top-lists, the heap and the queue would be distinct for each top-list. In the update procedure which will be described below, the following notations are used:

\( w \)  
A time window for the number of olden days

\( k \)  
The number of top tuples to keep track of

\( H_w \)  
The heap holding top tuples for time window \( w \)

\( CMS_w \)  
The Count-Min Sketch for time window \( w \)

We would like to query top tuples over recent time windows, that is, \( w \) is equal to the recent days that we are interested in. To do so, for every sub time window within the range \( 1 \ldots w \), a corresponding heap and Count-Min Sketch is kept. Thus, the sequence of heaps are: \( H_1, \ldots, H_{w-1}, H_w \) and the sequence of sketches are: \( CMS_1, CMS_{w-1}, CMS_w \). For instance, \( H_1 \) will contain top tuples over the last day, whereas \( H_w \) will contain top tuples over the last \( w \) days. Since \( k \) determines the number of elements that each top-list contains, it is also the size of the associated heaps. Similarly, the length of the FIFO queue is set to \( w \), hence the queue works in a circular manner, where old information is removed first.
For simplicity, assume that we want to store the top three tuples for the last three days, i.e. \( k = 3, w = 3 \), and that the first tuple was observed at 2017-01-01. Also, assume that a stream of tuples has been received for four days, where the corresponding daily frequencies are summarized in Table 6, along with pre-calculated values of what each point query would return from the three sketches.
Table 6: Daily frequencies for tuples A, B, C, D between 2017/01/01 and 2017/01/04

<table>
<thead>
<tr>
<th>Date</th>
<th>Tuple</th>
<th>Frequency</th>
<th>CMS1</th>
<th>CMS2</th>
<th>CMS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017/01/01</td>
<td>A</td>
<td>40</td>
<td>40</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2017/01/01</td>
<td>B</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2017/01/01</td>
<td>C</td>
<td>51</td>
<td>51</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2017/01/01</td>
<td>D</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2017/01/02</td>
<td>A</td>
<td>20</td>
<td>20</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>2017/01/02</td>
<td>B</td>
<td>40</td>
<td>40</td>
<td>90</td>
<td>0</td>
</tr>
<tr>
<td>2017/01/02</td>
<td>C</td>
<td>30</td>
<td>30</td>
<td>81</td>
<td>0</td>
</tr>
<tr>
<td>2017/01/02</td>
<td>D</td>
<td>46</td>
<td>46</td>
<td>59</td>
<td>0</td>
</tr>
<tr>
<td>2017/01/03</td>
<td>A</td>
<td>40</td>
<td>40</td>
<td>60</td>
<td>100</td>
</tr>
<tr>
<td>2017/01/03</td>
<td>B</td>
<td>22</td>
<td>22</td>
<td>62</td>
<td>112</td>
</tr>
<tr>
<td>2017/01/03</td>
<td>C</td>
<td>19</td>
<td>19</td>
<td>49</td>
<td>100</td>
</tr>
<tr>
<td>2017/01/03</td>
<td>D</td>
<td>21</td>
<td>21</td>
<td>67</td>
<td>80</td>
</tr>
<tr>
<td>2017/01/04</td>
<td>A</td>
<td>15</td>
<td>15</td>
<td>55</td>
<td>75</td>
</tr>
<tr>
<td>2017/01/04</td>
<td>B</td>
<td>25</td>
<td>25</td>
<td>47</td>
<td>87</td>
</tr>
<tr>
<td>2017/01/04</td>
<td>C</td>
<td>25</td>
<td>25</td>
<td>44</td>
<td>74</td>
</tr>
<tr>
<td>2017/01/04</td>
<td>D</td>
<td>15</td>
<td>15</td>
<td>36</td>
<td>82</td>
</tr>
</tbody>
</table>

The timestamp calculations are different in nature than previously described queries. First, the time difference between the previously received tuple and the newly received tuple is calculated as:

\[
    ts_{diff} = ts_{new} - ts_{prev}
\]

(22)

Then, a timestamp residual is calculated as:

\[
    ts_{res} = ts_{diff} \mod w
\]

(23)

There is a certain case that requires extra care. If the calculated timestamp residual is greater than the previous one, a new heap with
$w = 1$ is created and enqueued as in Figure 18. In both cases a new time window has passed by (in the example one day). If the queue is already full, $H_w$ is dequeued as in Figure 21.

The timestamp and the timestamp residual is kept in the top-list for comparison with coming tuples. Moreover, for each of the $w$ heaps and $w$ sketches, a point update is executed, followed by peeking at its current frequency estimation, i.e. performing a point query for that specific tuple. Then, the tuple is inserted into the heap if its frequency estimation is greater than any of the tuples currently in the heap or if the heap is not full. When the heap is full, it replaces the tuple with the one with the lowest frequency estimation. Figure 18-21 depicts the procedure, based on the scenario in Table 6.

![Figure 18: The FIFO queue at the end of 2017/01/01.](image1.png)

![Figure 19: The FIFO queue at the end of 2017/01/02.](image2.png)
Figure 20: The FIFO queue at the end of 2017/01/03.

Figure 21: The FIFO queue at the end of 2017/01/04. The queue is full, therefore the heap at the front is dequeued at the beginning of 2017/01/04.

6.3.4 Top-k query

To query the top-list for the top \( k \) tuples \( d \) days back, a copy of the queue has to be created, and dequeued \( w - d \) times, as the front of the queue will now be the heap containing the top tuples for \( d \) days back, as Figure 22 illustrates.
Figure 22: Querying the top-list for the top tuples of the last day.

In combination with the point query, we can provide a top-list that contains the top $k$ tuples with their corresponding frequency estimations.

6.3.5 Subscribe service

For each tuple that has been processed and stored in a cache, a client can query it via the REST API as described in the previous sections. However, it could be useful for the client to receive real-time updates when a specific tuple is updated. Hence, there is a subscribe service, accessible via the route: "/live". It is used to subscribe and/or unsubscribe for one or more tuples. The process can be described as follows: the client connects to the WebSocket route and sends a message with the following format:

```json
{
    "action": "",
    "tuples": []
}
```

The "action" field can be either "SUBSCRIBE" or "UNSUBSCRIBE". The "tuples" field can be a sequence of tuples. When there is a new subscription, the server will store that particular client and observe that specific tuple for new updates. In the current implementation, the server application will wait for a tick period of 5 seconds before notifying the client. The response message will include the point estimation for that specific tick period. Also, it will include the ISO date for the start of that tick period.
As a result, it can be used for real-time plotting. As an example, the response may be as follows.

```json
{
    "tuple": {
        "item": "halebop",
        "retailer": 1234,
        "value": 120,
        "date": "2017-01-06T16:44:10.470Z"
    }
}
```

### 6.4 Client application

The client application has two actions: to query the REST API in a user-friendly manner and to visualize the response. The creation of the client application was done to demonstrate the capabilities of the server application from a graphical perspective hence its simplicity. The application is built up by two routes. The first route is the start page, which consists of one component, an input field which uses an existing cache name (found in-memory on the server application) as input. Therefore, if the cache exists, the user will be redirected to the "dashboard" route, where any further action corresponds to that specific cache. Figure 23 depicts the cache search field.

![A dashboard for visualizing data.](image)

**Figure 23: Cache search field**

The "dashboard" route consists of three components: a menu component, a data finder component and a plot grid component.
The menu component basically has one action: to hide or show the data finder component. It is placed at the bottom right of the page and has, due to its simplicity, been left out from the illustrations.

Figure 24 depicts the data finder component. It consists of three sub components. To the left there is a form that is used to communicate with the REST API on the server application. The fields in the form are the keys of the tuples that the cache accepts as input. The input to these fields identifies which tuple we are querying statistics for. At the top, there is a drop-down menu with three actions: point query, range query and subscribe. The point query returns point estimates for each date within the specified date range. The range query returns a sum for the whole date range specified. If the subscribe action is selected, the client application will inform the server that the subscriber is interested in a specific tuple and wants immediate updates when it changes. This means that updates are immediately reflected without reloading the page because of the subscribe service. At the bottom left there is an add button, which sends the request to the server. To the right in the data finder component there is a list that reflects the response data from the server application. In other words, data is queried for or removed via this component.

![Figure 24: Data finder](image)

Each action in the data finder component is immediately reflected in the plot area component. For example, data was queried for the most frequent items in the month of January. In Figure 25, at the top, the results from the point query are displayed. Point queries are presented as scatters plots with a line connecting each point. The line plot is interactive; it can be zoomed in and/or out and data can be hidden. At the bottom left, range queries are represented as bar plots. At the bottom right the top-list that
reflected the top-k query is found. Since the top-list can indicate hints on interesting trends, it is always visible. By hovering over each colored arch in the plot the corresponding tuple is shown. The top-list can be interactive; dragging the "range field" to the right, will immediately reflect the changes for the previous day.

**Figure 25: Plot area**

The subscribe plot is difficult to visualize as a figure since it changes with time, but Figure 26 could give an intuition. In Figure 26 the estimations for three tuples have been received over a period of time. What the figure cannot describe is that the points on the y axis are "flowing" to the left as new points are added from the right, i.e. with time.
TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics
Henrik Karlsson 2017-05-24

Figure 26: Subscribe plot
7 Results

7.1 Query error

Figure 27 depicts the MAPE of point queries issued for each time instance within the 31-day time interval, and for various sketch widths. The global trend for the error is an increase when the time queried for is increased. Furthermore, the error decreases as the width per sketch increases. For instance, a width per sketch of 512 has a MAPE of less than 25% for the whole-time interval, whereas a width per sketch of 4096 has a MAPE of less than 1.1%. In Table 7, the width and number of sketches are translated into the required space to support the point query and the range query. It can be noted that $n = 5$, since we require $\lceil \lg(31) \rceil = 5$ sketches, to support the range query for all 31 time instances.

![Graph of MAPE over time for different sketch widths](image)

*Figure 27: Point query error estimation for all time instances within the 31 days.*
Table 7: Memory space required to support the point query and the range query.

<table>
<thead>
<tr>
<th>w</th>
<th>M</th>
<th>M · n</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>(\frac{32 \cdot 512 \cdot 4}{8}) = 8.192 KB</td>
<td>8.192 (\cdot \lceil \log(31) \rceil) = 40.96 KB</td>
</tr>
<tr>
<td>1024</td>
<td>(\frac{32 \cdot 1024 \cdot 4}{8}) = 16.384 KB</td>
<td>16.384 (\cdot \lceil \log(31) \rceil) = 81.92 KB</td>
</tr>
<tr>
<td>2048</td>
<td>(\frac{32 \cdot 2048 \cdot 4}{8}) = 32.768 KB</td>
<td>32.768 (\cdot \lceil \log(31) \rceil) = 163.84 KB</td>
</tr>
<tr>
<td>4096</td>
<td>(\frac{32 \cdot 4096 \cdot 4}{8}) = 65.536 KB</td>
<td>65.539 (\cdot \lceil \log(31) \rceil) = 327.68 KB</td>
</tr>
</tbody>
</table>

Figure 28 depicts a point query error comparison between the baseline CMS, the CMS with conservative updates and the proposed CMS. It can be noted that the CMS with conservative updates and the proposed CMS is more accurate than baseline CMS. Moreover, the proposed CMS performs better than both the others for more recent time instances.
As described in section 5.4.2, the top-list has a fixed $w = 4096$ per sketch, which is based on a MAPE of less than 1.1%, based on the previous measurements. In addition, the measurements were performed for the last 7 days hence $n = 7$. In Figure 29 the top query error estimation is presented. Each bar corresponds to the percentage of correct estimations that were made for that specific $k$. The results show that the error increases as the number of top frequent transactions captured is increased. It can be noted that for $k > 8$ an error is introduced. In Table 8 the memory required to support the top-$k$ query is presented. The size of the heaps and the FIFO queue are neglected since their memory space is constant.
TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics
Henrik Karlsson 2017-05-24

Figure 29: Top-k query error estimation averaged over all time intervals within the last 7 days.

Table 8: Memory space required to support the top-k query.

<table>
<thead>
<tr>
<th>( w )</th>
<th>( M )</th>
<th>( M \cdot n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>( 32 \cdot 4096 \cdot 4 )( \frac{4}{8} )</td>
<td>( 65.536 \cdot 7 = 458.752 \text{ KB} )</td>
</tr>
</tbody>
</table>

7.2 Query latency

Figure 30, depicts the results from the measurement of the query latency for the top query, the range query and the point query and a comparison to the query latency of their queries in the SQL database. On the x-axis, the sample size has power-of-two step size. On the y-axis, the query latency is measured in microseconds. The query latency is the mean query response time average over the three separate queries previously mentioned. It is important to notice that the query latency on the x-axis is in log scale.

It is likely that, when querying the cache, the query latency is constant (about 150 \( \mu s \)), whereas the SQL query latency grows linearly with the increased sample size. At its most extreme, when the sample size is \( 2^{20} \), the mean SQL query latency is roughly 16 s.
Figure 30: Mean query latency for SQL vs CACHE for various sample sizes.

7.3 Insertion time

Figure 31 depicts the results of the measurements of the throughput of batch insertions with various batch sizes of \(2^{10-17}\). The y-axis shows the insertion time in seconds, for a specific batch insertion, whereas the x-axis shows the corresponding batch size. It can be noted that for both data sources, the insertion time grows linearly (the x scale is logarithmic but not the y scale).
Figure 31: Insertion times for SQL vs CACHE for various batch sizes.

From the measurements and by using Equation (17), the cache has a mean number of insertions per second of:

\[
\frac{1}{17-10} \left( 2^{10} \cdot 0.062 + 2^{11} \cdot 0.098 + 2^{12} \cdot 0.198 + 2^{13} \cdot 0.42 + 2^{14} \cdot 0.80 + 2^{15} \cdot 1.56 + 2^{16} \cdot 3.10 + 2^{17} \cdot 5.88 \right) \approx 23218 \text{ tuples/s} 
\]

The SQL database has a mean number of insertions per second of:

\[
\frac{1}{17-10} \left( 2^{10} \cdot 0.75 + 2^{11} \cdot 0.93 + 2^{12} \cdot 1.61 + 2^{13} \cdot 2.48 + 2^{14} \cdot 4.58 + 2^{15} \cdot 9.05 + 2^{16} \cdot 17.90 + 2^{17} \cdot 36.20 \right) \approx 3414 \text{ tuples/s} 
\]
8 Conclusion

In this thesis, a framework, to process and store a possibly infinite stream of temporal data with varying fields is proposed, such that statistical information can be retrieved in real time. The measurements show that we can expect a constant query latency of about 150 μs, independently of the volume of data. Also, the insertion time is about $\frac{23218}{3414} \approx 6.8$ times faster than a traditional database system. As with everything, this comes at a cost, which in this case is a query error vs. memory space tradeoff. The error is dependent on how much memory we are willing to sacrifice along with the number of unique tuples in the distribution of data. For this particular data set (of size 80 MB), it has been shown that we can support the point query and the top query for the most frequent tuples, with a less than 1.1% error by using $327.68 + 458.752 = 786.432$ KB of memory. The range query error could be expected to grow by at most $2\lg(t)$, but since the temporal data is binary aggregated, it means that the greater the range queried for the higher the time resolution of the involved dyadic ranges. This means that we can expect fewer hash collisions since less space is used, thus the overall error range query error will always be smaller than for the point query. The fast insertion time along with the low query latencies, makes the solution scalable and well-suited for real-time analytics. The fluctuations in MAPE for the point query error, is expected due to the choice of $d = 4$, but also because hash collisions will occur at random and the peaks in MAPE arises when a hash collision has a high impact on the frequency. The measurements for the top-k query error implies that the overall error only holds for the most frequent tuples, which is expected due to the low number of unique high frequent transactions in the data set, and gets worse for the less frequent ones. This is acceptable, since the solution has its focus on trendy data. To sum up, the goals stated in section 1.4 were achieved.

There are two improvements that could be carried out as further work. The first is that different fields of application may require data of different time resolution. As an example, sensor data may have to be stored at second resolution, whereas sales data may have to be stored at a daily resolution. In the current implementation, only daily resolution is provided. This could be a parameter provided in the cache configuration. Since an arrival timestamp of each tuple is taken, it could easily be converted into the wanted time resolution.
The second is that at least one more data set should be tested, so that an expression for the width per sketch (that is based on the number of unique items in the expected distribution) could be derived and become a parameter in the cache configuration. This is, at the same time, a drawback of the proposed framework, since the client must have some intuition about the expected distribution beforehand.

From an ethical point of view, the data is stored as hashes within the caches, which means that without knowing what to search for, nothing can be found and therefore the solution maintains some extent of privacy. Nevertheless, the solution is good for real-time analytics, which can save the time required to perform the traditional batch processing. In most cases, time is equal to money hence a user of this application can benefit from this solution from a social perspective.
References

[1] Amazon Web Services, "What is streaming data? ",
https://aws.amazon.com/streaming-data/

[2] Yang, Qiang, and Xindong Wu. "10 CHALLENGING PROBLEMS IN
DATA MINING RESEARCH", International Journal of Information Technol-


Internet is Changing Everything (April 2011).", White Paper by Cisco Inter-
net Business Solutions Group (IBSG), 2012.


[8] Shahrivari, Saeed. "Beyond batch processing: towards real-time and

[9] Ellis, Byron. Real-time analytics: Techniques to analyze and visualize

processing.", Synposes for Approximate Query Processing: Samples, Histо-

pp. 485-509.
TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics
Henrik Karlsson 2017-05-24


TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics
Henrik Karlsson 2017-05-24

[22] TypeScript, "Documentation",
https://www.typescriptlang.org/docs/tutorial.html


https://www.typescriptlang.org/docs/tutorial.html

[26] IETF, "The websocket protocol.",

https://www.mongodb.com/mongodb-architecture
Retrieved 2017-05-02.

[28] HP Labs, "Zipf, power-laws, and pareto-a ranking tutorial.",
http://www.hpl.hp.com/research/idl/papers/ranking/ranking.html,
Retrieved 2017-05-03.


https://netbeans.org/
Retrieved 2017-05-03.

http://materializecss.com/about.html
Retrieved 2017-05-03.
TupleSearch – A scalable framework based on sketches to process and store streaming temporal data for real time analytics
Henrik Karlsson 2017-05-24

[32] Oracle, "nanoTime",
https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#nano-Time()
Retrieved 2017-05-03.

[33] PostgreSQL, "About",
https://www.postgresql.org/about/
Retrieved 2017-05-05.

[34] Knuth, Donald Erwin, Ronald L. Graham, and Oren Patashnik. Concrete mathematics. 2e. Adison Wesley, 1989.