Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Sandra Marie Skarshaug

Ranking Streaming Data With Continuous Queries

Masteroppgave i Datateknologi
Veileder: Heri Ramampiaro
Juni 2019

Sandra Marie Skarshaug

Ranking Streaming Data With Continuous Queries

Masteroppgave i Datateknologi
Veileder: Heri Ramampiaro
Juni 2019

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Abstract

An increasing amount of data is being generated as part of the digitialization of our society. In 2013, research found that 90% of the data in the world was generated during the past two years. Social media platforms have become a part of people's daily life, and the usage of these platforms can generate large data streams, such as the stream of messages from microblogging platform Twitter, where millions of "tweets" are posted daily. This data can be analyzed in real-time to gain insight into many subjects, for instance natural disasters or other crises as they happen, but it would be overwhelming for a user with a specific information need to cherry-pick the most relevant posts from such an immense volume of data. This has led to a need for automatic, real-time systems for handling such tasks. Real-time analysis of streaming data is not a new idea, but many of the previous approaches have required the use of several independent systems which are "glued" together.

Bearing in mind the above challenges, this project investigates how to implement ranking of items in a data stream generated by a social media platform using an existing, unified big data management system. The goal is to be able to continuously identify and retrieve the most relevant items by ranking based on a user's information need at every time instant, and thus address the information overload effect users can be subject to when using the web.

In the proposed system, the first step is to filter the data stream by the means of a continuous user-defined query to avoid processing data not found relevant. Next, an online clustering algorithm is applied to the remaining tweets to further reduce the search space of relevant items. Then, a scoring function calculates the relevance score for each cluster with respect to the user query, and these are ranked to find the top-k most relevant ones. Finally, only tweets in the highest ranked cluster are retrieved and persisted, and the storage is updated as the most relevant items change as time passes by. A real-time experiment show that filtering and ranking is applied to the data stream, and that the system updates the retrieved result based on the current ranking with low costs. This study show that streaming data can be handled natively within AsterixDB, yielding no need for combining several systems for that purpose.

Sammendrag

En økende mengde data genereres som en del av digitaliseringen av samfunnet vårt. Forskning fra 2013 viser at 90% av all dataen generert i verden frem til det tidspunktet, ble generert i løpet av de to foregående årene. Sosiale medier har blitt en del av hverdagen til folk, og måten mennesker bruke disse sosiale mediene kan generere store datastrømmer, slik som for eksempel strømmen av meldinger fra mikrobloggen Twitter, som genererer millioner av "tweets" daglig. Denne dataen kan analyseres i sanntid for få innblikk i mange temaer, for eksempel hvilke naturkatastrofer eller andre kriser som rammer verden i et gitt øyeblikk. Men, det er ikke mulig for en bruker med et spesifikt informasjonsbehov å navigere den store mengden av data for å finne akkurat den dataen som er mest relevant for henne. Dette har skapt et økende behov for automatiske sanntidssystemer for å håndtere slike problemstillinger. Sanntidsanalyse av denne typen strømdata er ikke en ny idé, men mange av de tidligere tilnærmingene til løsninger har vært avhengige av å "lime" sammen flere uavhengige systemer.

I lys av de overnevnte utfordringene utforskerer denne oppgaven hvordan elementer i en datastrøm generert av et sosial medium kan rangeres ved å benytte eksisterende systemer som håndterer Big Data. Målet med å utføre rangering er å til enhver tid kunne identifisere og hente ut den mest relevante informasjonen fra datastrømmen for et gitt informasjonsbehov. Dette adresserer problemet med informasjonsoverflod som brukere kan oppleve på nett.

Det første steget i det foreslåttet systemet er å filtrere datastrømmen basert på en stående brukerspørring, og dermed redusere mengden data som må prosesseres. Videre blir en grupperingsalgoritme tatt i bruk på de resterende elementene i datastrømmen for å redusere antallet enheter som må rangeres. Deretter blir relevansen mellom grupperingene av Twitter-meldinger og brukerspørringen kalkulert, og det blir produsert en liste over de k mest relevante grupperingene. Til slutt vil bare tweets som er lagret i den høyest rangerte grupperingen bli persistent lagret, og lagringsmediumet blir oppdatert kun når det er endringer i rangeringen av grupperinger. Et sanntidseksperiment viste at filtrering og rangering blir påført med hell på datastrømmen, og at systemet oppdaterer resultateat basert på den nåværende rangeringen med lav kostnad. Denne oppgaven viser at strømdata kan håndteres internt i AsterixDB, og fjerner behovet for flere systemer til å løse et slikt problem.

Preface

This thesis is submitted to the Norwegian University of Scicence and Technology (NTNU), as the final requirement for a degree in Master of Science. The work was carried out at the Department of Computer Science during the spring of 2019.

During the specialization project in the course TDT4501 at NTNU in the autumn semester of 2018 – entitled *Filtering and Clustering of Tweets In AsterixDB* – I and my partner, Rosita Høybakken, created a solution for real-time clustering and filtering of tweets. As much of the background theory and the related work on streaming data and Big Data are still relevant for the chosen method in this study, these parts will be included in this master thesis.

Trondheim, June 13, 2019

Sandra Skarshaug

Acknowledgements

The idea of this project was raised by Heri Ramampiaro at the Department of Computer Science. During the last year he has been my supervisor, and I would like to express my sincere gratitude for his guidance and feedback. By letting me work independently, but still helping me back on the right track whenever I got lost. The developers at AsterixDB must also be thanked – especially Xikui Wang, PhD candidate at University of California, Irvine – for the support given me on questions related to AsterixDB. He went beyond all my expectations to ensure that no questions related to AsterixDB was left unanswered. I would also like to thank my boyfriend for the emotional support and feedback on this report, and my sister likewise.

Table of Contents

List of Tables

List of Figures

Acronyms

ADM Asterix Data Model. 18, 19

API Application Programming Interface. 18, 21

BAD Big Active Data. 25

BDMS Big Data Management System. 18

CC Cluster Controller. 18, 19

DAG Directed Acyclic Graph. 19

DBMS Database management system. 10

DSMS Data Stream Management System. 10

GloVe Global Vectors for Word Representation. 12

HTTP HyperText Transfer Protocol. 18

JSON JavaScript Object Notation. 18

NC Node Controller. 19

TFIDF Term Frequency-Inverse Document Frequency. 11

UDF User defined function. 19, 48

Chapter 1

Introduction

This introductory chapter will present the necessary background information for understanding the problem at hand. Thereafter, the research questions and objectives will be presented, followed by a section describing the research strategy chosen for this project. The scope and limitations of this project will then be introduced, followed by the contributions to the research field provided by this project. The chapter ends with an outline of the subsequent chapters.

1.1 Motivation and Background

In recent years, an increasing amount of attention has been given to the field of Big Data, and to the potential insight provided by real-time analysis of data generated from the web and social media. In [32] and [55], for example, we see that real-time analysis of social media posts on platforms such as Twitter can be used in multiple ways – from detecting disasters immediately after they happen, to analyzing the public opinion towards presidential candidates. The increasing interest in analyzing data in real-time has accelerated the implementation of tools and systems which can handle streaming data, as discussed by Zaharia et al. [59]. However, with this great volume of available information and tools, rises a need for handling the effect of *information overload* towards web users: Data streams must be filtered, delivering only the most relevant elements to users. The purpose of this thesis is to help users find relevant tweets for their information needs by ranking data stream elements using scoring functions in real-time.

Twitter generates an enormous amount of data, averaging at over 6000 tweets per second[1]. When analyzing this volume of streaming data, two issues must be addressed to achieve effective processing. The first issue is that tools that handles real-time analysis today tend

[1]DSayce Twitter statistics (2018): https://www.dsayce.com/social-media/tweets-day/

to work by tying together several independent systems. Typically, one must combine storage engines with streaming engines, as discussed by Zaharia et al. [59] and Alkowailet et al. [4]. For instance, a state-of-the-art solution combines MongoDB and Apache Storm. In a scenario where high performance and low latency is crucial, this can lead to unnecessary steps in processing the data is retrieved and provided to users. In addition, using several engines can lead to developers spending more time familiarizing themselves with the systems. The second issue is that analyzing tweets in real-time is difficult: Algorithms must be stream compatible, and the content often includes misspellings, proper nouns are not necessarily capitalized, and the language often contains slang and abbreviations. In order to address these problems, one needs complex real-time processing within a unified system, where the system can both perform to high standards and analyze the streaming data without requiring functionality offered by other systems.

The work in this thesis is built upon a preliminary project[22], called “Filtering and Clustering of Tweets In AsterixDB”, conducted in the course TDT4501 at NTNU. This preliminary project investigated how streaming data could be filtered, and then clustered by semantic similarity using an online, one-pass algorithm.

1.1.1 Problem Description

The core motivation of this project is to investigate how a unified Big Data Management System can be used to handle complex processing of textual streaming data, with the aim of detecting and ranking tweets relevant to a continuous user-defined query in real-time. The focus is specifically set to investigate how to make ranking of tweets in real-time efficient, and how to able to continuously update the list of the most relevant tweets to a user-defined query. Additionally, this task should not require unnecessary storage space. If found suitable for the overall problem, the investigated Big Data Management System could be used by others with similar use cases, without requiring developers to spend time familiarizing themselves with several systems to create a solution.

There are three different challenges to address in the task at hand. First, filtering the data stream is necessary to reduce the amount of data to process. A continuous user-defined query can be created by filtering the arriving elements in the data stream according to the query content. Second, clustering tweets into groups by semantic similarity should be considered, as this could further narrow the search space during ranking. The last – but main – challenge is to continuously update a ranking list as new items arrive, and thus be able to continuously determine the most relevant items to the user.

Filtering and Clustering

The first aspect of this project is concerned with filtering and clustering arriving tweets. First, a query describing the user’s information need must be inserted, stored and then used to create a continuous user-defined query. Then every incoming tweet is filtered based on the user query content, and irrelevant tweets are disregarded. Clustering can further limit the ranking complexity by reducing the vocabulary from terms in all tweets, to only concern the cluster centroids. This will result in fewer items to evaluate when ranking. Then, the system will use the query content to filter out arriving, non-relevant tweets, and only cluster the ones which passes the filter. These clusters can then be ranked with respect to the user query.

Real-Time Ranking of Tweets

The main task of this project is to rank items which have been found relevant for a user query in real-time. In this thesis, relevancy is defined as recently posted tweets with semantically similar content to the user query. For the scope of this project, the goal is to maintain a continuously updated list of the top-k ranked elements in a data stream for *one* user-defined query, at every instant. In order to do so, the system needs to continuously update the ranking result based on new tweets arriving in the data stream. Data which have been considered amongst the top-k items at some point may become outdated when new data arrive, which is why deleting or updating the stored data is crucial. Only the top ranked data should be stored, as this optimizes the storage space.

1.2 Research Goals and Questions

This thesis is a continuation of the work conducted in the specialization project, which resulted in a solution where streamed tweets were filtered, processed and clustered by using a unified Big Data Management System. Building upon this previous work, the goal of the current project is to:

Investigate how ranking can be implemented to retrieve the most relevant set of tweets from Twitter with respect to a user-defined query in an efficient manner.

This means that two aspects must be studied: how to implement a scoring function which ranks continuously, and how to maintain and make use of a user-defined continuous query when filtering a data stream. The main research question (RQ) defined by the specified goal is:

Main RQ: How to make ranking of tweets in real-time efficient with continuous queries?

The following set of research questions are used to guide the research, and can be seen as supplements to the main RQ:

RQ1: How can relevant tweets be detected in a Twitter stream based on a continuous query defined by a user? This research question focuses on handling the information overload problem presented in Section 1.1. It involves both how to set up a pipeline for retrieving elements from a data stream of tweets from Twitter in real-time, but also how to filter these with respect to being relevant for a continuous user-defined query. It also concerns how to actually retrieve the most relevant results of the the user-defined query.

RQ2: How can clusters of Tweets be ranked as relevant with respect to a continuous user-defined query? To further reduce the number of tweets to provide a user, it is necessary to find the top-K ranked elements with respect to the issued query. Ranking individual tweets in a data stream would require frequent updates of the data structure holding the top-k items. This is why the specialization project was concerned with creating clusters where tweets were grouped together based on semantic similarity. To provide the user with the most relevant information, these clusters – and not individual tweets – would be ranked with respect to the query issued by the user. Adding a tweet to a cluster would not necessarily affect the cluster's computed similarity score with regards to the query, and therefore not require frequent changes to the ranking list order.

RQ3: How do continuous queries and ranking affect the retrieved results? As previously discussed, filtering and ranking items in the data stream are methods of reducing information overload and the amount of data to process. The consequences this has on the quality of the results and on processing time must be investigated. Stored data must continuously be updated as new data arrive and the ranked list may evolve over time. Additionally, as the system ought to operate on streaming data, it is essential to also consider the effect filtering by user queries and ranking items have on the total processing time.

1.3 Research Approach

In this section, the overall research approach adopted in this project will be described. A similar thesis [38] investigating the field of continuous queries and streaming data has utilized components from the research process suggested in *Researching Information Systems And Computing* [39]. As several of the research questions investigated by Norrhall are related to the task for this project – only using another combination of systems – parts of the research process followed by Norrhall will be adopted in this project.

Oates describes that research questions arise from personal experiences and motivation, as well as reviewing the literature from the field to study [39]. By reviewing the literature within the field of study, one can find which subjects to undertake by investigating what research have already been conducted and limitations of research performed [39]. The goal of this thesis was formed by the combination of motivation and investigating recent research, and the start of the research process was therefore concerned with studying relevant literature. In the specialization project, the literature review focused on finding state-of-the-art and related work for handling streaming data and filtering. The following work in this project have required a literature review of state-of-the-art for relevance ranking and continuous top-k queries. The NTNU University Library[2] and Google Scholar[3]

[2]NTNU University Library: https://www.ntnu.edu/ub

[3]Google Scholar: https://scholar.google.no

have been used when navigating the relevant literature. To perform the search, specific keywords relevant for the context of the problem to solve were used. As the field of study is in constant change with rapid advancements, the publication date was set as a important criteria in the selection phase after a set of articles had been retrieved.

Research strategies are used as techniques for answering research questions, and typically, one research question has one research strategy [39]. The strategy adopted in this project is *experiment*. The experiments will produce quantitative data by using *observation* as a data generation method. This data is generated by performing systematic observations, where the types of events to observe have been pre-defined. How long specific events take, and all happening within pre-specified time intervals will be observed. Lastly, a *data analysis* of the quantitative data will be performed to evaluate if the pre-specified events behave as expected.

The solution for the task at hand was identified and implemented in an existing big data management system. A minimal viable version for the required system components was first built, and iterating on this functionality until the goal was met. The development part of the project was divided into three implementation phases, covered in Section 4.2. The first phase built a minimal viable version of the components needed, the second phase defined a set of requirements and combined the different components so that ranking could be applied to streaming data, and the last phase implemented the logging technology for record-keeping to observe events in the different experiments. A total of five experiments were performed to evaluate different parts of the proposed solution.

1.4 Scope and Limitations

This project is concerned with maintaining a continuous user-defined query, processing tweets in real-time using a complex model, and implementing a ranking function which continuously rank tweets in a data stream. Because of limited resources, experiments regarding the process just mentioned will be performed on a single computer. In this thesis, the solution is implemented in an already existing Big Data Management System (BDMS), called AsterixDB. AsterixDB is treated as a black box for handling the use case of the project task. Other similar systems suitable for solving the use case could be used as well.

Parts of AsterixDB's features used in this project are still under development. This is an experimental study, and the results of this thesis is affected by the limitations of AsterixDB. This thesis is therefore not only concerned with implementing ranking of tweets in a data stream, but also with exploring how the adopted features makes it easier or more challenging to solve the task at hand.

A general architecture with suggested components for being able to process multiple continuous user queries over data streams have been proposed by Babu and Widom [9]. Furthermore, Park et al. [43] propose an effective index to maintain continuous queries, which is built on the queries instead of records. Given the scope and time, a single continuous query will be maintained as a proof-of-concept when investigating if ranking can be implemented in the unified BDMS. All of the required components suggested above will therefore not be met.

The work in this Master's thesis must both be implemented and written in the spring of 2019. The total time period set for completion is 21 weeks, with a due date on the 13th of June.

1.5 Contributions

Several state-of-the-art approaches to process streaming data in real-time require gluing together several systems. Additionally, many suggested methods for retrieving relevant information to users from a data stream apply the publish/subscribe pattern, not letting users specify keywords of interest nor ranking the retrieved elements. This could result in too many or too few retrieved tweets, with the former situation resulting in information overload for the user. Furthermore, the state-of-the-art methods for relevance ranking are based on machine learning and deep learning approaches. These approaches value high relevance, but lack discussions concerning efficiency. An increasing volume of generated data requires an efficiency-centred focus. When ranking must be performed with real-time requirements, whilst also considering relevance, it is essential to discuss the aspect of speed.

The contribution of this thesis is to apply methods for ranking textual data stream elements in real-time to a unified Big Data Management System, without gluing together several systems. Additionally, this project ought to combine the focus of speed and relevance. The implemented system aims to help users, with a specific information need, avoid the information overload effect. Lastly, the limitations found when using a unified Big Data Management System instead of a combination of systems for the specific use case are pointed out and deliberated for future improvement.

1.6 Thesis Outline

The remainder of this report is structured as shown in Table 1.1.

Chapter	**Description**
1. Introduction	An outline of the background and motivation for this project, the emerged research questions and goal, as well as the scope and limitations.
2. Background Theory	Necessary context for understanding the problem at hand.
3. Related Work	A presentation of related work and state-of-the-art.
4. Continuous Ranking of Tweets in AsterixDB	A detailed description of the implemented solution and alternatives faced during the development.
5. Experiments & Results	An explanation of the conducted experiments and their results.
6. Evaluation & Discussion	Results from the experiments and the implemented system are evaluated and discussed.
7. Conclusion & Future Work	A conclusion is presented and future work is suggested.

Table 1.1: Thesis outline.

Chapter 2

Background Theory

In this chapter, necessary background theory for comprehending the research area and its related concepts are provided. This includes a brief introduction to Big Data and Streaming data, followed by an introduction to text pre-processing and representation, as well as relevance ranking and clustering. Finally, a description of the technologies used – mainly the big data management system AsterixDB – is presented.

2.1 Big Data and Streaming Data

Two terms highly relevant to modern data processing and analysis, and therefore this project, are *"Big Data"* and *"Streaming Data"*. A definition proposed by McKinsey Global Insitute characterizes Big Data as:

Definition 1: Big Data

"(...) datasets whose size is beyond the ability of typical database software tools to capture, store, manage, and analyze" [23].

Gartner, on the other hand, introduces the three V's for characterizing Big Data: *volume*, *velocity* and *variety*, that can be utilized to gain insight, offer decision making and atomise processes. Here volume captures the perspective of size, velocity captures the speed at which the data is created, consumed and processed, whilst variety refers to the unstructured data produced [16]. Others have also proposed *veracity* and *value* as supplementary dimensions of the Big Data definition. Veracity implies trustworthiness – data used for analysis should be checked for credibility, while value indicate that more insight should yield greater value [16].

So, Big Data is the term describing the enormous amount of data that are being generated. With the right tools, this data can be processed to gain insight that can be useful for decision makers, or the common user. However, due to the vastness of the data and the high speed it is generated at, this can be a difficult task, thus creating a need for tools tailored to handle Big Data to gain this insight.

There are two different methods to process data: *stream processing* and *batch processing*. Whilst both methods may deal with a high volume of data, they are different in the way data is ingested. The former will feed data into the system piece by piece, and the latter will acquire a set of data over time before it is sent to the system for processing. Due to this distinction, stream processing is efficient for handling real-time processing of data, opposed to batch processing which often is concerned with analysis of a great volume of data already in persistent storage.

Streaming data is defined as data that is produced constantly by a large number of sources [8]. It is also characterized as an unbounded sequence of ordered data items that arrives at a high rate, with possibly infinite volume [37]. Examples of streaming data are users log files from visiting an application's website, microblog posts or information from geospatial sources.

In a typical Database management system (DBMS), the data arrival rate is limited by the time to read from disk. This differentiates data processing in DBMS from stream processing, where the arrival rate is not controlled by the system. Streaming data also differs from data in conventional databases by mostly being unstructured or semi-structured. This yields a need for a Data Stream Management System (DSMS) that can handle the high-velocity arrival and the unstructured nature of streaming data.

Due to the high arrival rate of streaming data, one of the main challenges in stream processing is related to memory – storing the whole stream is not feasible. To efficiently process each streaming element, this must be handled in memory. This, in turn, requires algorithms which operates in memory. To be able to do this, such algorithms tend to summarize the data stream in some way, reducing the number of data elements to operate on [28], and several approaches for summarizing exist. Using a fixed-size window of the recently arrived elements is a common solution, where only elements in this window can be analyzed.

There are two different methods to ask queries when working with streaming data, *continuous queries* and *ad-hoc queries* [28]. Continuous queries are defined by Babu and Widom as:

Definition 2: Continuous queries

"(...) queries that are issued once and then logically run continuously over the database." [9]

Continuous queries are in a manner constantly executing. Queries issued in a typical DBMS have a fixed answer, whilst the answer to continuous queries issued over a data stream can change as time passes due to the characteristics of streaming data. One example of a continuous query is that the system gives a notification whenever the temperature received from a sensor is below 0 degrees Celsius. Ad-hoc queries, on the other hand, are used to retrieve information about the current state of the stream, usually asked once. If the user has a good indication of what ad-hoc queries will be relevant to ask the system, it is possible to only store the useful parts of a stream or keep summaries of streams.

2.2 Text Representation

Traditional approaches to text representation involve creating term vectors or bag-of-words models: each document is assigned a vector, where each element represents a given word in the document corpus' vocabulary, and the value represents how often the given word appears in the document. A common enhancement of this model is to compute the traditional Term Frequency-Inverse Document Frequency (TFIDF) weight for each term. A term's importance is determined by how often it occurs in the document, including how absent it is in the entire corpus. However, the order of the words – as well as their semantics – are neglected in this approach. The TFIDF method is prone to the curse of dimensionality, and the entire vocabulary must be known beforehand, which would not be the case with a data stream of arriving tweets.

Many of the aforementioned issues can be solved when utilizing methods based on *word embeddings* for representing text. Rather than representing a text as a vector of independent features, losing possible correlations between words, the purpose of word embeddings is to represent all words in a low dimensional space. Every word is assigned to a specific point in this N-dimensional vector space. If a word embedding model uses 300 dimensions, this means that a single word will be represented with 300 different numbers. Here, the intention is to place words which are semantically similar to one another close in either cosine similarity or in Euclidian distance [25]. For instance, the words *permit* and *allow* are similar in meaning, but written differently, not captured by traditional text representations. However, they will be more similar when placed in the embedded space. When representing words using numbers, the natural language is made understandable for computers. Several approaches to word embeddings exist, and these will be outlined in the following paragraphs.

Mikolov et al. (2013) introduces a method for representing words as vectors, and the tool is called *Word2Vec*. This approach calculates vectors by using a neural network, and the result is able to understand the semantic connections between words [33]. Learning the word vectors can be done in two ways, either using a Continuous Bag-of-Words model (CBOW) or the Skip-Gram model on a corpus. To make the concept easier to grasp, imagine adding the word vector *Man* to the word vector *Princess*. When using word embeddings, this calculation will yield the word *Prince*. Additionally, when representing text instead of words, the average "point" of a sentence can easily be computed by finding the average of each dimension of the word vectors. This approach is also language independent, so one could train a model specifically based on Twitter data.

Three other methods stand out in this research field as well, namely GloVe, Paragraph Vectors, and Fasttext. Pennington et al. (2014) proposed *Global Vectors for Word Representation* (GloVe) [44] the year after Word2Vec's introduction. GloVe and Word2Vec both represent words as vectors, but GloVe's approach is to build models in a count-based manner. Training the model involves finding the word-word co-occurrence counts. There exists a pre-trained, available GloVe model which is built on Twitter data, taking the noisy language of Twitter into consideration. To cope with the limitations of order preservation of words in bag-of-words models, Le and Mikolov (2014) suggest a general, unsupervised algorithm which can be used on texts of varying length – *Paragraph Vectors* (Paragraph2Vec) [27]. It can be seen as an extension of the Word2Vec model, but it can create a dense vector representation of text of varying length. The created paragraph vector is connected with the generated word vectors from the paragraph, and these are used to predict successive words. Finally, Facebook's AI Research lab introduces the *FastText* [1] library. In this approach, vectors are learned from each whole word, as well as the n-gram found within each word [11]. This is used to create vector representations for each character n-gram, and words are represented as a sum of these. Even though a word is not mentioned in the training corpus, FastText is able to create word representations even for words absent in the training data. However, building vectors require more time with FastText due to the n-gram level training, whereas Word2Vec and GloVe only use word-for-word.

2.3 Relevance Ranking

The core intent of any information retrieval system is to retrieve and provide relevant information based on users' information needs [10]. Manning et al. defines *relevancy* as follows:

Definition 3: Relevancy

"A document is relevant if it is one that the user perceives as containing information of value with respect to their personal information need." [31]

In order to express the degree of relevance of a given document with respect to a user query, it is common to use a *ranking algorithm* which considers different measures. Ranking algorithms output a list of documents and their calculated score in descending order of relevance, and these lists often contain the *top-k* most relevant documents. The following paragraphs describe different approaches to ranking, and which characteristics to consider when performing ranking in real-time. Then, a set of common measures to score items is presented in Section 2.3.1. Lastly, the standard metrics used for evaluating the quality of the retrieved information is deliberated in Section 2.3.2.

There exists several approaches to the task of ranking documents: The traditional approach is concerned with generating a scoring function which considers different parameters, whilst newer approaches – like learning-to-rank (LTR) – make use of machine

[1] https://fasttext.cc/

learning[30]. However, the latter method is supervised and employs a large collection of pre-labeled training data, whilst the former can be done in an unsupervised fashion.

Traditional approaches for text search can be seen as a top-k problem over a set of static documents, as discussed by [62] Earlier systems tended to use a large collection of stored documents, and retrieved and ranked these offline without having to consider time and memory restrictions. The data was available when and if wanted [28]. Now, with the nature of streaming data, there are other problems to tackle: New items arrive continuously in real-time and must be processed at arrival. If not they will be lost, imposing new requirements for relevance ranking. Instead of calculating the ranking results offline, applications which ought to provide users with information rapidly can now require a ranked lists of items to be produced and updated in real-time with restricted memory.

In the context of data streams, the problem of maintaining the top-k items with respect to user-defined queries by applying a scoring function – here within the data stream window – is formulated as *continuous top-k ranking* [34]. Items in the data stream must be ranked in order to find the top-k relevant items to include in the query result. When real-time updates is a requirement, it is essential to update the list of top-k items continuously, and this must happen as new items arrive. Additionally, a tweet which is relevant for the user query right now, may be outdated within hours. This all depends on how one define *relevance*, and which characteristics to consider when scoring items. Furthermore, the frequency of new items being produced by a data source can be extremely high in the streaming data context, yielding a possibly very high volume and rapid arrival. This is why ranking *all* incoming items in a data stream is highly undesirable, and why *filtering* the data stream before ranking could be beneficial to early disregard irrelevant data for the user query.

2.3.1 Calculating the Relevance Score

When using a ranking algorithm to produce a list of the most relevant items, the relevance can be assessed by different measures. This section introduce a set of common approaches suggested in the literature to score items, where some of the measures also are applicable to the streaming data context.

Text Similarity as a Ranking Measure

When ranking documents with respect to a user query, a commonly used approach is to consider topic similarity, by measuring whether their content is similar or not. Typically, the document and query is represented using the Vector Space Model, as discussed in the beginning of Section 2.2. Deciding the similarity between a query q and a document d is found by calculating the *cosine similarity* between their respective vectors [48]. The cosine similarity between two vectors is defined as:

$$cos(\boldsymbol{q}, \boldsymbol{d}) = \frac{\boldsymbol{q} \cdot \boldsymbol{d}}{||\boldsymbol{q}|| \cdot ||\boldsymbol{d}||} \tag{2.1}$$

As mentioned in Section 2.2, TFIDF is often used as an enhancement to the commonly used bag of words model, which only count the occurrences of terms. When representing text as a vector of the term importance of each term in the vocabulary, the cosine similarity is defined as:

$$cos(\boldsymbol{q}, \boldsymbol{d}) = \sum_{t=0}^{n} \boldsymbol{w}_q t \cdot \boldsymbol{w}_d t \tag{2.2}$$

Where $w_q t$ is the weight of term t in the query's vector, $w_d t$ is the weight of the same term in the document's vector, and n is the number of unique terms obtained from both query and document.

When dealing with streaming data, this measure can be applied if the streaming items are textual. However, when dealing with streaming elements with short text length – like microblogs – the text is often noisy and contains misspellings. Applying TFIDF to texts with these characteristics are found to be counterproductive [36].

Estimated Probability of Relevance as a Ranking Measure

Another approach which considers topic similarity for measuring relevance is language models, which is based on Bayes' formula. This model is used to estimate the probability of a document d generating the user query q. The probabilty of d generating q is defined as:

$$p(\boldsymbol{q}|\boldsymbol{d}) = \prod_{i} p(\boldsymbol{q_i}|\boldsymbol{d}) \tag{2.3}$$

Recency as a Ranking Measure

Only measuring topic similarity is not adequate in all situations. In some applications – for instance when retrieving relevant news – recently published documents are more relevant than older ones. Models which considers the time dimension are denominated as *time-aware ranking*, with recency-based ranking being a subcategory [24]. Recency-based method aims to elevate recently published or updated documents. In applications where this is favourable, temporal features such as creation time, publishing time or temporal expressions in the text [49] can be used to measure relevance.

Applications which deal with streaming data can take advantage of items being fresh at their arrival time. This can be utilized to, for instance, notify users in real-time of new, relevant items. This is why applications using streaming data often considers newly arrived items to be more relevant. However, this also introduces the problem of maintaining a notion of the sequential ordering of the items in some way, and the literature propose two approaches to handling recency when dealing with streaming data:

Sliding Windows: When applying a sliding window over a data stream, the n last seen elements, or all items seen during the t last time units, are maintained [28]. Figure 2.1

illustrates the former type of window, with size $n = 6$. In sliding windows, items must be removed when new ones arrive or the time unit shifts. Relating this to relevance ranking, only items within the specified window are evaluated as candidates for the ranking. As the maintained items changes dynamically, sliding windows will naturally ensure to not hold old items.

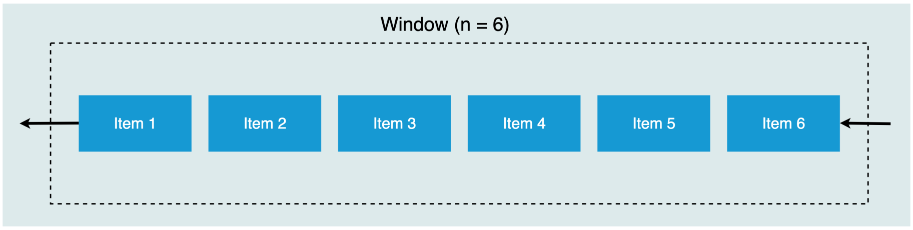

Figure 2.1: Example of a count-based sliding window, where item 1 was the first to arrive, and item 6 is the most recently arrived item.

Decaying Windows: Decaying windows differ from the previous approach by not restricting the items to evaluate by a constant unit [28]. Instead, decaying windows apply a decay function to items as they arrive, giving less weight to the items which arrived earliest. Using this approach, the decay function should be adopted into the scoring function.

Scoring Function

A scoring function can then be defined to be a linear combination of chosen measures which is applied to each document-query pair, with the aim of finding the top-k ranked items. To consider recency when dealing with streaming data, a decay function can be added to the scoring function if adopting the decaying window approach. If sliding windows are adopted, old items will naturally fall out of the window, not being evaluated in the ranking.

Considering the streaming data context, the relevance score of an item is calculated at its arrival time. Additionally, its score may change as time passes by. As the list of the most relevant items can change as new items arrive, this list of the top-k must also be continuously updated.

2.3.2 Evaluation Metrics for Relevance

After documents have been ranked and retrieved, the retrieval performance – how precise the resulting ranking list is – can be evaluated. In *traditional* information retrieval, there are two evaluation metrics which are widely used for this task: *precision* and *recall* [10]. For a given query which has resulted in the answer set A of size $|A|$, there exist a set of *actual* relevant documents, R, with size $|R|$. One can think of R as the ground truth. The documents which are both considered relevant and are retrieved, Ra, is the intersection between A and R.

Precision is defined as the fraction of the retrieved documents in the answer set A which are relevant, Ra.

$$Precision = \frac{|Ra|}{|A|} \tag{2.4}$$

Recall measures the fraction of the relevant documents, $|R|$, which has been retrieved, $|Ra|$.

$$Recall = \frac{|Ra|}{|R|} \tag{2.5}$$

F-measure is single a metric which combines Precision and Recall, and it is tunable to favour one of the metrics or to give them equal importance [31]. Here, β^2 is the tunable parameter, where $\beta = 1$ means that equal importance should be given to both Precision and Recall.

$$F = \frac{(\beta^2 + 1)PR}{\beta^2 P + R} \tag{2.6}$$

However, the abovementioned metrics does not consider the ordering of the retrieved documents. When the retrieved results are ranked, other metrics such as the more recently adopted Mean average precision (**MAP**) is used [31]. It provides a single measure for the entire over all the users' information needs. It is defined as:

$$MAP(Q) = \frac{1}{|Q|}\sum_{j=1}^{|Q|}\frac{1}{m_j}\sum_{k=1}^{m_j} Precision(R_{jk}) \tag{2.7}$$

where Q is defined as the set of queries, and R_{jk} is defined as the set of ranked retrieval results for all documents from the top and to the kth document.

When the task of retrieving relevant information is do be done over streaming data instead of a static collection of documents, these metrics do not necessarily fit as well. There is no ground truth to what is relevant for the specific user query, as the data is dynamic and produced in real-time. Take for instance $|R|$ as defined above, which is the number of actual relevant documents (ground truth). As a continuous user-defined query is *long running* over an *unbounded* data stream, this metric does not directly apply to the context of ranking streaming data, as it is not possible to know this entire set beforehand since the data to evaluate the query against is produced *after* the query is issued. Additionally, MAP is used as a metric to evaluate the system performance over several queries. As this system is scoped to only be able to maintain *one* user queries, it can not retrieve results for more than this single query. However, a set of requirements for how the ranking function must behave in the streaming data context will be defined in Chapter 4, and these requirements must be evaluated by experiments.

2.4 Clustering

To be able to separate tweets which are highly different, and detect tweets which are similar to each other, one can perform partition based clustering for finding a structure in the unlabeled data. This can be seen as an unsupervised learning approach, as prior knowledge about a tweet's label (i.e group/cluster) is unknown. Clustering algorithms utilize information about, for instance, a tweet's content and cluster together tweets which are similar in that manner. There is a range of applications for clustering: tweets could be grouped by topic, by location, or as representing tweets similar to the interest of a user. These algorithms use a similarity measure – either on features or text – to allocate which cluster a data element belongs to.

Some clustering algorithms - like K-Means - requires a pre-defined number of clusters one wishes to separate the data into. However, in some applications, one does not have an a priori understanding of how the data will structure itself. As for instance for Twitter data, one does not know how many different events users are posting tweets about at a given time. Here, the number of clusters is ambiguous: New tweets are constantly posted, and new events can happen at any time. Therefore, tweets must be analyzed at arrival, and clusters must be updated accordingly. Using algorithms which dynamically creates as many clusters as suitable can eliminate the issue of initially choosing the cluster number. Incremental clustering approaches can allocate tweets to clusters if the similarity is greater than a given threshold [17], and simply generate a new cluster if this case is not satisfied. As discussed by Ozdikis et. al [40], several studies have considered incremental clustering as a suitable approach for grouping textual elements which are continuously arriving.

However, traditional clustering methods do not necessarily consider the challenges related to real-time processing of streaming data [37]. Clustering methods which apply to streaming data may be restricted to operate on the data in a single pass (i.e. an arriving element can be read at most *once*), and may have to keep a summary of the elements in the data stream in memory. To be able to handle a possible unbounded stream of data, tweets must be removed from memory by some policy, due to memory restrictions. Repp and Ramampiaro suggests an *online thread*[2] *clustering algorithm* [47], based on the work of Petrovic et al. [45], which considers streaming data as well as an incremental strategy. Only the pre-defined w number of recently arrived tweets are kept in the *window* – restricting the algorithm to only operate on these w elements at a time. By assuming tweets about the same, event will be posted in quick succession, it is reasonable to restrict the number of tweets to only be the last w tweets.

[2]The terms *thread* and *cluster* will be used interchangeably throughout this thesis.

2.5 System and Libraries

This section will present the explicit system and libraries used to handle the task of ranking tweets from a data stream in real-time.

2.5.1 AsterixDB

AsterixDB is defined as a Big Data Management System (BDMS), built to exploit the best aspects of the database world and the world of distributed systems [65] by being a unified system. Whereas other Big Data analytic platforms may lack the capability of either querying or storing data, AsterixDB offers features such as a semi-structured data model, continuous data ingestion, a full query language, automatic indexing and data management [18]. From 2009 throughout mid-2013 the development of AsterixDB unfolded, and the project was open-sourced in 2013 [5]. The architectural decisions made when building AsterixDB were based on the goal *"One size fits a bunch"* – offering several features for a range of use cases without having to affect the system performance. As an example, the work done by Alkowaileet et al. shows how AsterixDB can be used as an end-to-end platform for data analysts with its support for the loading-training-prediction life cycle in data analytics [4].

Modeling Semistructured Data

AsterixDB comprises several core features, with one being the data model, *Asterix Data Model* (ADM). It aims to support unstructured data, which is – as mentioned in the previous section – a common phenomenon in the world of streaming data. The model is a superset, and an extension of, JavaScript Object Notation (JSON) – making all JSON objects valid ADMs, but not necessarily all ADMs valid JSON objects. Furthermore, AsterixDB operates with a notion of *Dataverses*, *Datasets* and *Datatypes*, which must be created in order to store data. Datatypes holds the definition of how records in a Dataset looks like. In addition, Datatypes are flexible – they can either be open, meaning that additional fields can be included – or closed, restricting records in the Dataset to *only* contain these fields. AsterixDB offers a query language, SQL++, which aims to query big semi-structured data, and is specifically tailored to fit the Asterix Data Model.

Architecture (or Manager Node and Worker Nodes)

AsterixDB's architecture is made up of several components with different responsibilities, which together make up shared-nothing computing clusters with an execution layer called Hyracks. The clusters within AsterixDB comprises of one *manager node*, as well as several *worker nodes*. The manager node runs a Cluster Controller (CC) process, which job is to handle incoming user requests via an HTTP API and to manage Hyrack clusters. Additionally, it converts the received SQL++ statements into jobs for the Hyrack level, and for the Job Executor, as well as allocates work to the worker nodes. All Hyracks jobs are comprised of connectors and operators. When the Node Controller (NC) – the process ran on the worker nodes – receive requests for executing different tasks from the CC, the operations to be performed and the connectors within a computation make up a Directed Acyclic Graph (DAG), which the Hyracks execution layer facilitates.

Current Feed Ingestion Process

Another functionality within AsterixDB is *Feed Adaptors* – used when external data is to be ingested into AsterixDB for storage. Their tasks include connecting AsterixDB to external data sources and receiving this data, as well as parsing it into the supported record format, ADM [19]. During the data ingestion process, AsterixDB can execute something called a User defined function (UDF) on the arriving data elements. A UDF can be seen as an extension of typical operations found in the query language [7], and it can be used to for instance pre-process or apply machine learning [4] to the incoming data. This is possible when implementing Java UDFs, where specialized libraries such as Stanford CoreNLP[3], Weka[4] or Deeplearning4j[5] can be imported and used to extract specific information. A feed adaptor can optionally include a UDF, and if it is included, this function will be applied to every incoming record of the data stream. There are two possibilities when defining a UDF: It can either be defined with SQL++ or with Java. When complex processing is required, especially if the processing requires external libraries, defining the function with Java is desirable [19], which is the case in this project. Data then flows into the feed adaptor and through the UDF before it is stored in a dataset in AsterixDB. Every UDF consist of three stages: initialize, evaluate and deinitialize. During this initialize phase, resources which are necessary for the function – typically external libraries or loading models – can be accessed and utilized [19]. In the evaluation stage, the function logic is applied to the incoming record.

When *Feeds* are created and connected in AsterixDB, the compiler is responsible for collecting the definitions of components used for data ingestion, namely the *Feed*, *Feed Adaptor*, *Function*, and the *Dataset* to store data into [18]. Referring to the previous section concerning architecture, SQL++ statements are created into Hyracks jobs. This is the case for the `connect feed`-statement as well, and the dataflow of this specific job is called the *feed ingestion pipeline*. The feed ingestion pipeline in the current version of AsterixDB is made up of three stages – intake, compute and store – each being a Hyracks operator. The intake job contains the *Feed Adaptor* and the data parser, the compute job ensures that the optional *Function* is applied to the data, and the store job moves the data into storage in the given *Dataset*. In the current framework, Hyracks jobs can not access each others data frames at runtime.

[3]https://stanfordnlp.github.io/CoreNLP/
[4]https://www.cs.waikato.ac.nz/ml/weka/
[5]https://deeplearning4j.org/

The New, Decoupled Ingestion Framework

To handle data enrichment (and changes in referenced data) which is not supported in the current ingestion framework, a new, decoupled ingestion framework was suggested by Wang and Carey [56]. This framework differs from the one described in the previous section by adopting a new Hyracks operator called a *partition holder* is added in the new framework. It is a tool for increasing efficiency when passing data between jobs, and it does so by maintaining a queue for the incoming data frames between jobs. The new framework then extends the current framework by attaching a partition holder at the tail of the intake job [56]. It enables the compute job to request (pull) batches of data from the intake job. Additionally, a partition holder which – rather than waiting for a job to pull from it – pushes the data it receives directly downstream is added to the storage job. This lets the storage job push the data it receives from the computing job directly into storage. The computing job is invoked when batches of data arrive from the intake job. The Active Feed Manager, applied to the CC, is responsible for invoking the computing job once per batch. The computing job is called every time it receives a new batch of data, and it runs the UDF on all records within the batch. The batch size is configurable when creating the feed.

2.5.2 Deeplearning4j

The open-sorue library Deeplearning4j[6] offers – amongst other things – an implementation of Word2Vec, the vector representation introduced in Section 2.2. There exists a published, pre-trained model built on the Google News corpus[7], trained on roughly 100 billion words. Around 3 million words and phrases are supported in this model, meaning that it contains 3 million vectors, all with a dimension size of 300. Deeplearning4j can be used to create word embedded representations of textual items in Java. By applying this embedding, each word in a text can be assigned a 300 dimensional vector, each element being a number representing the words place in that dimension. This representation can further be used for calculating an average vector representation (AvgWord2Vec) for all words within a text. This resulting vector can then serve as a base structure for measuring cosine similarity between textual items, where a similarity of 1 means that the two items are semantically similar.

[6]https://deeplearning4j.org/

[7]Model found at: https://code.google.com/archive/p/word2vec/

Chapter 3

Related Work

This chapter explores a selection of the related work to this thesis. First, related frameworks and methods for handling streaming data are presented. Then, the focus is set on related research to retrieve relevant data from a data stream using either approaches based on continuous top-k queries or the Publish/Subscribe pattern. Lastly, state-of-the-art approaches for relevance ranking are considered.

3.1 Related Frameworks and Methods for Streaming Data

With the increasing interest in Big Data analysis, it has become apparent that processing very large datasets using traditonal warehouses and Relational Database Management Systems (RDBMS) poses a challenge. **MapReduce** was one of the early efforts that sought to make Big Data handling easier by concealing difficult aspects – such as scaling – from developers, while also simplifying exploitation of resources in a distributed system [15]. MapReduce utilizes data locality by dividing an application into many smaller jobs which are then shipped to nodes in a cluster that holds the data to be operated on. These nodes then work in parallel, and when all nodes are finished, the results from each node are aggregated into a combined result. The MapReduce API exposes all of this.

MapReduce is the native batch-processing engine in the **Hadoop** framework – the first widely used Big Data framework. The Hadoop eco system consists of tools which enables for instance storing data in HDFS[1] and querying data from Hive[2] distributed file systems. Around 2007, leading companies in the industry, such as Facebook, LinkedIn and Twitter, used Hadoop to deal with the challenge of storing the extensive data volumes

[1]https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[2]https://hive.apache.org/

required for organizations of their size. However, MapReduce had many responsibilities to handle, such as both the resources in clusters and processing, slowing it down. In addition, MapReduce was batch-oriented which brought latency into the picture, as this kind of processing often is concerned with *large*, non-dynamic datasets, not streaming data. Batch-processing yields extended computation time, due to the fact that records are grouped together, and time may pass between the data generation and the actual analysis of the data. This means that applications which consider processing time a critical factor should not use batch-processing, which often is the case for real-time analysis. There is no native support for stream-processing with Hadoop and MapReduce, since it was initially built for batch-processing. This resulted in the need for "stopping and storing" the streaming data, before it could be processed, as well as do aggregation over batches. Today, the need for processing data at a very rapid rate is much more prominent, yielding systems with lower latency and faster processing time than what was offered by Hadoop and MapReduce.

The issue of real-time data computing on continuous data streams is handled by **Apache Storm**[3]. This is a stream-only framework based on event-processing, which was used for real-time analysis at Twittter [52] for over three years. The main components of Storm's architecture are bolts, spouts and the data stream. The bolts and spouts are components of a *Directed Acyclic Graph* (DAG), which represents the processing steps that must be completed on the arriving data. Spouts can be seen as the source of the data stream. From this stream, data is passed directly as tuples to bolts components. Here, each data tuple is processed, producing a new stream as output. Bolts can then be connected to other bolts, together creating a network of small operations to be executed on data elements, since each bolt can have different "tasks". In Apache Storm, all data structures reside in memory. However, as it is a stream processor, it must integrate with other storage systems if the developer requires storage.

Twitter moved from Apache Storm to the real-time stream processing engine **Heron**[4], an optimized, backwards compatible, re-implementation of Storm. The development was initiated in 2014, the engine was open-sourced by 2016, and was donated to Apache Software Foundation in 2018 [53]. Its rise came from the fact that Twitter struggled with – amongst other things – an expanding amount of data [26][53], and Twitter needed an optimized solution to handle this. Results show increasing parallelism using Heron, up to 14x better throughput, and a 5-10x reduction of processing time for tuples [26].

Apache Spark, released in 2010, is another Big Data processing framework, which differs from Apache Storm by being batch-oriented. Its abstraction for data sharing and computation in-memory is made possible through the data structure Resilient Distributed Datasets (RDDs) [58]. Computations in Apache Spark are executed in a distributed environment: Streaming data is split into batches, stored in memory, then parallel operations process these batches. However, this is not the most optimal for real-time processing due to its batch orientation. To make Apache Spark a hybrid – both able to be a batch and stream processor – one can use Spark Streaming, which serves as a wrapper around its batch processes. Spark Streaming performs micro-batching, by separating the stream into stateless

[3]http://storm.apache.org/

[4]https://apache.github.io/incubator-heron/

batches. Apache Spark can integrate with Hadoop, switching out the MapReduce engine which may increase performance when dealing with real-time processing. As it is a general purpose computing engine, Spark requires integration with another system if storage is desireable, similar to Apache Storm.

Processing streaming data is also made possible with **Apache Flink** – another stream processing framework. Similar to Apache Spark, it can handle batch processing, as it unifies stream and batch processing in a single engine [12]. Sinks, sources, operators and streams serves as the central elements in a Flink pipeline: A data stream arrives through a source, operators have tasks to execute on the data stream, which then produces another stream. The sink is responsible for flushing the data out of the Flink system, typically to data storage in another system. This internal pipeline is somewhat similar to that of Apache Storm. An analysis performed by Spangenberg et al. [51] implies that data mining and graph processing has higher performance with Apache Flink than Apache Spark, whilst Spark should be favoured for batch processing algorithms.

“Gluing” together systems is common when building solutions which perform real-time analysis of streaming data. The combination of the stream processing engine **Apache Storm** and the key-value store **MongoDB** is considered as one of the state-of-the-art approaches. A comparison of a solution built with these systems and **AsterixDB** was performed by Grover and Carey [19]. They claim that the combination of Apache Storm and MongoDB depends on more manual work by the user – for instance attaching bolts to spouts – whilst the compiler of AsterixDB will create the workflow automatically. Their results indicate that AsterixDB has better performance, lower latency, and can tackle a higher workload than the opponent.

Spark Streaming for stream processing with **Cassandra** for persistence is also considered a common approach for tackling the same problem as mentioned above. P´a´akk´onen presents a comparative study of Spark Streaming with Cassandra and AsterixDB [41]. The goal is to assess the performance of these stream-based processing systems against each other when processing semi-structured data. P´a´akk´onen concludes that using AsterixDB for stream processing and persistence yields the highest throughput and lowest latency. Spark Streaming is, however, able to read over a thousand tweets a second when reading data directly from Spark using Cassandra’s Java driver. The findings suggest that AsterixDB’s throughput is faster than Spark’s regardless of Spark being attached to a database or not. Additionally, AsterixDB scales better, and can process up to 10 000 tweets a second when using 2 nodes, whereas Spark can process around 2000.

The task of identifying relevant information for users based on streaming data is investigated by Jacobs et al. [13]. By drawing on the functionality offered in AsterixDB, the authors are able to extend this BDMS towards Big Active Data (**BAD**) – a platform which lets users subscribe to topics of their interest. Utilizing both real-time data as well as historical data, the platform is able to provide relevant information. So, when deciding if the incoming data is relevant to a given user, all of the data is considered as a whole. In order to achieve this, the suggested approach has extended AsterixDB with Channels – which they define as continuous queries with parameters which users can subscribe to. The Channels are defined with UDFs written in AQL – the query language used in AsterixDB before

SQL++. Essentially, the publish/subscribe-pattern is used, as the platform operates with a notion of Publisher and Subscribers, in addition to the Channels. The subscriptions are maintained in the Data Cluster and the Broker network serves as a layer between the end users and the Data Cluster. The suggested system is scalable: It can collect data from several data publishers at once, as well as serve information to a vast amount of users.

3.2 Related Methods for Retrieving Relevant Data From a Data Stream

Recently, several studies have investigated how to best find the most relevant documents for a query on data streams. This section focus on the approaches which adopt the continuous top-k queries approach [54][62][64], those which follow the Publish/Subscribe pattern [13], and which who draw from aspects of both methods [46][14]. Either way, one of the most important aspects is time, as retrieving the most relevant documents in a stream must take into account the recency of the stream elements, as well as users' information need either expressed as queries or as selected topics.

Some of the closest related work to this study is the work of Vouzoukidou [54], which studies how to maintain continuous top-k queries over real-time web streams. A set of in-memory data structures for indexing queries efficiently are suggested. The proposed method uses the traditional Vector Space Approach [31] for representing items and queries, weighting terms by TFIDF. Their scoring function considers both TFIDF, other query-independent measures, as well as time decay. The top-k ranking lists of all maintained queries are updated in the event of arriving items. The computational cost of updating scores often when using decaying functions are discussed, but their work uses their earlier developed algorithm for handling this efficiently. However, the work conducted by Vouzoukidou differs from this study as their goal is to maintain any given number of queries. Therefore, they focus on reducing the search space of finding which queries' top-k items are updated when a new item arrives. Additionally, the proposed approach considers textual documents in general, not microblog posts. Therefore, they have designed in-memory indexes which are built to manage a large vocabulary.

The work of Norrhall [38] proposes a solution to real-time ranking of tweets in a data stream, which utilizes a combination of Spark, Kafka and Elasticsearch: Kafka is used to extract data, Spark handles the data processing, while Elasticsearch is as a search engine and for storage. A continuous user query is used to filter a data stream of tweets. Relevance is defined to be the textual similarity between a tweet and the user query. A TFIDF approach is used for scoring tweets, and item freshness is also considered by employing a decay function in the scoring function. Experiments showed that the system could handle 15 000 tweets/second. However, the work in this study differs from that of Norrhall by attempting to maintain a continuous query and ranking streaming data by leveraging a unified BDMS instead of combining tools for storage and stream processing.

Zhang et. al [62] suggest a framework for updating the top-k results for a large number of queries, called Minimal Reverse ID-ordering (**MRIO**). A server maintains a number of continuous top-k queries, and the k highest ranked items in the data stream for each of the queries are retrieved. By building indexes over queries instead of documents, applying reverse ID-ordering instead of frequency-ordering, and lastly adopting a technique for calculating the score of arriving items by considering as few of the queries as possible, their framework outperforms state-of-the-art. The motivation behind indexing queries as described is to find those queries whose top-k results would be altered by an arriving document in an efficient manner. Their scoring function considers textual similarity between the vectors of a query and a document, and incorporates a variant of a decay function to handle document freshness. Lastly, they uses the execution time of updating the top-k results when a new item arrives as a performance metric.

Zhu et. al. [64] suggest a self-adjustable framework, **SAP**, for supporting top-k continuous queries. The problem to tackle in this paper is to maintain a continuous top-k query which returns the k objects within the query window with the highest scores. By utilizing sliding windows, they further partition the window into sub-windows. In doing so, they reduce the incremental maintenance, by maintaining top-k objects and other candidates in each partition. The meaningful objects of each partition is maintained in an index structure which utilizes a type of self-balancing binary search tree. Adopting these techniques, the update caused by the window sliding only affects *one* partition in the window. The approach is able to obtain logarithmic complexity when incrementally maintaining the set of candidates.

In the task of finding items of interest to users, many approaches make use of the *Publish/Subscribe* (Pub/Sub) pattern, where users subscribe to already defined topics instead of self-issued keywords. Here, messages – marked as being related to a specific topic – are published by a publisher, and users can then subscribe to these topics. However, approaches using this pattern differ from that of continuous top-k queries, as Pub/Sub approaches typically does not perform relevance ranking of items, and the topics are typically pre-defined, not defined as a query issued by a user.

Jacobs et al. [13], as mentioned in Section 3.1, propose a system called **BAD** which adopts this pattern. Users can identify what they want to subscribe to using the interface provided by BAD, and the user input is used to create a UDF. The way this works, is that the user input is stored in a AsterixDB *Dataset*, and this information is then extracted from storage to create such functions. Their suggested approach differ from that of other Pub/Sub systems by considering arriving items' connection to other, stored data when deciding whether an element is of interest to a user. Additionally, the resulting items delivered to the users can be enriched by other data. However, the work in this project differ from that of BAD by letting users define their information need by a query, instead of subscribing to topics.

Others who have have studied the problem of how to prevent information overload by detecting relevant elements in a data stream are Pripužić et al. [46]. They propose a distributed approach based on the Pub/Sub pattern and sliding windows, called **Top-k/w publish/subscribe**. A subscription is defined as a time-independent scoring function, and they suggests different scoring functions, with one of them being relevance ranking using cosine similarity between the vectors of the subscription and the publication. When a new item arrives, it is evaluated against other items in the window to find which items belongs to the top-k. This approach differ from that of normal Pub/Sub approaches by letting subscribers specify the number of publications they wish to retrieve. The proposed approach is able to keep a set of candidates in memory, as they may become relevant for the top-k list at a later time. In this paper, the authors also highlight the issues which can occur in Pub/Sub systems, namely users receiving too many or too few publications and the lack of ranking.

Chen et al. [14] suggest an approach which is able to consider Temporal Spatial-Keyword **(TaSK)** queries when retrieving the k most relevant geo-textual items for a user query from a data stream. In order to calculate which objects are most relevant, their approach leverages recency, text relevance and spatial closeness. A language model is used to measure text relevance, by calculating score for each term. To measure recency, they have implemented a decay function.

3.3 Related Methods and Frameworks for Ranking and Clustering

The concept of ranking is vital in the task of retrieving and providing relevant information to users, and this field has therefore been an object of research for decades. With the emergence of machine learning techniques and deep learning approaches, methods which utilize these for performing relevance ranking have evolved, and at the same time led to progressions in the research field. As many currently are researching this, this indicates the problem relevance.

Real-time ranking and clustering data from microblogs have been considered in several studies, but many of these perform ranking without considering a user query. One study which investigate the task of applying ranking to clusters when evaluating microblog posts in real-time is the study of Abdelhaq et al. [1]. They propose a framework called **EvenTweet** which aims to detect events – such as for instance local emergencies – in real-time from a stream of Twitter data. They maintain the most recent tweets within a time-based sliding window, which is relevant for the domain of this thesis. The spatial signatures of the tweets' keywords are computed, and keywords are assigned to the cluster with the most similar centroid in one-pass. This similarity is found by calculating the cosine similarity between the spatial signature and cluster centroid – which is an an average vector of all spatial signatures in the cluster. Each time the sliding window shifts, the clusters' scores are updated, by considering the scores of all keywords in the clusters. However, as cluster scores only are updated when a time frame ends, this method is not able to detect events in real-time. Lastly, they do not consider any semantic correlation, as discussed by [61].

Similar to EvenTweet proposed by Abdelhaq et al. [1], Zhang et al. [61][60] also investigate local event detection in a stream of Twitter data. The papers suggests a method called **GeoBurst**, and a newer version called **GeoBurst+**. Geoburst retrieves all localized event within a specified time window, and also updates the list of events continuously in an online fashion. They are able to detect geo-topic clusters of tweets in the query window which are semantically alike and close in geographical distance. These are considered candidate events, which later on are ranked, finding the spatial and temporal bursty ones. A data structure called *activity timeline*, which contains summaries of the stream, is used when ranking the candidates. When the query window shifts, the resulting list is updated in real-time. In GeoBurst+, the framework is improved by using word embeddings to capture the semantics of tweets.

The ranking algorithm, **TimeRA**, proposed by Liang et al. [29] does take a query into consideration. The algorithm combines the output of different microblog search algorithms to a single, more relevant ranking list. By considering time, they found an increase in performance in comparison to state-of-the-art approaches to rank aggregation in microblog search. This indicate the importance of considering time when ranking microblogs. The combined score of documents in all lists within the time window is then used to find the total rank of a document. Given a small time frame with presumably relevant documents, documents which are posted in the neighbourhood of a highly ranked document will be rewarded. The complexity of the algorithm is low, and the authors show that the approach can handle near real-time requirements. and are also able to merge ranking lists in near real-time.

Yin et al. proposes a one-pass clustering algorithm, **MStreamF**, which operates on short text in streaming data. This approach is model-based, aimed to deal with limitations found within typical similarity-based stream clustering – the need for defining a similarity threshold. The suggested methods tackles this by being able to estimate the probability of a document belonging to clusters. They adopt a vector representation for the clusters which lets documents easily being added or removed from them. This representation stray from the normal path of using the mean document vector, to instead make use of a tuple representation, containing a list of term frequencies in the cluster, as well as the number of terms and documents within it. This approach operates on batches, and documents are not deleted from clusters before the batch they belong in is outdated.

Ozdikis et al. [40] propose an approach to event detection by performing clustering and burst detection. A hybrid sliding window is adopted, which maintains both the aspects of time and count. Tweets are represented using the bag-of-words approach. They find similar terms by using a co-occurrence technique, and the generated vectors from this process are used to calculate the cosine similarity between keywords. A cluster is represented as vector of different features: the cluster centroid vector, its creation time, term-frequency pairs and the ids of tweets in it. The tweet and cluster centroid vectors are expanded using the similar terms found by the co-occurrence technique. Incremental clustering is then applied to these, and this process allocate tweets to clusters if the similarity between the tweet and its most similar cluster centroid is higher than a pre-defiend threshold.

As mentioned, many recent approaches are utilizing deep learning for ranking documents, but these do not consider the real-time aspect as the approaches listed above, nor the domain of microblogs. However, they do consider queries. One study which has worked on the task of applying deep learning when computing relevance for a document with respect to a query is the study of Guo et al. [20]. The paper points out the differences between semantic matching and relevance matching, and their approach is concerned with the three factors in the latter perspective, namely exact matching signals, the query term importance, and diverse matching requirements. Guo et al. argue that previous deep learning approaches only considered the former perspective. Their deep relevance matching model (**DRMM**) contains three parts: a matching histogram mapping, a feed forward matching network, as well as a term gating network in which scores from all query terms are aggregated. When building local matching signals between texts, they adopt word embeddings and use cosine similarity to measure the semantic similarity. However, aspects of speed is not included, so the efficiency of the algorithm can not be discussed.

Similar to Guo et al. [20], Pang et al. [42] proposes a deep learning approach for relevance ranking of documents with respect to a query, and they suggest an architecture called **DeepRank**. The model exploit the three steps of the human judgement process to assess relevance, distinguishing it from Guo et al. approach. By modelling these steps, the relevant contexts are first found, then the local relevance in these contexts are measured, and an overall relevance score is produced by aggregation. When measuring this score, both the importance of query terms, proximity heuristics, as well as exact and semantic matching signals are considered. In summary, the approach uses a detection strategy, a measure network based on a convolutional neural network, and the score aggregation is computed using a gating network and a recurrent neural network. Their studies have shown that DeepRank outperforms other IR methods using deep learning, and its performance also able to surpass that of current LTR approaches. Nevertheless, the approach is supervised as it requires a training phase and a great amount of data. Additionally, the processing time of finding the relevance score of a document is not focused on in this paper, and therefore it is not possible to state whether their approach could handle real-time requirements of ranking.

Collectively, the two abovementioned approaches focusing on machine learning and deep learning highlight a need for considering speed if these advances in the field of IR are going to be applicable to real-time ranking. The ranking performance of the listed approaches are state-of-the-art in terms of relevance, but lack discussions concerning the efficiency of the algorithms. If such methods are going to be suited for ranking streaming data elements in the future, the time aspect will be important to take into consideration.

Chapter 4

Continuous Ranking of Tweets in AsterixDB

This chapter will introduce the implemented system which aims to solve the overall research goal of this thesis:

To nvestigate how ranking can be implemented to retrieve the most relevant set of tweets from Twitter with respect to a user-defined query in an efficient manner.

First, an overview of the proposed solution and its core components will be given. Furthermore, the theory, related work and concepts chosen to apply to this study will be outlined in the theoretical solution. A presentation of the development phases and the different alternatives faced during these will then be given. Lastly, the final system is described, by detailing the implementation of each required component for handling the task at hand.

4.1 Solution Overview

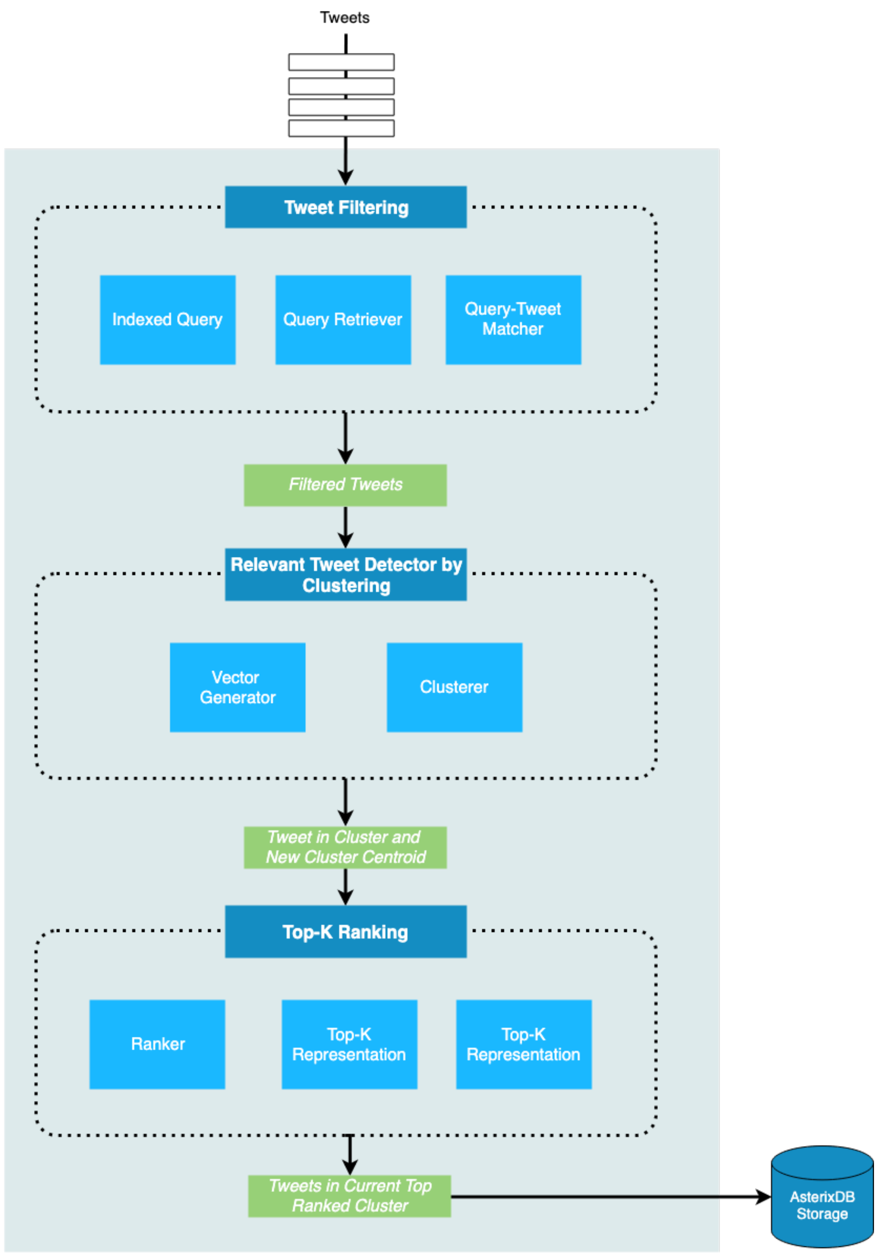

Figure 4.1: An overview of the proposed components used for detecting and ranking tweets in a data stream with respect to a user query. The green boxes represent the output of each specific part of the system, used for input for other parts.

Figure 5.9 shows an overview of the proposed solution. All functionality is maintained within a UDF which can be applied to a *DataFeed*, and therefore will be ran over every incoming item. Tweets in the data stream are pushed to the system, and are met by a *Tweet Filtering* function. A user query is indexed and made available through the *Query Retriever* in the UDF. It is used by the *Query-Tweet Matcher*, which evaluates the arriving tweet against the query. If the tweet passes the filter, it is sent to the *Relevant Tweet Detector* part. Here, the *Vector Generator* is used to map the tweet, the cluster centroids, and the user query to the same n-dimensional space. The *Clusterer* is responsible for grouping together tweets by semantic similarity using the *Model*, and to update the cluster's centroid if a tweet was added to (or removed from) it. After the tweet is clustered, and the cluster centroid is updated, the data is sent to the *Top-k Ranking* part. The *Ranker* component calculates the new similarity score between the cluster centroid and the user query, and evaluates whether the *Top-K Representation* should be updated based on the new cluster score, and whether the topmost ranked cluster has changed or is the same as in the previous evaluation. Only tweets found to reside within the topmost ranked cluster can be inserted into storage. Please disregard the double *Top-k representation* component, it is only supposed to be one.

4.2 Theoretical Solution

This section will describe the proposed approach to real-time detection and ranking of relevant tweets with respect to a given user query.

4.2.1 Retrieving Top-K Items In a Data Stream

The main goal of this thesis is to investigate how ranking can be implemented to find the most relevant items from a data stream of tweets with respect to a user-defined query, and if this use case is solvable when employing AsterixDB, a unified Big Data Management System. A user can issue a query with either keywords or a full sentence which represents the user's information need. This query should be stored/indexed in AsterixDB, in order to make it possible for the system to consider information from the query while evaluating the data stream. The user-defined query should both be used to filter the data stream, and when ranking items to find the top-k ones.

By maintaining either a sliding window or applying a decay function – as introduced in Section 2.3.1 – the time aspect, giving preference to fresh items, would be considered. Either only the recently arrived tweets would be considered during evaluation, or the score of older tweets would decrease as time passes by.

As an average of 6000 tweets are posted each second, applying a filter on the data stream based on the query content would early disregard irrelevant items. This would reduce the search space to not consider all arriving tweets while ranking, only the ones initially found relevant for the user query. As clustering can be used to reduce search space [10], i.e. reduce the set of items an algorithm searches, clustering can be applied to reduce the number of items to rank items with respect to a user query. Therefore, by taking the tweets which have passed the filter and grouping them into clusters, the search space could potentially be further reduced: only the cluster *centroids* could be evaluated with respect to the user query while ranking. As discussed in Section 2.1, the nature of streaming data requires real-time processing of its elements to be handled by efficient algorithms which can operate in memory. Repp et. al [47] proposed an online, memory-efficient, incremental clustering algorithm, based on the work of Petrovic et. al [45]. By building on this approach, but evaluating tweets against cluster centroids instead (as we have not performed classification beforehand), one would obtain clusters of semantically similar tweets. Incremental clustering have also been adopted by several others which studies the task of grouping continuously arriving textual items [40][50][57][63]. An incremental approach is adopted as the number of different topics discussed on Twitter is vast, and therefore no prior assumption about the number of clusters can be made.

Limiting the information overflow of a user can be achieved by ranking the elements in the data stream, retrieving only the most relevant information for a given user query. To assess which items in the data stream are ranked the highest, one should use a linear scoring function to rank these, as discussed in Section 2.3.1. Mouratidis et. al [35] suggested to use a scoring function based on similarity when asserting the relevance of items in a data stream. The approach used in this thesis is built upon this method, and have also been used by others investigating similar problems [54]. However, the concept of similarity in this thesis is oriented towards *semantic* similarities between items, as presented in Section 2.2. How this representation can be applied to this study will be described in Section 4.2.2.

In order to be able to constantly update the ranking list efficiently, the top-k ranking should be represented by an in-memory data structure. As inserting new elements to the top-k list only should happen when an element is ranked higher than the *lowest* element currently in top-k, it is valuable to adopt a data structure that has both fast retrieval of the lowest score, and can efficiently insert new elements. By adopting a data structure with these features, the requirements of fast, in-memory updations of the elements within it would be handled.

The top-k elements will dynamically change due to the nature of streaming data. At every time instant, the top ranked thread[1] is assumed to hold the most relevant tweets for the user's information need. Only tweets residing within the top-ranked thread should be persisted into a dataset in AsterixDB. Furthermore, a user interface should show the retrieved tweets, and dynamically change if the top ranked thread is changed: If a new thread outranks the previous top-ranked thread, the previously inserted tweets should be deleted from storage, as these are not found the most relevant anymore. This will ensure that storage space is not occupied by irrelevant data, and that the retrieved tweets in the user interface are updated, always showing fresh, relevant tweets to the user.

[1]Thread and cluster are used interchangeably throughout the thesis.

4.2.2 Tweet, Thread Centroid and Query Representation

As tweets are short documents – limited to 280 characters as of 2017, with an average of 33 characters – representing all tweets by vectors from the entire vocabulary seen would require large vector dimensions. As the language used in tweets is noisy, with frequent misspellings and abbreviations, the vocabulary size would be affected by these if considering all distinct terms as in normal vocabulary creation. Naveed et. al [36] carried out an analysis which found that approximately 85% of all tweets only include each term at most once. Such characteristics make TFIDF less suitable for the task of scoring tweets.

The approach in this study differs from that of Vouzoukidou et. al [54] by considering methods based on word embeddings rather than the traditional TFIDF vectors. When using Word2Vec to create word embeddings, each word in a text is mapped to a vector $\mathbb{R}^n$, where n is the dimension size, as discussed in Section 2.2. Furthermore, by averaging each of the vectors generated per word in a text to *one* vector $\mathbb{R}^n$, one can obtain constant space for the entire text. As there exists pre-trained models for mapping text to word embeddings, it would not be necessary with a training phase for creating a model.

By using word embeddings to both represent the content of tweets, the threads' centroids and the user query, all are mapped to the same n dimensional space, and similarities can easily be computed. This further means that using the possible very large vocabulary V of words from tweets as the vector dimension is avoided. Having all three representations mapped to the same space, calculating the similarity between these vectors would indicate how semantically similar the content of the vectors are.

Similarity

As the data is represented as n dimensional vector by using Word2Vec, the cosine similarity measure defined in Equation 2.1 in Section 2.3.1 can be adopted. The semantic similarity between a tweet T and a thread centroid C can then be found by calculating the cosine similarity between their respective vectors. Let V_T be the tweet's averaged Word2Vec, and let V_C be the centroid's averaged Word2Vec. The similarity between V_T and V_C is found by:

$$cos(\boldsymbol{V}_T, \boldsymbol{V}_C) = \frac{\boldsymbol{V}_T \cdot \boldsymbol{V}_C}{||\boldsymbol{V}_T|| \cdot ||\boldsymbol{V}_C||} \tag{4.1}$$

The semantic similarity between a thread centroid C and a user query Q is then also found by calculating the cosine similarity between their respective vectors, V_C and V_Q:

$$cos(\boldsymbol{V}_C, \boldsymbol{V}_Q) = \frac{\boldsymbol{V}_C \cdot \boldsymbol{V}_Q}{||\boldsymbol{V}_C|| \cdot ||\boldsymbol{V}_Q||} \tag{4.2}$$

4.3 Implementation

In this section, the final implemented system aiming to solve the overall research goal will be presented. First, a thorough description of what have been done in the different implementation phases in this study is described. The motivation behind this description is to highlight the faced alternatives, the difficulties that arose, and the decisions which were made. Furthermore, a presentation of the final system is given, describing the preconditions and how the different components are implemented.

4.3.1 Initial Phase

As stated in Section 1.3, the purpose of the first phase was to study the related work and state-of-the-art within the field of ranking and continuous top-k queries, and to implement the minimal required components for the system. It was also concerned with an investigation of how to improve the filtering and clustering methods from the initial solution built in the specialization project.

Development of Required System Components

Even though a lot work with developing the initially required components and features was conducted in the specialization project, the provided solution – from now referred to as the *initial system* – was not made to tackle the task for this thesis. In this study, the system needs to handle the maintenance of and using information from a user query, and ranking arriving tweets from the Twitter API in real-time. In the *initial system*, the user query was defined by a static set of keywords declared within a UDF. One of the goals of this phase was therefore to investigate how a user could issue a query, in which the system could base its filtering and ranking on.

As explained in Section 2.5.1, AsterixDB operates with a notion of *Datasets* and *Datatypes* in order to store data. Therefore, a *UserQueryType* should be implemented to describe queries, and a *UserQueries* dataset should be created to hold records of user queries.However, exploring how to insert this data called forth alternative approaches: Queries from users could either be inserted through socket based insertion by creating a *DataFeed*, which would require the development of a separate user interface. Or, queries could simply be inserted through the interface provided by AsterixDB using SQL++ statements. As the main task of this thesis is concerned with efficient ranking of elements in the data stream, it was decided to insert queries through the AsterixDB query interface, as this interface is already provided as a native part of AsterixDB.

Furthermore, the *initial system* was set up to retrieve tweets from a DataGenerator, where the data were pre-processed to be ADM compliant before it was sent to the *DataFeed* for insertion. For this project, a connection should be set up between AsterixDB and the Twitter API instead, making Twitter data continuously flow into the system for processing. AsterixDB offers native functionality for establishing a connection with this API, and parsing tweets directly to the ADM format, within what is called a Twitter adaptor. In order to use the adaptor, one must create a AsterixDB *Feed* and configure it to use this specific adaptor. For the applied UDF to be able to process Twitter data, the argument passed to it must be defined to be of a specific *Datatype* created to hold tweets. As AsterixDB provides functionality for both connecting to the Twitter API, processing elements in the data stream as they arrive, as well as persisting them, AsterixDB was found suitable for the overall problem of this study.

The last component considered was the ranking component. In order to calculate the relevance of elements with respect to a user query, it was necessary to implement a scoring function and ensure it was applied to all tweets in the data stream. A baseline scoring function was therefore implemented as a Java UDF to work as a minimal viable solution for the ranking component. Both the user query and the arriving tweets were mapped to a TFIDF vector representation. Then, the similarities between tweets and the user query were scored by following the definition in Section 2.2. At first, an attempt was made to create a TFIDF *Datatype* within the Java UDF, and to output records of type *TweetRelevantType* which included the TFIDF score, as shown in Listing 4.1.

```
CREATE TYPE TweetRelevantType AS CLOSED {
  "id": int32
  "score": TFIDFType,
  "tweet": string
};
```

Listing 4.1: ...

But, this approach became rather intricate, as it was necessary to use AsterixDB's APIs to create a *derived* type – a object – for the score before outputting the record. It was therefore found more straightforward to use Java classes for maintaining the scoring information, and outputting *primitive* AsterixDB types instead However, this implementation did not focus on quality, only on how to apply a scoring function to all arriving tweet.

Challenges During the development of the required system components, an issue with connecting AsterixDB to the Twitter API became apparent. When issuing a SQL++ query for starting the *DataFeed*, the query never finished executing – it was pending until manually terminated. In other words, the *DataFeed* was never started and data could not be retrieved nor persisted. The system was at this time configured to use the same AsterixDB version as in the specialiaztion project. Possible explanations thought of at first included issues with connecting to the Twitter API, wrongly defined *Datatypes*, *Dataset* or *DataFeed*, or just a bug within the AsterixDB version. The data definition language (DDL) statements used for reproducing the issue was sent to a developer at AsterixDB, to investigate whether the same problem occurred in his environment. As connecting to the Twitter API worked in his environment, wrongly defined DDLs and not being able to connect to the Twitter API were ruled out. Therefore, it was investigated if the AsterixDB version – cloned from the master branch during the autumn of 2018 – caused the problem, by ensuring that the newest version of the master branch was cloned instead. However, this did not solve the halting query issue either. After a thorough analysis of all components which could cause the error, the explanation was found to be a change in twitter4j

APIs in their newest version, 4.0.7. This threw an exception in one of the Node Controller processes, causing the halting query. Switching to an earlier version of the twitter4j dependencies, namely version 4.0.3, solved the issue with the halting query.

Improve Filtering and Clustering

After the preliminary versions of the missing components were implemented, the focus was set to improve two of the components built in the specialization project: the filter and the clustering algorithm. In the *initial system*, word embeddings were utilized to group together semantically similar tweets in the sliding window. This was useful for making the space dimensionality constant as all tweets was mapped to the same 300 dimensional space. However, the implemented algorithm assigned tweets to the same cluster as their most semantic similar tweet, and did so by a manually set similarity threshold. This could cause some unwanted behaviour, if for instance the most similar tweet found was considered an outlier in its cluster. An outlier is formally defined by Hawkins as:

Definition 4: Outlier

"... an observation which deviates so much from the other observations as to arouse suspicions that it was generated by a different mechanism." [21]

These are created *"When the generating process behaves in an unusual way"* [3]. When looking at Definition 4, it is clear that the assignment of tweets to threads was generated by different mechanisms in the *initial system*: The same criteria was not used in the assignment process, as the most similar tweet very likely is different from tweet to tweet. By representing a thread's centroid using word embeddings, incoming tweets could be evaluated against these centroids instead of letting the allocation be based on Tweet - Tweet similarity. If the number of threads are less than the number of tweets in the window W, this change could also reduce the complexity, as every Tweet - Centroid pair are calculated instead of every Tweet - Tweet pair. In order to achieve this improvement, the algorithm should be modified to hold a suitable data structure representing thread centroids. This change would require a new method for setting a thread's centroid, which should consider all tweets contained within it at a given time. This also implies that a thread's centroid should be updated as tweets are added or removed from it, to reflect the dynamic nature of the cluster. In the clustering approach in the specialization project, the clustering algorithm could in theory produce a larger maximum distance for every new, assigned tweet, as a centroid was not evaluated when assigning tweets to clusters.

Another aspect considered in the initial phase was how to improve the filter. The UDF implemented in the initial system contained a filtering function which was able to discard tweets which did not mention a specific, static keyword, nor mentioned a relevant location entity. However, the initial system was only able to mark the "relevant" field of the record to output as irrelevant – they were still persisted in AsterixDB. This is not optimal for a solution which is going to rank tweets in real-time, as this would result in an enormous amount of stored tweets. Additionally, the filtering function could be improved by performing query expansion [31] by automatically adding semantic similar words using the Word2Vec model, which is beneficial due to the short length of tweets.

While working on the specialization project during the last months of 2018, we were informed by a developer from the AsterixDB team that there existed a feature in the *DataFeed* not described in their documentation: *query predicates*. Query predicates could be used to only persist tweets containing relevant content from the data stream. It can be compared to a `WHERE` clause in SQL: It is able to discard data not matching a given query predicate, and only return data which does. Referring to the description provided by a developer at AsterixDB in Figure 4.2, data which have passed the *filter function* can be sent to the *applied function* (UDF) for further processing or be directly persisted in AsterixDB. Using unnecessary storage space on irrelevant tweets could be avoided if the query predicate functionality is adopted. A preliminary version which applied a Java UDF query predicate to a *DataFeed* was implemented and tested on a set of pre-defined query keywords on a small dataset, to ensure that filtering by query predicate worked.

Figure 4.2: An example provided by a developer at AsterixDB, on how applied UDF functions differ from filter functions.

Challenges At the end of the initial phase, some time was spent looking into the features offered by AsterixDB, and how a stored user query could be utilized for filtering within a UDF. As the problem to tackle in this study is real-time ranking of results for a continuous user-defined query, it was found important to consider if the system could adopt to changes made to the user query. In February, a developer at AsterixDB made us aware of limitations in the current ingestion framework by referring to a paper which discussed stateless and statefuls UDFs [56]. They differ in that stateful UDFs utilizes either resource files or external resources, whilst stateless UDFs only considers the arriving record. External resources could for instance be records in a dataset containing user queries. If other than the arriving record are accessed, this will – in the current ingestion framework – potentially result in query plans which can not be evaluated, or it can create intermediate states which does not consider updates in the referenced data. The paper discusses that connecting the stateful type of UDF to a *DataFeed* could cause this unwanted behaviour, if for instance the referenced data is modified during the ingestion process. These attached UDFs are therefore restricted to being stateless in the current ingestion framework. There was then a moment of realization: If the current ingestion framework is used, the system to implement in this study would not be able to adopt to changes in the user query. However, a way to tackle this issue was presented in the paper: a new, decoupled ingestion framework. Recall that this framework was briefly described in Section 2.5.1. Due to the task at hand, this new framework was found both relevant, necessary and engaging to investigate. As AsterixDB's master branch did not contain the new framework, a compiled snapshot version was obtained by mail, shown in the first entry of Table C.1. Henceforth, many of the parts to implement was tested using the new ingestion framework as well, to see whether it was more suitable than the current framework for the task in this study.

4.3.2 Second Phase

The previous phase set up components for connecting AsterixDB to the Twitter API, storing user queries, applying a simple scoring function to tweets, extending threads to hold a word embedding based centroid, and to filter by query predicate. The second phase was first concerned with exploring how to combine the stored user query and filtering function into a continuous user-defined query. Afterwards, the focus was set to improve the ranking algorithm, explore how to maintain a data structure in-memory of the top ranked items, and how to continuously update this. Lastly, it explored how to combine both the filtering function and the scoring function with the user query. Thus, the goal of this phase was to explore possible solutions to RQ1, RQ2 and RQ3. Several implementation alternatives were considered during this phase, and the chosen approaches will be summarized in the end of Section 4.3.2.

Continuous User-Defined Query and Filtering

As mentioned in Section 2.1, continuous queries – also called *standing queries* – are defined as: *"... queries that are issued once and then logically run continuously over the database"* [9]. The start of the second phase was set to explore RQ1, and investigated how to combine the filter function with a user query, which in turn would serve as a *continuous* user-defined query following the abovementioned definition. In order for a query to continuously run on incoming data, it is necessary to define it in advance. By basing the filter on a user-defined query and afterwards attaching it to a data stream, the filter will run on every incoming record and consider their relevance to the issued user query. As described in Section 4.3.1, a preliminary version which filtered by using a query predicate was developed, and *Datatypes* and a *Dataset* to hold user queries was implemented. However, the filtering and the user query was not yet combined at this stage. Extending this simple filtering into a continuous query required it to switch out the static keywords with content from the indexed, user-defined query.

Alternative Approaches to Filtering Function With the above requirement in mind, three approaches for implementing a query-based filtering function became evident:

1. *Alternative - Implement Filtering Function in a Separate SQL++ UDF:* Implement a SQL++ UDF which queries the data stored in the *UserQueries Dataset*, and holds the result of this query in a variable. The filter should perform a matching between the words of arriving tweets and the query terms, to see if they contain one or more similar words. In this alternative, one can only do basic string matching filtering. It would not possible to utilize word embeddings for performing query expansion, which would have been beneficial due to the restricted vocabulary in the tweets. However, it is possible to access the indexed query without sending requests to the AsterixDB API, and thus the user query can retrieved and used as a filtering predicate without slowing down the pipeline. A test version of the SQL++ UDF was implemented and attached to the *DataFeed* as a query predicate. This showed that the query predicate was able to filter elements based on the user query content.

2. *Alternative - Implement Filtering Function in a Separate Java UDF:* Implement a Java UDF which sends a POST request to the AsterixDB API during its initialization phase. This request should select data from the *UserQueries Dataset* and store it in a class field in the UDF, making it accessible during the UDFs lifespan. In the UDFs evaluation phase, the keywords in the user query are matched with the words in the arriving tweets. In this approach, it would be possible to perform query expansion by using word embeddings, as the adopted Word2Vec model offers functionality for retrieving a specific word's n most semantic similar words. However, it is not feasible to use the model in two different Java UDFs (read: both in the filtering function and in the applied function). This is because it would require two running UDF instances both reading the model into memory, as the model can not be shared between UDFs. Furthermore, if using the new ingestion framework presented in Section 2.5.1, the initialization phase would be invoked after each batch, which means that the model would be read into memory again. As a result, if one filtering UDF and one main processing UDF are implemented, and if the ingestion framework is adopted, the model would be read into memory twice per batch. This is not scalable with with regards to memory usage. A separate Java UDF for filtering was implemented as a test version following the proposed approach in this alternative. The query predicate was able to disregard irrelevant data and pass the relevant data to the applied UDF. A class was also implemented to test whether the query could be retrieved by sending an API request to the AsterixDB API.

3. *Alternative - Not Implementing Filtering In a Separate UDF:* Implement the functionality for filtering arriving items within the same UDF responsible for clustering

and ranking items. This approach will be able to utilize the model for both scoring items and filtering items, as it can perform query expansion by finding semantic similar words to the query keywords. As this alternative only requires one UDF, the problems related to running two UDFs, as discussed in the previous alternative, would be avoided.

Challenges A challenge emerged while testing the 2. alternative to the filtering function described above. The intent was to retrieve the user query through a request to AsterixDB's API. This was not completely straightforward: When curling the request in Listing 4.2 from the Java UDF, the response did not contain the query content. However, when curling the same request from the terminal, the response was correct and contained the query. After some investigation, it was found to be due to a multi-statement issue, which also had recently been discussed in AsterixDB's mailing list. An example of the multi-statement is shown in Listing 4.2. When creating a single statement instead, as in Listing 4.3, the issue was resolved, and the user query could be accessed from within the Java UDF.

```
statement=use relevance; select value q.query from UserQueries q WHERE q.id=1;
```

Listing 4.2: A multi-statement request.

```
statement=select value q.query from relevance.UserQueries q WHERE q.id=1;
```

Listing 4.3: A single-statement request.

Ranking Clusters of Tweets

After different approaches for filtering items had been evaluated, the second phase continued with exploring RQ2 and RQ3. In order to only present the most relevant information to a user in real-time, it is necessary to continuously rank the arriving elements. As this is considered the main task of this project, the goal of this phase was to to find an efficient solution to ranking items from the data stream. The following paragraphs will describe the different alternatives which were considered during the development of the ranking component.

With the abovementioned goal in mind, it is necessary to define the notion of relevance. In this study, relevance is defined to be query-dependent: the textual semantic similarity between tweets and the user query. The relevance score should be found by using the Word2Vec model, and calculating the similarity between the word embedded representations of threads' centroids and the user query. Additionally, fresher items should be considered more relevant. However, it must be stated that the scoring function performance, in terms of finding the most relevant items, is not the primary focus of this project. The primary focus is to investigate how the ranking can be performed *efficiently* in a unified big data management system.

With the notion of relevance being defined, the criteria for the ranking component behaviour could be set:

- A list of the top-k ranked items must be preserved at every time instant. If a new element arrives, the top-k elements must be updated if necessary, potentially changing the records stored in AsterixDB if these stored records are not found relevant anymore.
- Updating the top-k elements should not be too time consuming, nor be too computational heavy, as it must be done in-memory and in real-time.
- The scoring function should be query-dependent, by considering the content of a user-defined query.
- To limit the amount of items to rank, the scoring function should be applied to elements which have passed the filtering function.
- Item freshness should be considered.

Alternative Approaches to Ranking Perspective With the requirements for the ranking component being set, it was explored how to improve the baseline ranking function implemented in the initial phase – which only calculated the TFIDF score and stored all arriving data without being able to update it. Now, which perspective to use when ranking had to be considered as well. The need for choosing between a local or global ranking perspective became apparent. Local ranking would consider the generated clusters, whilst a global ranking approach would on the other hand ignore clusters, and instead consider all tweets within the sliding window w. Four alternative ways to implement the ranking component were examined, where all approaches assumes that filtering has been performed beforehand:

1. *Alternative - Top ranked clusters (local):* All tweets in a cluster are assumed to represent candidate tweets relevant to the user. The centroids of the threads in memory are ranked with respect to the user query, finding the top-k most relevant centroids. Then, the topmost thread is assumed to hold the most relevant tweets for the user query. If there exists many clusters at a given point in time, the complexity of this alternative is higher than in the case of few clusters. This is because each arriving tweet would have to be evaluated against the entire number T of threads currently in memory during the allocation phase. However, the number of maintained threads in a window will never increase the number of tweets in a window, which means that the complexity of this approach will not become greater than that of *Alternative 3*: In worst case, every tweet in window W will be assigned to different threads, making $W = T$. By removing threads which does not contain any tweets currently in the sliding window, the total number of threads is restricted. Moreover, let t_1 be the most recently arrived tweet, and that it was allocated to thread T_1, updating its centroid, C_1. It is only necessary to update the top-k items if C_1's score is higher than the lowest score in the top-k list. Lastly, only tweets which are allocated to the top ranked thread during its evaluation phase can be persisted in AsterixDB.
2. *Alternative - Top-k ranked tweets within top ranked cluster (local):* This approach assumes that tweets within the topmost ranked thread are *not* equally relevant for the user query. As the threshold for allocating tweets to a thread must be manually

set, it is reasonable to believe that not *all* tweets within a thread is just as important. Therefore, after the initial ranking finds the thread most relevant for the user query, the tweets within it should be ranked by the same similarity measure. In this approach, only the top-k tweets in the topmost ranked thread can be inserted into storage in AsterixDB. However, this alternative would introduce an additional ranking step compared to *Alternative 1*, which would slow down the performance. Recollect that the ranking must be performed in real-time, and therefore the execution time per tweet should not be too high. Whenever the topmost ranked thread is switched out, a new re-ranking of all tweets within the new top ranked thread must be performed.

3. *Alternative - Top-k of all tweets in window (global):* In this approach, the effect of clustering is disregarded and all tweets in the sliding window are individually evaluated against the user query. However, the top-k items should be maintained in this approach as well: After the scoring function has been applied to a new tweet, it must be checked whether the produced score is higher than the lowest score in the current top-k. As this approach looses the aspect of grouping semantic similar tweets together, it is reasonable to believe that the resulting list of the top-k items will be updated more frequent than when adopting a local, clustering perspective. To compare this approach to Alternative 1, recall t_1 which was allocated to T_1 following the approach in the first alternative. Now, also consider t_2 which is the most recently arrived tweet in the context of this alternative. At arrival, t_2 is ranked highest amongst all tweets seen so far, and it is rewarded at the topmost position and is therefore sent to storage. However, the first n tweets arriving after it, t_3, t_4, ..., t_n, are all scored higher than t_2, and they are all discussing the same exact topic. This would result in frequent updates to the data structure maintaining the top-k items, and also frequent deletions of stored records which have become outdated. In worst case, this would happen for every arriving tweet. If, however, the approach in Alternative 1 was adopted instead, t_2, t_3, t_4, ..., t_n would end up in the same topmost ranked thread T_2. In that alternative, the top-k data structure would only be updated if the scores of centroids in memory changes so much that the top-k positions must be altered. This is however not as likely, as a threshold is manually set to control how similar a tweet must be a centroid before it is allocated to it. This comparison illustrate that Alternative 3 may lead to more maintenance of both the top-k data structure and the tweets in storage.

4. *Alternative - Top-k tweets within top-k threads (local):* This approach assumes that all the created threads can hold relevant tweets for the user query. Therefore, all threads must be ranked with regards to the user query, similar to Alternative 1 and 2, but all tweets within these top-k threads must be ranked as well. Compared to Alternative 2, which only ranks tweets within the topmost ranked cluster, this approach would consider all tweets from all top-k ranked threads. Following, the highest ranked tweets from all top-k ranked threads can be inserted into storage. The additional step which consider tweets from *all* top-k threads introduced in this alternative yields higher complexity compared to Alternative 1 and Alternative 2. However, compared to Alternative 2, this approach would not need to re-rank tweets

before a thread is completely removed from the top-k list of threads, as opposed to each time the topmost thread is switched out.

Alternative Representations for Storage After evaluating the different perspectives to take on when ranking threads or tweest, how to represent the top-k items in storage was investigated. As the top-k items ideally should be queryable in order to make them accessible for a user, and as it is advantageous to store only relevant items to save storage space, an approach for how to define the top-k items' *DataType* had to be chosen. Two alternatives were considered:

1. *Alternative - Creating Specific Datatypes For Ranking:* Specific *DataTypes* are created for representing a single item within the top-k list, and also for the entire ranking list: `RankingType` and `RankingResultType`. The former contains the top-k item's score, either the specific tweet or a list of all tweets it contains (depending on which alternative is chosen in ranking perspective), and posting time. The latter contains k fields, one per top-k item. To hold records of the `RankingResultType`, a *Dataset* `RankingResult` is created, as shown in Listing 4.4. To insert elements of this specific *DataType* into storage, the UDF connected to the dataset must output records of `RankingResultType`. If the UDF which scores and ranks items is going to output records of this type, the UDF is restricted to be an *applied function*. It can not be a *query predicate*, as a query predicate would output a boolean value, and then pass on the entire record which it got as input if the boolean value is `true`. An applied function would on the other hand be able to output a *DataType* which have been given additional fields.

2. *Alternative - Storing Tweets Using Tweet Datatype:* The data to persist in AsterixDB are of the same *DataType* as the UDF's input records, a *Tweet*. If decided that the UDF which scores and ranks is implemented as a query predicate (meaning that the filter happens within the same UDF which ranks), this alternative is the only option. If all processing functionality lies within a query predicate UDF, persisted records will have the same format as passed to the *DataFeed*. Therefore, the score assigned to each item can not be added as a field to the output record, and thus not be stored in AsterixDB either. This means that a user will not be able to see the calculated scores of the persisted items. However, only tweets which are found relevant are inserted into storage as the query predicate makes sure that items not matching the predicate is filtered out.

```
CREATE TYPE RankingType AS CLOSED {
  id: int32,
  score: double,
  tweet: Tweet, // Either a single Tweet, or a list of all tweets in Thread
  time: datetime
};
CREATE TYPE RankingResultType AS OPEN {
  id: int32,
  first: RankingType,
  second: RankingType,
  third: RankingType,
  fourth: RankingType,
  fifth: RankingType
};
CREATE DATASET RankingResult(RankingResultType) PRIMARY KEY id;
```

Listing 4.4: Alternative 1 for storage representation which stores all top-k items within a single record in a dataset.

```
CREATE TYPE Tweet AS OPEN { id: int64 };
CREATE DATASET RelevantDataset(Tweet) PRIMARY KEY id;
```

Listing 4.5: Alternative 2 for storage representation which maintains a record per tweet found relevant.

Alternative Approaches to Implementing Scoring Function After looking into the different ways to represent data in AsterixDB, an investigation of how to implement the ranking function could start. In AsterixDB, there exists three possible ways one could implement a ranking function which operates on a data stream: It could either be implemented as a Java UDF, as a SQL ++ UDF, or it could utilize a combination of both. Referring to the goal of this study, the ranking must be performed in an efficient manner and it must – as the filtering component – also consider the user-defined query when ranking. To rank items with respect to a query, and to continuously update the ranking list, the ranking function needs to know both the the user query's content and the state of the ranking list at all times. The three different approaches considered are listed below:

1. *Alternative - Ranking implemented in a SQL ++ UDF:* The first alternative ought to maintain all ranking functionality within a single SQL++ UDF. As discussed by Wang and Carey [56], these UDFs are suitable when processing incoming records requires the result of a query over data stored in AsterixDB. This could seem to be the case as the scoring function needs information about the stored user query, and the current data in a ranking list which may – depending on the chosen alternatives in the previous paragraphs – be stored in a *Dataset*. However, a SQL++ UDF would be unable to utilize the Word2Vec model to create embedded representations used for calculating semantic similarity. This alternative is therefore not applicable when considering the relevance definition used in this study.
2. *Alternative - Ranking implemented in a Java UDF:* This approach implements all functionality related to scoring and ranking within a Java UDF. One important factor is that a Java UDF makes it possible to use the Word2Vec model for measuring similarity, as opposed to Alternative 1. Furthermore, Wang and Carey [56] discussed that Java UDFs are best suited to access existing data by loading resource files during its initialization phase. Storing the user query in a resource file is not optimal, as this would restrict the system behaviour. Additionally, if the UDF needs information about the top-k items and this information is stored in a dataset, it is not feasible to write this data to file. Te limitation of writing to file can be avoided, because it is possible to retrieve information stored in *Datasets* by sending a request to the AsterixDB's API. Furthermore, as only data considered amongst the top-k should be persisted, these records must be updated from within the Java UDF. This could also be achieved by utilizing the AsterixDB API, as there does not exist any other functionality for removing older records during feed ingestion. The Java API could

then further be employed to maintain the top-k items in-memory and to store the user query as a class field, restricting the number of API requests. If decided to use AsterixDB's API for sending requests, it must be ensured that these are not sent that often, as this will slow down the pipeline. If requesting the user query during the initialization phase using the *current* ingestion framework, the ranking component would not be able to consider changes to the user query. However, if adopting the new framework, the Java UDF would be sensitive to changes in referenced data.

3. *Alternative - Ranking implemented using both types of UDFs:* Afterwards, it was looked into connecting two different UDFs to the data stream: A Java UDF which clustered and scored the incoming data and outputted a flag saying if the tweets should update the ranking list, and SQL++ UDF which then would retrieve the output, access the stored ranking list and could update the records of the ranking list. However, the functionality for appending several UDFs to a data stream is deprecated in the newer AsterixDB versions. After a thorough investigation of the different approaches, it was found that Java UDFs could be called from within SQL++ UDFs. If implementing the main ranking UDF as a SQL++ UDF, it could access stored data about the current ranking list and the user query, and if necessary pass these as arguments to the Java UDF. The Java UDF within the SQL++ UDF could be responsible for clustering the incoming tweets and assigning them a score based on their relevancy to the user query, and output a record with this score.

Chosen Alternatives As light have been shed on the different approaches to *(1.)* how to implement the filtering function, *(2.)* which perspective to use when ranking, *(3.)* how to represent the top-k items for storage, and *(4.)* how to implement the scoring function, final decisions could be made, taking all perspectives into consideration. This paragraph will therefore describe which alternatives were chosen in *(1.)* through *(4.)*.

The final, chosen approach performs filtering, clustering, scoring and ranking within a query predicate type of Java UDF *(4.)*. Due to this, the system can also perform filtering *(1.)* within the same UDF, before the core processing of the tweets begin. When the query predicate outputs *FALSE*, tweets are not sent to storage, whilst the opposite happens when the query predicate outputs *TRUE*. This enables the system to not insert *all* arriving items into storage, which was the case in the specialization project. As the main processing happens in the same UDF as the filtering, they could share the Word2Vec model, and therefore use it for both performing query expansion and to represent the tweets, centroids and user query in the same space. Keeping the functionality within the same UDF also escapes the issues related to running two UDFs if they both wishes to use the model. Furthermore, as the UDF is a query predicate, it can only output records of the same *DataType* that arrived, and therefore the Tweet type is used for storage *(3.)*. Additionally, as the ranking is performed within a Java UDF *(4.)*, it is possible to create an in-memory data structure to represent the current top-k items, which new items can be evaluated against at arrival. Changes to the top-k items in this data structure could then initiate deletion of old records stored in the database using the AsterixDB API. With the focus of this study being to efficiently rank tweets, it was chosen to adopt the local approach which used the fewest rounds of ranking *(2.)*. When ranking items, the top-k threads are maintained in-memory, and only tweets in the topmost ranked thread are inserted into storage, by setting the output value of the query predicate to `TRUE`. The chosen approach have also been affected by the challenges listed below.

Challenges During Second Phase

When trying to start a feed when using the new, decoupled ingestion framework, some issues became apparent. The query for starting the feed never completed. Through discussion with one of the developers from AsterixDB, two possible reasons behind the problem were presented: It could either be a known bug in the UDF data type inference system which was not included in the decoupled version, or the changed function signature in the decoupled branch. Another version which contained the included bug fix for the type inference system was obtained from the AsterixDB developer on 8 April, seen in the second entry of Table C.1. However, in this version, Maven was not able to build the project. This was after some investigation found to be due to a small bug in the `pom.xml` file, which did not correctly specify which `hyracks-control-nc` version to use.

During the evaluation of the alternatives in (4.), it was explored whether calling a Java UDF from within a SQL++ UDF was a suitable solution. It was then tested how to pass a record holding the current ranking list to a Java UDF. When doing so, some obstacles arose. Passing the record as an argument resulted in an compilation exception because of the record type. After discussion with the developers, it became apparent that there could be a bug in the how external function framework handles record types which are optional: There was a rule which failed when the record type was optional within the optimizer which deals with external functions. It was additionally tried to pass primitive types to the Java UDF instead of an entire record. However, the query were still not able to finish. One of the developers at AsterixDB implemented a fix in a new version which solved this issue and it was obtained on 30 April. It was then possible to call the Java UDF from within the SQL++ UDF.

Furthermore, when testing the new, decoupled framework with the normal configuration for the CC process and NC processes, it was found that both of the NC processes tried to read the Word2Vec model into memory. From discussion with a member of the AsterixDB team, it was found to be a bug related to a runtime parallelism hint that could be set. A new version was obtained on 3 May.

When applying a SQL++ UDF which called a Java UDF to the Twitter Feed, or when applying a Java UDF to the Twitter feed, an exception thrown stating that there was an error when processing tuple 0 in a frame. However, applying the UDF to a socket feed worked, which meant that the issue was related to retrieving data from the Twitter API. The error did not occur when dropping the UDF and only inserting data directly from the API. Thorough error handling was implemented in the Java UDF, checking that the arriving tweets contained the field which the UDF tried to use. This did not work. After an in-depth analysis of the stack trace from when the exception was thrown, it was found that AsterixDB failed to produce tuples of the arriving data. A project reproducing the error, and information about where the error occurred was sent to a member of the development team at AsterixDB. A new version which solved the issue of parsing twitter data when attaching a UDF was retrieved on 24 May.

Furthermore, when testing the new, decoupled framework with the normal configuration for the CC process and NC processes, it was found that both of the NC processes tried to read the Word2Vec model into memory. From dicsussion with a member of the AsterixDB team, it was found to be a bug related to a runtime parallelism hint that could be set. A new version was obtained, the new framework, instances of the UDF are running, memory issues (related to fourth version).

4.3.3 Third Phase

In the third phase, the focus was set to conduct several experiments. Before these could be performed, the test cases for the different experiments had to be configured. This involved implementing a Java class which could be used to log different metrics and write these to file during the experiments. When the experiments were completed, this last phase was concerned with evaluating the findings.

Challenges

When trying to run one of the experiments, the system crashed due to an exception. This was a `ConcurrentModificationException`, which can occur if something that is being iterated on is changed at the same time. After some debugging, it was found that one of the `List` was modified concurrently. Therefore, the `List` was replaced with a `CopyOnWriteArrayList`, which is an implementation of `List` which supports concurrent modifications.

4.3.4 Final Implemented System

The three previous sections described the implementation phases of this project. In the following section, an overview of the final system will be given, by describing how the different components in the system are implemented. First, the necessary preconditions of the system are outlined, followed by a thorough description of the components in sequential order of when the components operate in the pipeline.

Preconditions

AsterixDB is implemented in Java, and the development team of AsterixDB also have several code examples implemented in Java. This served as the main reasons for developing this project with the same programming language. It could have been possible to implement parts of the system using Python. However, this would have required to embed Python in Java using Java Native Interface [4]. Additionally, both Scala and Kotlin run on the Java Virtual Machine and could therefore have been used instead of Java, but choosing one of these languages would have led to additional learning overhead. Another important reason for choosing Java is because it is a battle-tested programming language, and it used by a very large number of people and companies. It is therefore easy to find resources related to problems which occur.

AsterixDB maintains a UDF template, `asterix-udf-template`[2], in a GitHub repository written in Java, which provided a starting point for the *initial system* implemented last year, and therefore also the starting point for this project. The project in this study is from now on referred to as *the project* to distinguish it from others. *The project* is extended with functionality specifically required for the UDF for the use case in this study: filtering, clustering, scoring and ranking. Building *the project* and handling the dependencies are done using Maven[3], and a version greater than 3.3.9 is required. To be able to use AsterixDB – both for a local running instance as well as a dependency required in *the project* – one must clone the AsterixDB repository[4]. Additionally, this must be built by following the steps provided in the repository's `README`. This ensures that AsterixDB is installed in the local Maven repository, and can be used as a local dependency in *the project*. The newest version available on GitHub at the time of writing is version `0.9.5-SNAPSHOT`. Additionally, when testing versions of AsterixDB which used the new, decoupled ingestion framework, these have been obtained through mail, and are not available in AsterixDB's master branch.

In order for AsterixDB to be able to parse the incoming tweets, it is necessary to use the Twitter4j library, namely `twitter4j-core` and `twitter4j-stream`. One must download the JARs – files which holds the Java classes, resources and metadata in a single distribution – of these dependencies and add these to the `repo` folder within the server before Twitter data can be ingested. The newest version of these dependencies at the time of writing, 4.0.7, are not compatible with the Twitter adapter used in AsterixDB. However, version 4.0.3 was tested and found to be working.

For making the local instance of the system able to utilize AsterixDB's HTTP API for sending requests, two more dependencies must be added to the Maven project: The *Http-Client* artifact from `org.apache.httpcomponents`, and the *json* artifact from `org.json` for handling the API response.

Retrieve Streaming Data from Twitter in AsterixDB

AsterixDB retrieves data from Twitter by using the functionality called *Feed Adaptors*. A brief description of this functionality was given in Section 2.5.1. These adaptors have two different modes to operate in: pull or push. The former implementation requires the adapter to send separate requests each time to receive data, whilst the latter only requires *one* initial request to let data be pushed to the adapter [6].

To have Twitter data continuous arrive into AsterixDB, it is necessary to set up a connection with the Twitter API by using AsterixDB's built-in, push-based Twitter adapter.

[2]https://github.com/idleft/asterix-udf-template
[3]https://maven.apache.org/index.html
[4]https://github.com/apache/asterixdb

Retrieving data from the Twitter API requires one to submit an application to Twitter describing what the APIs ought to be used to. If access is granted, one will obtain keys and tokens. When configuring the Twitter Feed, one must specify that the `push_twitter` adapter should be used, the keys and tokens obtained from Twitter, the *Datatype* of the incoming record, followed by the format of the Twitter data. Lastly, the decoupled field and the batch size must be set if the new ingestion framework is being used. Listing 4.6 shows how to define the Twitter feed using the decoupled framework, and the required *Datatypes* for describing a tweet. When using the specified adapter, AsterixDB ensures that the arriving items in the data stream are parsed to the correct type, so there is no need for manually doing this.

```
CREATE TYPE Tweet AS OPEN { id: int64 };

CREATE FEED TwitterFeed WITH {
    "adapter-name": "push_twitter",
    "type-name": "Tweet",
    "format": "twitter-status",
    "consumer.key": "******",
    "consumer.secret": "******",
    "access.token": "******",
    "access.token.secret": "******",
    "batch-size":"X", // Can be dropped if using old framework
    "decoupled": true // Can be dropped if using old framework
};
```

Listing 4.6: Creating a Datatype for representing Twitter data and creating a Feed which can retrieve tweets from Twitter API.

Creating a UDF For Processing Tweets

Before tweets in the data stream can be processed, the UDF must be defined. To define which *Datatype* the UDF takes as input and gives as output, the `library_descriptor.xml` file must be configured. As seen in line 7-8 in Listing 4.7, it takes a `ASTRING` as an argument, which is the tweet text, and outputs a `ABOOLEAN` indicating whether the tweet is relevant for the user query or not. The system will then discard all the tweets which returns `FALSE` from the UDF, which has been named `detectRelevance` as seen in Line 5. Additionally, Line 9 specifies that the Word2Vec model is going to be sent as a parameter to the UDF. The filtering, clustering and ranking functionality of the UDF will be described in detail in the following paragraphs, by referring to the UDF implementation in Listing A.1.

```
1  <externalLibrary xmlns="library">
2    <language>JAVA</language>
3    <libraryFunctions>
4      <libraryFunction>
5        <name>detectRelevance</name>
6        <function_type>SCALAR</function_type>
7        <argument_type>ASTRING</argument_type>
8        <return_type>ABOOLEAN</return_type>
9        <definition>org.apache.asterix.external.library.RelevanceDetecterFactory</
      definition>
10       <parameters>GoogleNews-vectors-negative300.bin.gz</parameters>
11     </libraryFunction>
12   </libraryFunctions>
```

```
</externalLibrary>
```

Listing 4.7: Configuring library functions in the the library descriptor file.

Filtering Arriving Tweets Based on a Continuous User-Defined Query

As of the previously mentioned scope of this project, it was chosen to only handle *one* user query at a time, as a proof-of-concept. To maintain the user-defined query to use when filtering – and ranking later on – a *Datatype* and a *Dataset* are created as shown in Listing 4.8. The SQL++ statements in Lines 1-12 are ran from the AsterixDB user interface.

```
1  CREATE TYPE UserQueryType AS OPEN {
2    id: int32,
3    time: datetime,
4    query: string,
5    k: int
6  };
7
8  CREATE DATASET UserQueries(UserQueryType) PRIMARY KEY id;
9
10 INSERT INTO UserQueries([
11     { "id": 1, "time": current_datetime(), "query": "Wayne Rooney soccer", "k": 5 }
12 ]);
```

Listing 4.8: Creating a Datatype to represent a user query, a Dataset for holding user query records, and inserting a user query into the Dataset.

To filter tweets which are being pushed to the adapter, the Java UDF defined in the previous section must first be attached to the *DataFeed* as a query predicate, as described in the end of Section4.3.2. Listing 4.9 attaches the query predicate UDF to the *Data Feed*. As Line 2 shows, the field in the arriving item which is going to be evaluated – here the text field – must be set as an argument to the UDF. The type of this field must match that of the argument type specified in Listing 4.7.

```
1  CONNECT FEED TwitterFeed TO DATASET RelevantDataset
2  WHERE detectRelevance(TwitterFeed.text) = TRUE;
```

Listing 4.9: Attaching a Java UDF named detectRelevence as a query predicate to the DataFeed.

Looking at the query predicate UDF in Listing A.1, it contains two important methods: `initialize` and `evaluate`. When the *DataFeed* is started, the `initialize` runs first. The part of this method which is relevant for filtering is found in line 9. It calls the `createQuery()` method, defined in line 14-21. This method sends a request to the API exposed by AsterixDB, asking for the user query. The query is stored as a class variable, making it accessible as long as the *DataFeed* is running. After the initialization phase is finished, the `evaluate` method processes every arriving tweet, as illustrated in Figure 4.3. In this method, the essential parts for the filtering is found in Lines 37-40. These lines checks whether the tweet's content matches that of the user query, by finding the number of common terms. As the `evaluate` method runs on every incoming tweet, the system filters by a continuous user-defined query.

Line 15 calls the `API.getQuery()` method which requests the user query from AsterixDB's API. In order to execute the request, a `HTTP Post` and a `HTTP Client` must be instantiated, as shown in Lines 4-5 of the API class in Listing A.2.

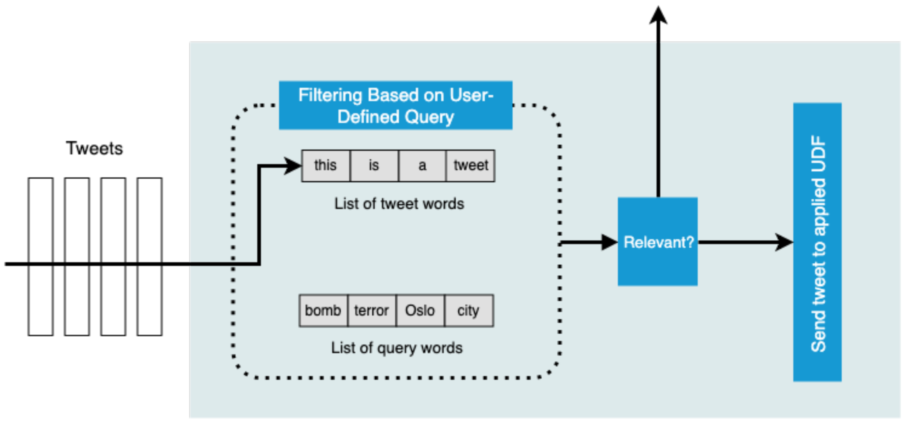

Figure 4.3: A data stream of tweets, where each tweet is sent to a filtering component which checks whether the tweet and the user query share any common keywords. Tweets which passes the filter are being passed on inside the UDF.

Clustering Relevant Tweets

In order to achieve the clustering improvement suggested in Section 4.3.1, the clustering algorithm in the *initial system* implemented in the specialization project was modified by extracting the thread logic to its own class, see Listing A.3. Thus the clustering algorithm holds a data structure `Map<Integer, Thread>` to maintain all current threads, and each `Thread` holds a `INDArray` representing its centroid. The initial work on the clustering algorithm followed the work presented in [47] which is based on [45]. The semantic similarity is now found between the centroid of a Thread and a tweet instead, by averaging the word embeddings of the current tweet and all tweets within a Thread.

Some configuration is however necessary before the clustering can happen, as it depends on the Word2Vec model. By still referring to the UDF in Listing A.1, a instance of the Clustering class is set up in Line 10 during the initialization phase. In Line 22-30, the Word2Vec model – which is passed to the UDF as a parameter – is being read into memory. When the `evaluate` method in Listing A.1 afterwards runs on the incoming tweets, line 42 calls the `clusterTweet` method with the tweet's content.

Tweets which then passes the filter are maintained in a count-based sliding window of size w, implemented as a first-in-first-out (FIFO) queue. At the arrival of a new tweet, the oldest tweet is efficiently removed from w, by polling from the top of the queue. Removing a tweet from the queue triggers the thread centroid which the tweet used to reside in to be updated. Whenever a tweet have been removed or added to the window w, the centroid of the Thread holding the tweet is updated by calling the `setCentroid` method in Line 16 in Listing 4.10. Line 14 completely removes a Thread held in memory if the Thread became empty when a tweet was polled from the queue. When a Thread's centroid is created, all words from all tweets within it is used. Line 8-10 in Listing A.4 extracts these words and uses them to create the mean word embedding of type `INDArray` representing the centroid. Given the changes made to the data structure, allocating a tweet to a Thread is now accomplished by finding the centroid most similar to the tweet's mean word embedding.

```
 1  private void removeFirstTweetFromWindow() {
 2      String tweetToRemove = tweets.poll();
 3      // Find thread to remove tweet from.
 4      Map.Entry<Integer, Thread> thread = threads.entrySet().stream()
 5          .filter(t -> t.getValue().getTweets().contains(tweetToRemove))
 6          .findFirst()
 7          .orElseThrow(() -> new IllegalStateException("Could not find tweet"));
 8
 9      boolean removed = thread.getValue().getTweets().remove(tweetToRemove);
10      if (!removed) {
11          throw new IllegalStateException("Could not remove tweet from thread");
12      }
13      if (thread.getValue().getTweets().isEmpty()) {
14          threads.remove(thread.getKey());
15      } else {
16          setCentroid(thread.getKey(), model);
17      }
18  }
```

Listing 4.10: Removing tweet from window and updating its old thread.

When a new tweet is to be clustered, the similarity between it and all the current centroids in memory are first calculated. Lines 2-11 in Listing 4.11 find the Thread in memory with the highest similarity score to the tweet's word embedded representation. First, all Threads are mapped to a `CosineSimilarityThreadPair` in Line 3, which represent the similarity between each of the mapped Threads and the new tweet. It's constructor requires the tweet to be represented as a word embedding as well before calculating the similarity, so Line 4 calls a method which creates this before calculating the similarity. To find the cosine similarity between a centroid and a tweet, both is represented as vectors. By using the external library deeplearning4j[5] and the pre-trained, loaded model which is built on Google News, Word2Vec representations are created for the centroids and the tweets.

```
 1  private Optional<CosineSimilarityThreadPair> nearestCentroid(String newTweet) {
 2      return threads.values().stream()
 3          .map(thread -> new CosineSimilarityThreadPair(
 4              cosineSimilarityThread(
 5                  model,
 6                  newTweet,
 7                  thread.getCentroid()
 8              ),
 9              thread
10          ))
11          .max(Comparator.comparingDouble(CosineSimilarityThreadPair::getSimilarity));
12  }
```

Listing 4.11: Method for finding a tweet's nearest centroid in clustering algorithm.

[5]https://deeplearning4j.org

Ranking Clusters of Tweets

Recall that the ranking approach taken in this study only considers *one* maintained user query, as specified in Section 1.4. By still referring to the UDF in Listing A.1, Lines 43-44 scores and updates the ranking list, and sets the flag `shouldUpdateStorage` to `TRUE` if the arriving tweet is going to be stored in AsterixDB. The user query is, as described earlier, already obtained as of the initialization phase, and it is passed as an argument to the scoring function together with the tweet in Line 43. As the query is retrieved during initialization, there is no need to request it for every new tweet which arrive.

The scoring function from the `Score` class, shown in Listing 4.12, is called in Line 43. It calculates the cosine similarity between the centroid of the Thread found in Line 41 – which the tweet was assigned to – and the user query. As the query and Thread's centroid reside in the same space, the method `queryThreadSim` can calculate the distance between the two vectors. The closer the score is to 1, the more semantically similar is the the content of the Thread and the user query. After the score is calculated, it is stored in the `score` variable.

```java
public class Score {
    public static double queryThreadSim(Query query, Clustering cl, Integer id){
        Thread thread = cl.getThread(id);
        return Transforms.cosineSim(
            createQueryVector(query.getText(), cl),
            thread.getCentroid()
        );
    }

    private static INDArray createQueryVector(String query, Clustering cl){
        Collection<String> queryLabels = Splitter.on(' ').splitToList(query);
        return cl.getModel().getWordVectorsMean(queryLabels);
    }
}
```

Listing 4.12: The score class, which calculates the similarity between a tweet and a thread centroid.

To represent the top-k results for the user query, a `TreeMap` is adopted and stored as a class variable in the `Ranking` class. Line 44 sends the calculated score and the Thread to methods in `Ranking`, shown in Listing 4.13. The top-k Threads are maintained in this data structure, with their associated score used as key. When retrieving the highest and lowest scores, this can be performed with $O(log\ n)$ complexity.

The `checkForUpdate` method in Listing 4.13 essentially initializes a variable with the current top-ranked Thread in Lines 8-10, updates the current ranking list if necessary in Line 12, and checks whether the top-ranked Thread has changed in Line 15. Only tweets which resides within what is currently evaluated to be the top-ranked thread are stored. This is the reason for sending a request to the AsterixDB API in Line 16, deleting old records when the top-ranked thread changes.

```java
1  public static boolean update(Thread thread, double score) {
2      boolean isUpdated = score >= THRESHOLD && checkForUpdate(thread, score);
3      return isUpdated;
4  }
5
6  private static boolean checkForUpdate(Thread thread, double score) {
7      removeEmptyThreads();
8      Thread currentTop = rankingList.isEmpty()
9          ? null
10         : rankingList.lastEntry().getValue();
11
12     boolean isUpdated = updateRanking(thread, score);
13     if (!isUpdated) { return false; }
14     Map.Entry<Double, Thread> newCurrentTop = rankingList.lastEntry();
15     if (currentTop != null && currentTop.getId() != newCurrentTop.getValue().getId())
         {
16         API.clearDataset(); //Remove old records stored in AsterixDB
17     }
18     return newCurrentTop.getValue().getId() == thread.getId();
19 }
```

Listing 4.13: Methods for checking if the ranking list is updated, and if the current top-ranked thread has changed.

In Line 12, the `updateRanking` method is called with the `Thread` and its score as arguments, and it returns `TRUE` if the ranking list is updated. This method is shown in Listing 4.14, and it is responsible for the main maintenance of the ranking data structure. First, Lines 2-5 adds the `Thread` and its score to the top-k list the list is empty. If, however, the top-k list is found to be full in Line 7, the tweet is only sent to storage if its `Thread`'s score is higher than the lowest score in the `TreeMap`. Line 12 checks whether the `Thread` already exists in the `TreeMap`, and updates its value if it has changed since the last evaluation. Then, Lines 17-18 replaces the lowest entry in the `TreeMap` with the `Thread` and its score. The last case, if the number of items in the ranking list is less than k, the `Thread` is either updated if it exists, or the `Thread` and its score is added to the top-k items.

During each evaluation of a tweet, it is only necessary to check if the score of the tweet's `Thread` is larger than the lowest score stored in the `TreeMap`. As the scope of this project is limited to only maintain one user query, this approach was found sufficient.

```
1  private static boolean updateRanking(Thread thread, double score) {
2      if (rankingList.isEmpty()) {
3          rankingList.put(score, thread);
4          return true;
5      }
6      if (rankingList.size() == 5) { //k=5
7          if (score < rankingList.firstKey()) {
8              return false;
9          }
10
11         if (rankingList.containsValue(thread)) {
12             replaceEntry(score, thread);
13             return true;
14         }
15
16         rankingList.remove(rankingList.firstKey());
17         rankingList.put(score, thread);
18         return true;
19     }
20     // Size < k
21     if (rankingList.containsValue(thread)) {
22         replaceEntry(score, thread);
23     } else {
24         rankingList.put(score, thread);
25     }
26     return true;
}
```

Listing 4.14: Call method to calculate rank

Overview

The filtering, clustering and ranking described in the previous sections is applied to every arriving tweet since the created UDF, `detectRelevance`, is attached to the *DataFeed*, as shown in Listing 4.15. Figure 4.4 shows the clustering and ranking process applied to tweets after they have been filtered.

```
CONNECT FEED TwitterFeed TO DATASET RelevantDataset
WHERE testlib#detectRelevance(TwitterFeed.text) = TRUE;
```

Listing 4.15: Method for maintaining the ranking data structure.

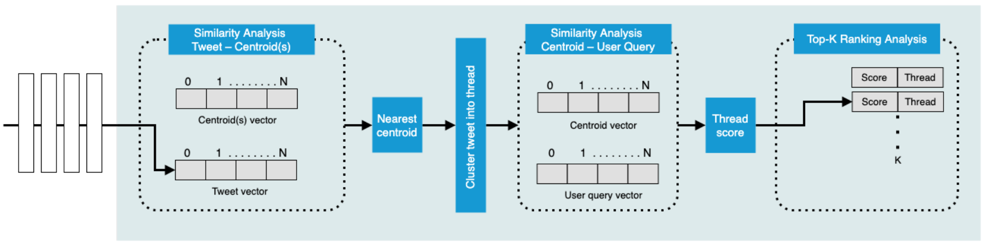

Figure 4.4: Processing of tweets after they have been filtered.

Chapter 5

Experiments and Results

This chapter will describe the preliminary work, setup and results for the different experiments carried out. The overall goal of the experiments will be presented, followed by the dataset used and the applied metrics.

Five main categories of experiments have been performed, focusing on different parts of the research questions. Each of the experiments focus is presented in Table 5.1. Detailed, individual goals for the different experiments will be given in the start of their respective section, together with the preparatory work done before the experiments were carried out. Furthermore, the results will be presented consecutively after each experiment.

Experiment	**RQ1**	**RQ2**	**RQ3**
Scalability			X
Filter Based on a Continuous User-Defined Query	X		
Maintaining the Top-k Ranked Threads	X		X
Total Processing Time of Clustering		X	
Ranking Threads of Tweets	X	X	X

Table 5.1: The focus of the different experiments with respect to research questions.

5.1 Goals with Experiments

Chapter 1 described the goal of this project, which is to develop a solution which ranks tweets from Twitter with respect to a user-defined query in an efficient manner. As the system should detect relevant tweets in real-time from an unbounded data stream, it is crucial that it can manage the *velocity and volume* characteristics of Big Data, discussed in Section 2.1. Therefore, the experiments must be set up so that this can be measured.

As the main goal of this thesis has been to implement ranking in a unified BDMS in an efficient manner, without overloading the user with information and not wasting storage space, the quality of the retrieved items is not the main focus.

5.2 Evaluation Methodology

5.2.1 Dataset

A static dataset of roughly 2.7M tweets – provided in the course TDT4305 – is used as test data for the experiments not ran in real-time. The acquired dataset format was `.tsv`, however, the dataset was transformed to match the ADM format before ingestion. This was done in order to make it possible to stream the data. Even though several fields were included in the dataset, such as information about origin country and the language of the tweet, only the fields `text` and `id`, as exemplified in Listing 5.1, were used in the experiments which did not use the live Twitter data.

```
{ "id": int, "text": string}
```

Listing 5.1: Example of the data format used in the experiments which are not performed in real-time.

In order to control the rate of tweets arriving during the experiments, a Data Generator[1] was used to mimic a data stream. It was implemented as a part of Thor Martin Abrahamsen's Master Thesis [2], and it has been used with his permission. Tweets, which have been formatted to be ADM compliant, are streamed to port 10002. The *DataFeed* configuration is set up to retrieve data from the port. As the initialization phase of the UDF takes almost 2 minutes due to reading the model into memory, the DataGenerator is set up to send data right after this phase is finished.

For the real-time experiment, the *DataFeed* was configured to use the Twitter adapter instead of the socket adapter, to be able retrieve tweets from the Twitter API.

5.2.2 Evaluation Metrics

As a tool for measuring different metrics, a Java class called `TweetProcessingLogger` was implemented. The class contained two public methods, `start()` and `stop()`. These methods were called from the Java UDF, reporting information from the specified lines of code in the UDF. The metrics to be evaluated are:

Number of tweets processed per second To evaluate the performance of the system, and if it the system is suited to handle the characteristics of streaming data, the experiments should measure the number of tweets processed per second.

Execution time per tweet The scalability of the system should be evaluated by measuring the execution time of processing an an increasing amount of tweets.

[1]Data generator: https://github.com/thormartin91/twitter-stream

Execution time of filtering function To evaluate the cost of issuing a continuous user-defined query which runs on every arriving item, the execution time of filtering should be measured.

Execution time of clustering As clustering has been used as a tool for reducing the search space and not having to update the ranking list that often, its execution time should be measured in an experiment to evaluate whether the benefits of clustering justifies the time spent on the process.

Execution time of scoring and ranking To investigate whether the cost of maintaining the data structure holding the top-k threads is high, its execution time should be measured.

Ranking requirements A set of requirements for the ranking component was defined in Section 4.3.2, and these should be used to evaluate the ranking function.

5.3 Experiments

5.3.1 Test Setup

For the experiments not using real-time Twitter data, the newest version available of AsterixDB on GitHub as of May 21st is used. The first experiment will evaluate the scalability of the implemented solution. All experiments were started with a warm-up phase of 6 minutes, where the `TweetProcessingLogger` did not log anything. Several preliminary steps are required before the experiments can be performed, and the order followed when executing the experiments are described in Step 1. through Step 9.

Step 1	Run `mvn clean install` within the Maven project to build the project and generate a package, which is subsequently located in the `/target` folder.
Step 2	Ensure the local instance of AsterixDB is running by starting a sample cluster using the supplied script, `start-sample-cluster.sh` found within the AsterixDB server.
Step 3	While the local instance is running, issue the query in5.2 from the AsterixDB user interface, in order to create the *Dataverse*, *Datatypes*, *Dataset* and the *DataFeed*.
Step 4	Shut down the local instance of AsterixDB.
Step 5	The folders for to hold the files to be deployed must be created: `/udfs/relevance/testlib` within the `/lib` folder of the AsterixDB server.
Step 6	Unzip the zip file generated in Step 1 and move it to the folder created in Step 5.
Step 7	Place the Word2Vec model within the `/opt/local/` folder of the AsterixDB server, which lets it be loaded correctly as a parameter to the UDF.
Step 8	Start the local instance of AsterixDB.
Step 9	Run SQL++ statements to connect and start the *DataFeed*, as described in Listing 5.3 for all experiments using test data. The experiment using live data uses the SQL++ statements shown in 5.4.

AsterixDB provides a user interface which is reachable at `http://localhost:19006/`. This interface was used to run the abovementioned DDLs. The sliding window size has to be set to $W = 2000$ in the Java UDF, as this is the same window size used in the clustering experiments in [47]. The similarity threshold for assigning tweets to threads was set to 0.7, as tests performed during the specialization project showed that a too low threshold introduced more spam. The local instance of AsterixDB was hosted on a Dell OptiPlex 7040 computer with hardware specifications described in Table 5.2. This PC was also used for performing the experiments.

Memory	16 GB DDR4 2133 MHz RAM
Processor	Intel® Core^{TM} i7-6700 CPU @ 3.40GHz x 8
Graphics	Intel® HD Graphics 530 (Skylake GT2)

Table 5.2: Hardware details of the local instance of AsterixDB.

```
CREATE DATAVERSE relevance;
USE relevance;
CREATE TYPE Tweet AS OPEN { id: int64 };
CREATE TYPE UserQueryType AS CLOSED {
    id: int32,
    time: datetime,
    query: string,
    k: int32
};
CREATE DATASET RelevantDataset(Tweet) PRIMARY KEY id;
CREATE DATASET UserQueries(UserQueryType) PRIMARY KEY id;
INSERT INTO UserQueries([{
    "id": 1,
    "time": current_datetime(),
    "query": "soccer",
    "k": 5
}]);
```

Listing 5.2: SQL++ statements used for creating Datatypes and Datasets used in all experiments.

```
CREATE FEED TestSocketFeed WITH {
"adapter-name": "socket_adapter", "sockets": "127.0.0.1:10002",
"address-type": "IP", "type-name": "Tweet", "format": "adm"
};

CONNECT FEED TestSocketFeed TO DATASET RelevantDataset
WHERE detectRelevance(TestSocketFeed.text) = TRUE;

START FEED TestSocketFeed;
```

Listing 5.3: SQL++ statements used for all experiments which uses the socket-based adapter.

5.3.2 Experiment 1: Scalability

As introduced in Section 2.1, the term velocity in the context of characterizing Big Data refers to the speed at which data is created, ingested and processed. On an average day, about 6000 tweets are posted every second[2]. As the goal of this thesis is to investigate how to rank *tweets* to a user query from a *data stream*, the motivation behind this experiment is to determine whether the system can handle a similar, or better, arrival-rate than the rate of produced tweets from Twitter. Additionally, as .. context of streaming data, the timing constraint of real-time analysis should be met, and this experiment will therefore evaluate this. The experiment is divided into two parts: Measuring how the system handles different arrival-rates, and measuring the time spent processing an increasing number of tweets.

Preparatory Work

The experimental setup in both parts of the experiment followed that of Listing 5.2 and Listing 5.3.

In order to achieve the most consistent environment for performing the first part of the experiment, the `DataGenerator` was used instead of retrieving real-time data from the Twitter API, as this allowed for manually set, fixed arrival-rates. The generator was configured to send 1000, 2000, 3000, 5000 and 10 000 tweets per second. To get a more accurate measurement of what the system is able to handle, three runs of testing each arrival-rate

[2]Source, retrieved 13.06.19: https://www.internetlivestats.com/twitter-statistics/

was performed. The results was then used to investigate the average processing time per tweet with an increasing arrival-rate. The `TweetProcessingTimer` was configured to log the total processing time of each tweet, the average arrival rate within each second, and the average processing time per tweet within the same second. The warmup time was set to 6 minutes, and each run lasted for 10 minutes.

The goal of the second part of the experiment was to measure the total time spent processing a given number of tweets. To achieve this, two new runs of the experiment was performed. Here, the `DataGenerator` was set to generate a fixed rate of 5000 tweets/second. The logs produced by the `TweetProcessingTimer` could then be analyzed to find the total time spent processing an increasing amount of tweets, as it logged the total execution time for each tweet. Again, the warmup time was 6 minutes, but this experiment ran until all data was evaluated.

Figure 5.1 shows the timeline of the communication between different parts in the system. It should be noted that during this experiment there was not performed any deletion of records when a new top-ranked thread was found, as the sole focus of the experiment was the input rate of tweets the system is able to handle in general.

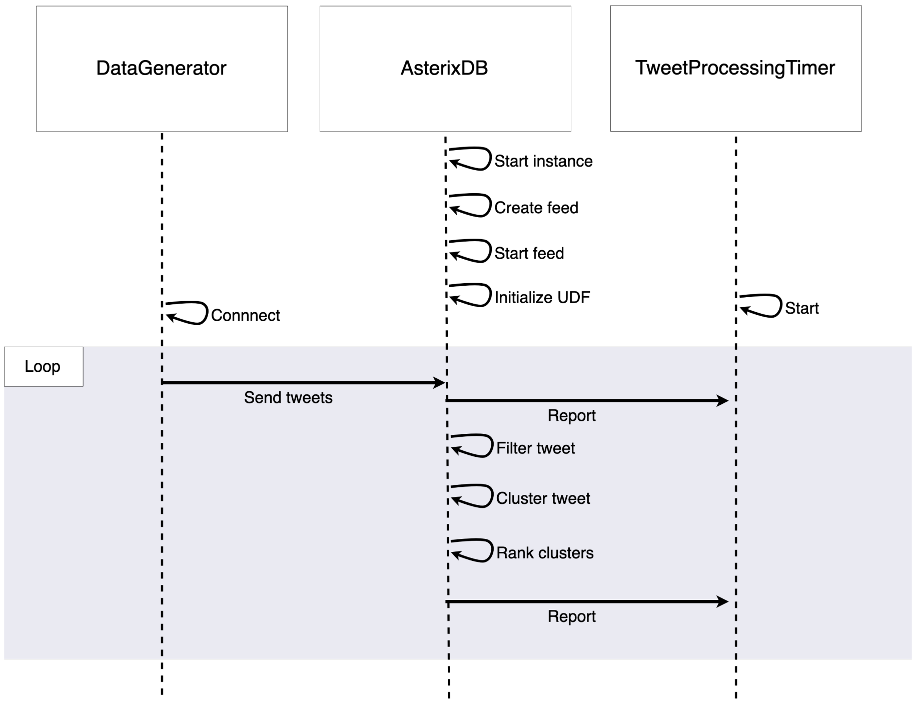

Figure 5.1: Sequential diagram for experiment 1

Results

Figure 5.2 shows the result from the experiment's second part, measuring the total time used by the system to process *1, 10, 100, 1000, 2000, ..., 10 000, 15 000, 20 000, 100 000, 500 000* and *1 million* tweets. Additionally, it shows the regression with a 95% confidence interval and an R^2 value of 1. A graph providing a more detailed view of the lower values on the *x*-axis can be found found in Figure B.1 in Appendix B.

The graphs produced by the fixed arrival-rate runs performed as the first part of the experiment can be found in Figure B.2 through Figure B.18 in Appendix B, and the numbers are presented as tables in Table C.2 through Table C.6. The observations from these runs have been used to produce Figure 5.3. It shows the average execution time per tweet in milliseconds, with an increasing arrival-rate of tweets. By averaging the observations from the three runs, which are summarized in Table 5.3, the graph in Figure 5.4 shows the total average of all runs.

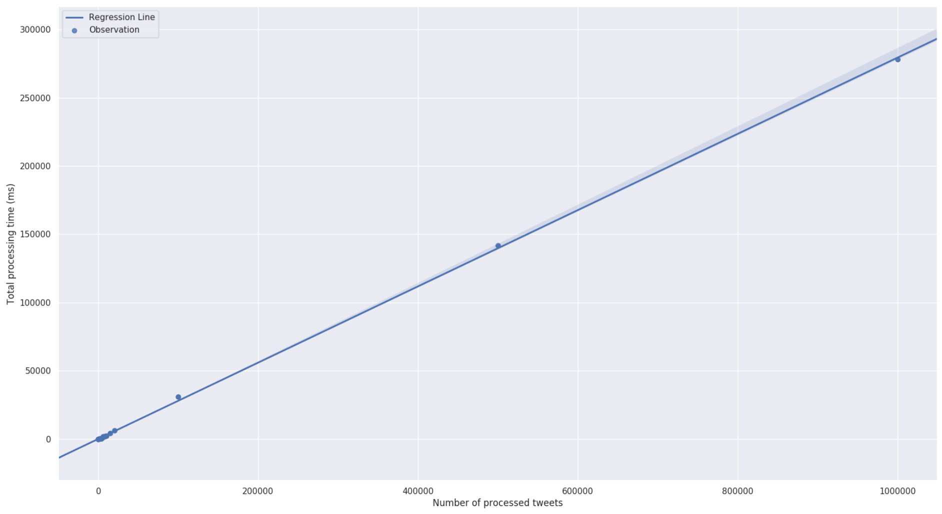

Figure 5.2: The observations from two runs of the experiment which measured the total time spent processing an increasing amount of tweets.

	1000	2000	3000	5000	10 000
Average arrival-rate	562.2	1234.2	2006.5	3449.8	6823.7
Average execution time (ms)	0.0213	0.0175	0.0199	0.0199	0.0206

Table 5.3: Averages produced from three runs of the experiment measuring the arrival rate.

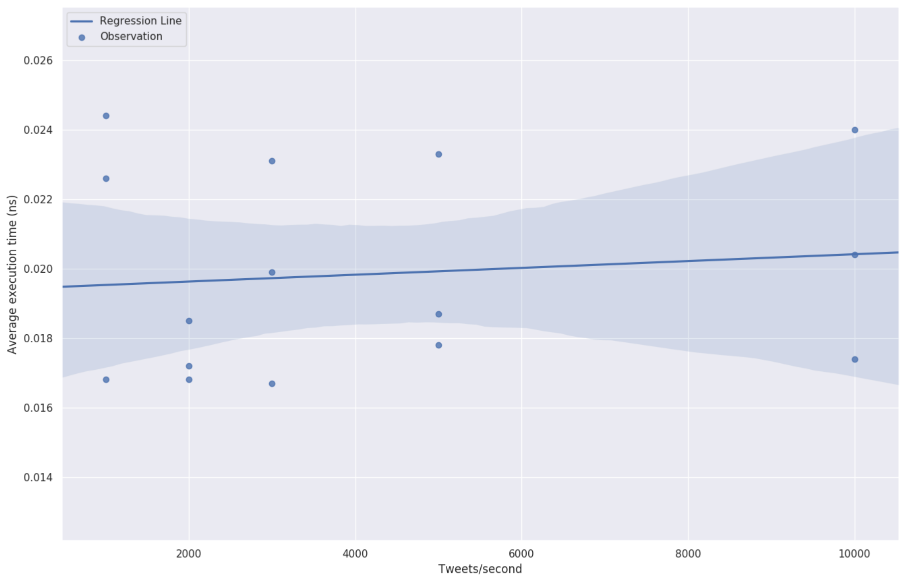

Figure 5.3: Processing time of tweets by an increasing arrival-rate of tweets, produced by performing three runs of the experiment.

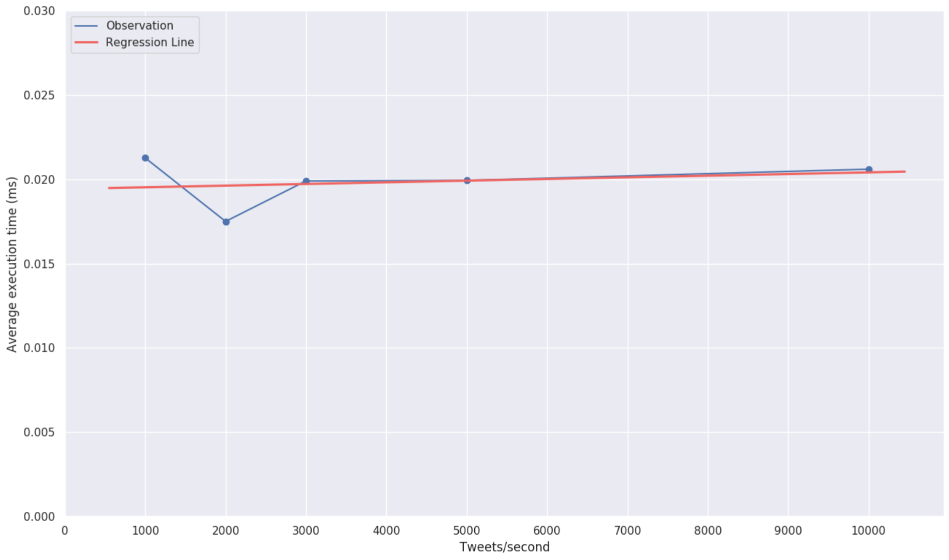

Figure 5.4: Average produced from three runs of measuring processing time of tweets by an increasing arrival-rate of tweets.

5.3.3 Experiment 2: Filter By a Continuous User-Defined Query

As presented in Section 1.1, the issue of information overload is an essential motivation for filtering the data stream content. The filtering function, which is based on a query, will run on every incoming record. Its execution time therefore directly affects the rate at which tweets can be ingested, and thus the throughput of the entire system. As the goal is to process tweets from the data stream at a rate as high as possible, without losing potentially relevant tweets, the execution time of the filtering function should remain low. This experiment was designed to measure the execution time from a tweet entered the UDF until it was either filtered out or found to match the filter predicate. As the system does not restrict the number of keywords in a query, this experiment will also investigate how the system performs with an increasing number of query keywords.

Preparatory Work

The setup described in Listing 5.2 and Listing 5.3 was used in this experiment as well. However, the filtering function within the UDF was in this experiment modified to expand the user query by n number of keywords. Starting with a query containing one keyword, the query was then extended in the next runs, finding the 2, 5, 10, 20, 30, .., 100 most semantic similar words for the initial query keyword, `soccer`, by using the Word2Vec model. The experiment was ran 11 times in total, once for each time the query was extended. The `TweetProcessingTimer` was configured to log the filtering duration of each tweet. Before increasing the number of keywords, the system was stopped, and the UDF was modified to find a higher number of similar keywords to use in the query expansion process.

For this experiment, the filtering component was isolated, and only its behaviour was logged to the `TweetProcessingTimer`. This is because it is desirable to only measure its execution time. Before data was logged, a warm up phase of 6 minutes was completed. The throughput rate was set to 3000, and the window size remained w = 2000. The user query was retrieved during the initialization phase of the UDF.

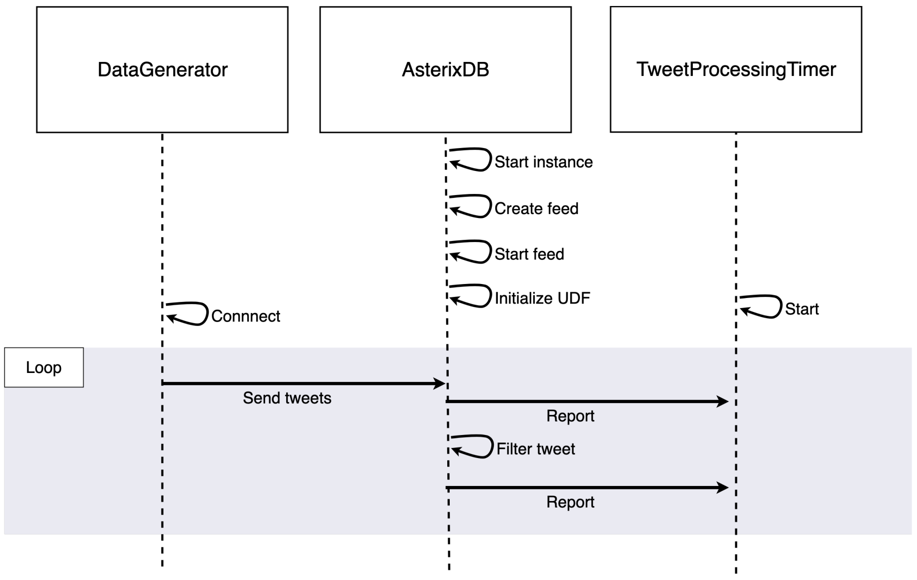

Figure 5.5

Results

Figure 5.6 shows the average filtering time, when a query of *one* word is extended with an increasing number n, of keywords. Each observation is an average of 10 000 tweets. All the raw observations are presented in Figure B.22, and a graph which provides more detail on the lower execution times from this Figure is shown in B.23.

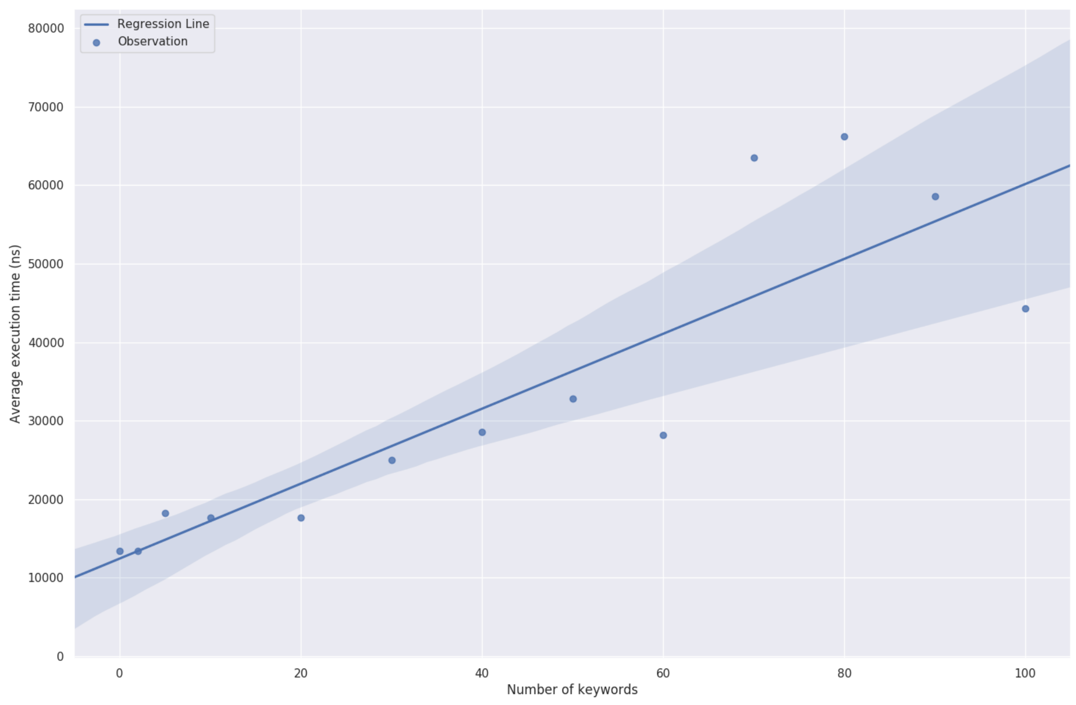

Figure 5.6: Average execution time for the filtering function per number of keywords, with `soccer` as user query and performing query extension.

5.3.4 Experiment 3: Maintaining the Top-k Ranked Threads

As part of the proposed solution presented in Section 4.3, the top-k ranked threads are maintained in-memory and updated continuously. As stated in Section 1.2, the main research question of this thesis is related to making continuous queries efficient when ranking tweets. To answer RQ3, it is necessary to measure the cost of applying ranking to the items in the data stream. Therefore, the motivation behind this experiment is to measure the execution time of scoring threads and updating the elements in top-k.

Preparatory Work

This experiment also followed the setup from Listing 5.2 and Listing 5.3. The `DataGenerator` was set to generate tweets at a fixed rate of 3000 tweets/second. To only measure the execution time of scoring and ranking tweets, the `TweetProcessingTimer` was configured to log the duration of executing line 43-44 in Listing A.1 in Appendix A. Lastly, Figure 5.7 shows the timeline of communication set up for this experiment.

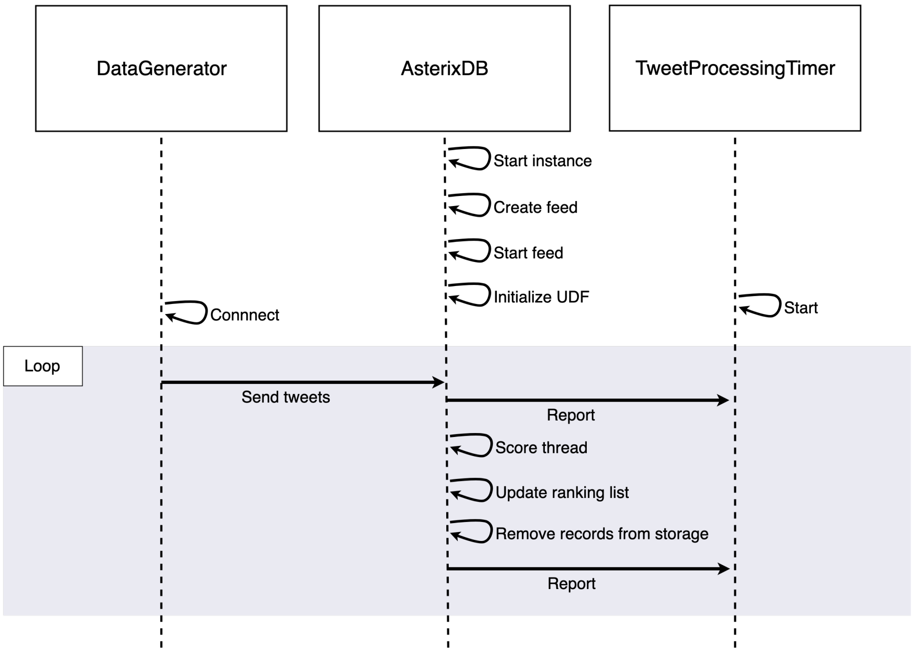

Figure 5.7

Results

Figure 5.8 shows the average execution time for scoring and maintaining the ranking list for all the tweets which were processed during each second of the experiment duration. Table 5.4 presents the summary statistics for the experiment.

Size of data structure holding top-k	280 bytes
Total size of Thread class	1296 bytes
Average execution time	0.1104 ms
Median execution time	0.0596 ms

Table 5.4: Results from running experiment which scored threads and updated the top-k items.

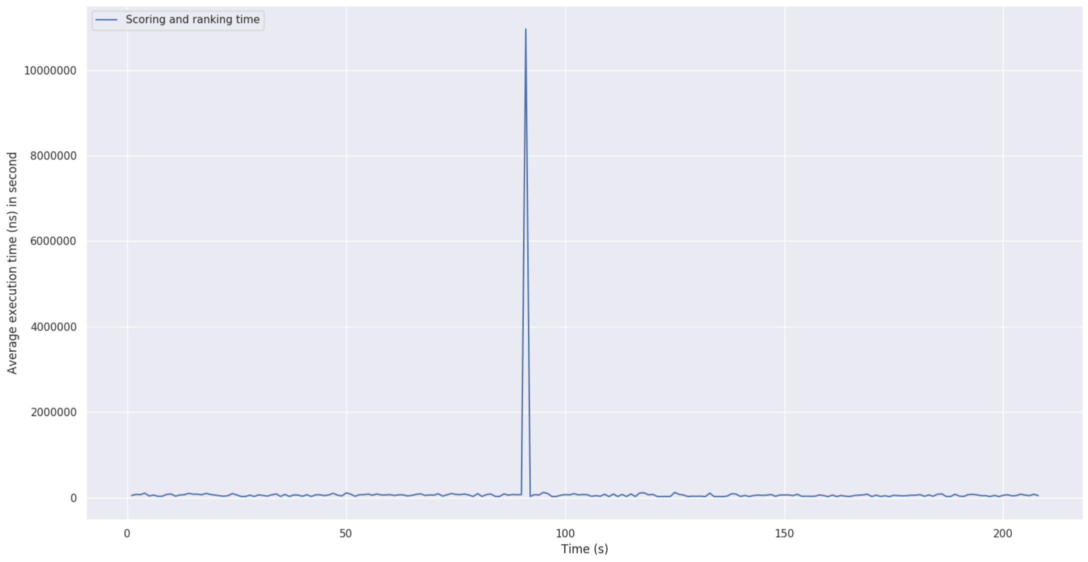

Figure 5.8: Average execution time for scoring and updating the ranking list in a given second during the execution of the experiment which tested the maintenance of the ranking list.

5.3.5 Experiment 4: Total Processing Time of Clustering

Twitter contains an enormous amount of unstructured information. Remember the *variety* characteristic of Big Data presented in Section 2.1: The format of the data can differ, and the data may be of unequal quality. Clustering has been used to group tweets into topics with similar semantics, reducing the search space and overcoming the immense variety in tweets. As the clustering algorithm must be performed online and conform with the timing constraints of real-time analysis, the execution time (by number of threads in memory) should be measured.

Preparatory Work

In this experiment, only the time spent on clustering tweets was to be measured. As with previous experiments, the system was given a warmup phase of 6 minutes and a window size w = 2000. The fixed rate of the `DataGenerator` was set to 3000 tweets/second. The clustering component was isolated, and the `TweetProcessingLogger` configured to log clustering timings. Additionally, the `TweetProcessingLogger` was set up to log the number of clusters currently held in memory, and the number of tweets in the sliding window. However, in this experiment the filter ran before the clustering phase started, as the clustering phase will never be applied to all items of a data stream in real life.

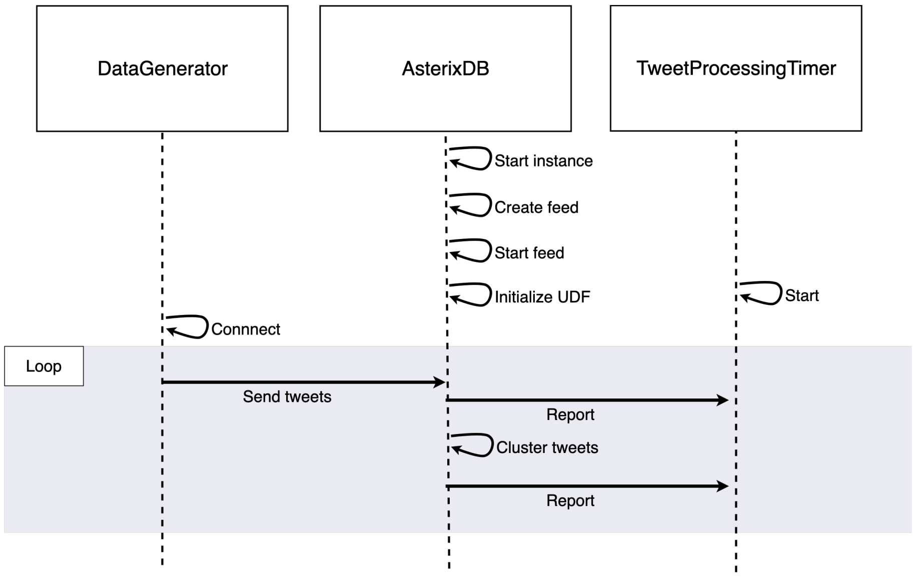

Figure 5.9

Results

Figure 5.10 shows the average execution time for clustering tweets with the number of threads currently held in memory during execution. Additionally, the grey lines indicate the uncertainty of the estimated processing time.

Tweets in window at end of experiment:	361 tweets
Max number of threads during experiment:	42 threads
Median execution time of clustering:	1.484 ms

Table 5.5: Statistics from experiment measuring the execution time of the clustering phase.

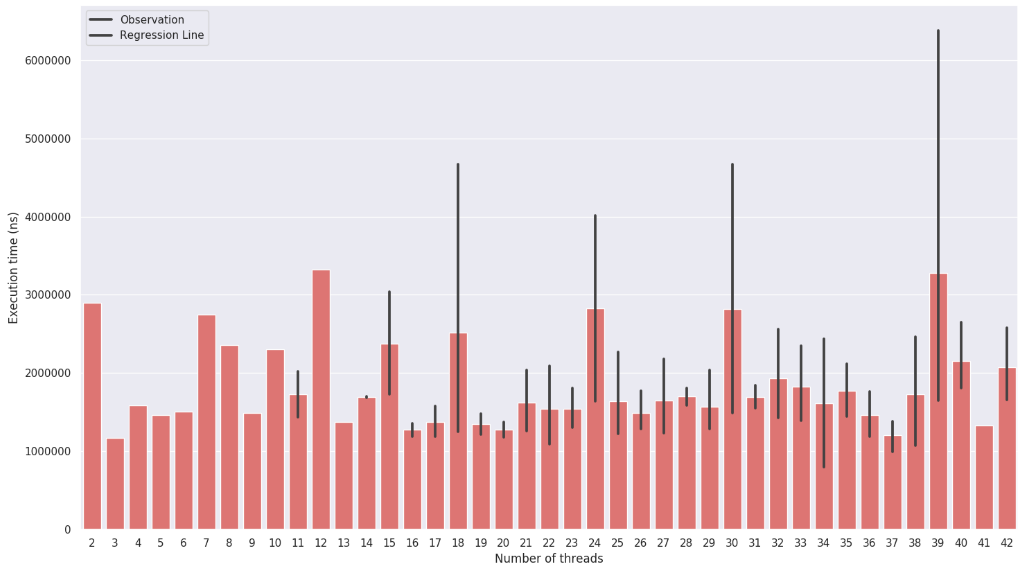

Figure 5.10: A comparison of the central tendencies for the average time spent clustering tweets, by the number of threads in memory.

5.3.6 Experiment 5: Ranking Threads of Tweets

To reduce information overload on users, continuous top-k queries are concerned with only finding the top-k elements in a data stream, instead of presenting all elements matching a user query. Therefore, RQ2 sought out to investigate how clusters of Tweets could be ranked as relevant with respect to a continuous user-defined query, with the aim of only retrieving tweets in the top-ranked cluster. This experiment is devoted to examine the result of the scoring function used to find these top-k elements. Two runs of this test was performed, one using the test data and one using the real-time data from Twitter. As the system will behave differently when using the old ingestion framework as opposed to the new one, two experiments were conducted in order to observe the different behaviours.

Real-Time Tweets and Old Ingestion Framework

Preparatory Work 30 minutes before this experiment was ran, the most prominently featured news was those of the death of the soccer player José Antonino Reyes by car accident. Consequently, the query `Sevilla Arsenal Jose Antonio Reyes dead car` was issued and indexed in AsterixDB in an attempt to ensure a large portion of relevant data. A *DataFeed* was created using the push-based Twitter adapter, as shown in

Listing 5.4. The system could then start to evaluate the data stream of tweets in real-time.

```
CREATE FEED TwitterFeed WITH {
      "adapter-name": "push_twitter",
      "type-name": "Tweet",
      "format": "twitter-status",
      "consumer.key": "******",
      "consumer.secret": "******",
      "access.token": "******",
      "access.token.secret": "******"
    };
CONNECT FEED TwitterFeed TO DATASET RelevantDataset
WHERE detectRelevance(TwitterFeed.text) = TRUE;

START FEED TwitterFeed;
```

Listing 5.4: SQL++ statements used for the real-time experiment.

Figure 5.11 shows the communication timeline between components in the environment of this experiment, which ran for roughly 4 hours and 30 minutes.

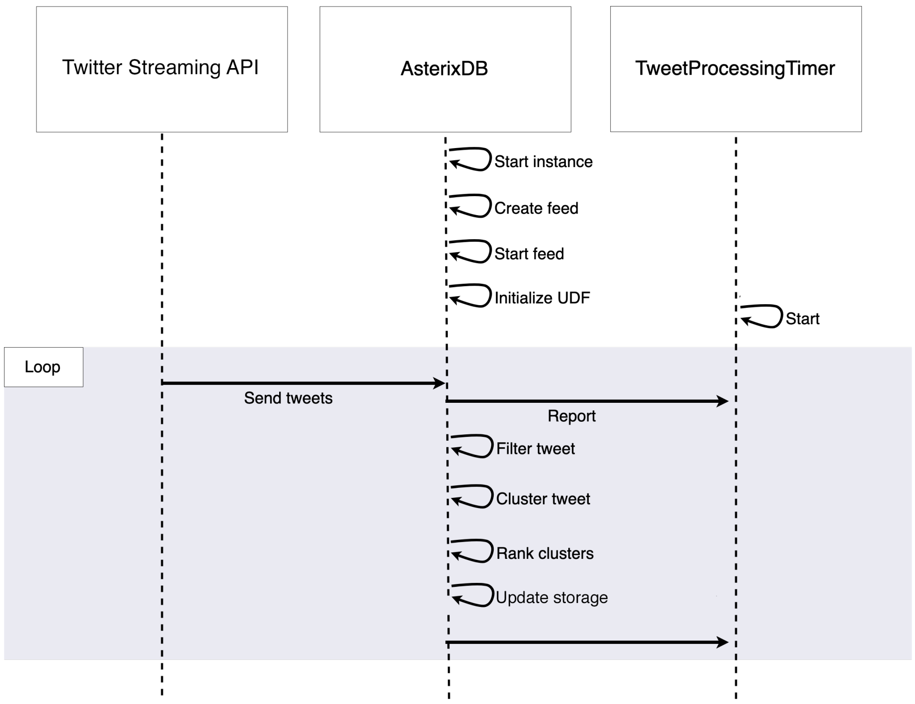

Figure 5.11

Results Table 5.6 shows persisted tweets from a `Thread` which was ranked the highest at the start of the real-time execution of the system. Table 5.7 shows how the data stored in AsterixDB later has changed, as another `Thread` has been ranked the highest. Usernames in the tweets have been replaced by *XXX*. The records from Table 5.6 have been removed from storage, in favour of other relevant – and more recent – tweets. The last top-ranked `Thread` had ID 51, and a score of 0.81124 with respect to the user query.

Created at	**Text**
Sat Jun 01 12:05:31 2019	Former Arsenal star Jose Antonio Reyes dies, aged 35 https://t.co/CdUUvmHIhc.
Sat Jun 01 12:10:06 2019	RIP Jose Antonio Reyes #,, Boy was class at arsenal. #baller
Sat Jun 01 12:11:44 2019	RIP José Antonio Reyes :((https://t.co/cUt5GXiTi3
Sat Jun 01 12:13:44 2019	RT @XXX: RIP Reyes :(I'm shocked
Sat Jun 01 12:16:45 2019	@XXX RIP Reyes #coyg @XXX
Sat Jun 01 12:17:59 2019	RIP Antonio Reyes and @XXX fraternity. It is sad to lose an icon ahead of @XXX final.
Sat Jun 01 12:06:52 2019	RT @SerieA_Lawas: RIP Jose Antonio Reyes. Posternya sempat menghiasi kamar dulu. https://t.co/3z6qHqww2B
Sat Jun 01 12:07:33 2019	RT @XXX: RIP Jose former Arsenal, Sevilla player Reyes dies in car accident https://t.co/hpHdzOS3Zj
Sat Jun 01 12:07:54 2019	RT @XXX: RIP, Jose Antonio Reyes. What a player he was. https://t.co/fbcL5b8YAN
Sat Jun 01 12:10:28 2019	Rest in peace, Jose Antonio Reyes - https://t.co/9lj63VUrXB
Sat Jun 01 12:10:33 2019	RIP Jose Reyes. #Arsenal #Reyes
Sat Jun 01 12:12:33 2019	RIP Reyes - Legend. Invincible. https://t.co/d7BiW6wolu
Sat Jun 01 12:12:51 2019	RIP Antonio Reyes https://t.co/0vVZNQD6eb
Sat Jun 01 12:15:12 2019	Rip Jose
Sat Jun 01 12:17:40 2019	@XXXX @XXX What a player he was. RIP Reyes
Sat Jun 01 12:17:49 2019	This is one of my favourite Arsenal goals ever. RIP Jose Antonio Reyes

Table 5.6: Tweets which have been persisted in AsterixDB from one of the earlier top-ranked threads during real-time execution of the system.

Created at	**Text**
Sat Jun 01 15:51:27 2019	RT @XXX: Jose Antonio Reyes has died tragically in a car accident // Video traffic Video 1 : https://t.co/VnoSj08xlM Video 2 :...
Sat Jun 01 15:52:23 2019	RT @XXX: 35-year-old Jose Antonio Reyes, formerly of Sevilla, Arsenal and Atletico Madrid, has died in a car accident. https://t.co/...
Sat Jun 01 15:56:04 2019	RT @XXX: RIP Jose Antonio Reyes, 1983-2019 The former @XXX forward has died in a traffic collision in Spain: https://t.co/1s...
Sat Jun 01 15:56:49 2019	RT @XXX: Former Sevilla, Arsenal and Atletico Madrid star Jose Antonio Reyes has died in a car accident in Spain at the age of 35. RI...

Table 5.7: Tweets which have been persisted in AsterixDB from the last top-ranked thread at the end of the real-time execution of the system.

Test Data and Old Ingestion Framework

Preparatory Work The system was set up to filter tweets and score threads based on the user query terms `Wayne`, `Rooney`, `Soccer`. The same communication timeline as in Experiment 1 is used, shown in Figure 5.1.

Top-k	**ThreadID**	**Score**
1	37	0.85344
2	50	0.75360
3	46	0.75093
4	3	0.72758
5	18	0.70239

Table 5.8: The top-k rankings from running experiment on test data.

ID	Text
354905	ROONEY !!!!!
354914	WTF ? Rooney ?
357749	GOOOOOOOOOOOOLLLLLLL ROONEY
1842129	DIOOOS ROONEY
355830	Wayne Rooney cant hit a cows ass with a banjo!! #MUFCvsCSKA
357748	Rooney porra
468129	Rooney foda
695006	MANCHESTER UNITED WAYNE ROONEY CHAMPIONS 19 SIGNED SHIRT- MEDIUM- NEW- PROOF COA https://t.co/b5Az2FDH9T https://t.co/Utf1GQN3of
1855905	Rooney off.

Table 5.9: Tweets which have been persisted in AsterixDB from the last top-ranked thread (ID: 37) at the end of the real-time execution of the system.

Results

New, Decoupled Ingestion Framework

Preparatory Work A similar experiment to that above, only using the new, decoupled ingestion framework, was attempted to be performed. However, the system crashed due to limited memory resources. This happened because AsterixDB read the entire Word2Vec model into memory each time a new batch – as discussed in Section 2.5.1 – was processed.

Chapter 6

Evaluation and Discussion

This chapter will perform an evaluation of the experiments conducted in Chapter 5, and discuss the obtained results. Afterwards, the focus will be set on giving a thorough discussion of the strengths and limitations of the implemented solution.

6.1 Evaluation

This section's focus will be to evaluate the results obtained during the experimental phase described in Chapter 5.

6.1.1 Scalability

The results of this experiment are presented in Section 5.3.2. Figure 5.2 shows the total time spent processing *1, 10, 100, 100, 1000, 2000, ..., 10 000, 15 000, 20 000, 100 000, 500 000* and *1 million* tweets respectively when the arrival rate was set to a fixed rate of 5000 tweet/second. In regards to the *processing* aspect of velocity in Big Data, the applied regression show that the processing time has a strong tendency to increase linearly, especially considering the high R^2 value. This yields a time complexity of $\mathcal{O}(n)$ for the total processing time of tweets. This is important as it indicates scalability. Considering that this experiment is only performed on a single, local instance of AsterixDB, it could be interesting to distribute the processing by scaling the number of nodes in the cluster and compare the results.

Looking at Figure 5.4, it shows the average processing time per tweet with increasing arrival rate. An observation that can be made from this figure is that the lower the arrival rate of tweets, the higher the variance in processing times. A possible explanation for this may be that operations which affects the processing time has a greater impact on the total time when the arrival rate is low. Another explanation could, when investigating the graphs in Appendix B, be that the system during some of the runs with arrival rate of 1000 and 2000 tweets/second simply did not benefit as much from the warm up phase compared to higher arrival rates, making the processing times more unstable. Figure B.4 and B.5 shows that the second and third runs using 1000 tweets/second arrival rate have a high variance in processing time, and Figure B.6 shows that the system did not benefit from the warm up phase during the first run of 2000 tweets/second. It is possible that the average processing time would have looked differently if the warm up phase lasted longer than the dedicated 6 minutes. One interesting finding in Figure 5.4 is that the averaging processing time per tweet from three runs show that the execution time becomes more stable with higher arrival rates. This makes it reasonable to assume that the average processing time will continue to be stable when the arrival rate is higher.

The actual number of tweets/second handled by the system when adjusting the arrival rate of the `DataGenerator` are presented in Figures B.2, B.6, B.10, B.14 and B.18. Table 5.3 shows an overview of the averages from all three runs, and these results show that the system was unable to handle the same arrival rate as sent from the `DataGenerator` in all cases. As the arrival rate is increasing, the system will eventually be able to manage approximately 2/3 of all tweets sent to it. Some of this deviation may be due to fact that the `DataGenerator` and the local AsterixDB instance were running on the same machine and were as such sharing resources. The isolation in this experiment should be improved to see whether the system is able to handle an amount of tweets more similar to the arrival rate. However, this alone should not limit the handled arrival rate as much as seen in the result. Other reasons could be that other processes on the machine are competing for resources, or that the external libraries used in the system are not built for real-time processing. Further investigation into the root cause of the tweet rate mismatch would require additional experiments to be performed.

It is worth noting that the processing time of tweets is closely related to the number of tweets that are found relevant for the given user query in this experiment. Few relevant tweets results in less time spent on complex processing tasks such as clustering, scoring and ranking. As this experiment was performed in order to explore how the system handled different velocities, it was considered reasonable to include the filtering function as this would also be applied in a real-time execution of the system. It should also be noted that the number of observations are fewer in the case where 5000 and 10 000 was set as arrival rates, because all tweets in the dataset were finished processing before the experiment run time of 10 minutes was over. This is because a high rate of tweets are sent during the warm up phase.

Furthermore, only the processing performed within the UDF is reported to the `TweetProcessingTimer`. Therefore, the process of storing the tweets – which happens right after the UDF is finished evaluating – is not contained in these results. Obtaining data regarding the processing time outside the UDF would require changing the source code of AsterixDB. This is outside the scope of this thesis, which is rather to utilize AsterixDB *as is* for the use case described.

6.1.2 Filter Based on a Continuous User-Defined Query

Looking at the results in Figure 5.6, the observations follow an increasing regression line as expected, as the number of keywords to match increases as the query is expanded. This indicate a time complexity of $\mathcal{O}(n)$ of the filtering function.

One unanticipated finding is that this graph shows that the average execution time at 70 and 80 keywords were higher than that of 90 keywords. Similarly, the execution time at 90 keywords was higher than at 100 keywords. The expected behaviour would be that the highest processing time was at 100 keywords. To explain this, all the 130 000 data observations which were used to generate the averages processing times per n must be considered, which are presented in Figure B.22. This graph shows that there are only three outliers, at 70, 80 and 90 keywords respectively. These have contributed to the very high average execution times seen in the previous graph. It is difficult to explain exactly why these outliers occur, but considering that the same dataset has been used in each of the experiments run for the different values of n, it could be due to the same tweet being processed. Investigation of which position the highest processing times occurred, found the results to be the 6235/10 000 tweet processed for $n = 70$, and the 8793/10 000 tweet processed for $n = 80$. This can help justify the theory that the same tweet caused the problem, as $n = 80$ would have generated more matches beforehand, causing the same tweet to be evaluated a bit later during the run in $n = 80$, although it is not possible to conclude with anything specific.

Retrieving the keywords to use for query expansion is performed in the initialization phase of the UDF. This means that the time used by the Word2Vec model to obtain n closely related keywords does not affect the execution time of the filtering function. However, if this was not performed during initialization, the keywords would have been retrieved once per incoming tweet, adding a complex processing step to every evaluation.

Admittedly, when disregarding the three aforementioned outliers, the findings indicate that the execution time of the filtering function is stable, and that the system is able to handle an increasing number of keywords. The findings also indicate that applying a continuous user-defined query to the data stream has a negligible cost related to it, when at the same time obtaining advantages such as the filtering of irrelevant data to reduce the information overload of the user. It must also be noted that these execution times are in nanoseconds. The slightly higher observed execution times are still relatively low, around 250 ms.

6.1.3 Maintaining Top-k Ranking

Figure 5.8 shows the average processing time to score and rank threads in-memory, and to update the storage if the top-ranked `Thread` is replaced. What stands out in this graph is the high execution time in $s = 91$. This is however explained by another `Thread` grabbing a hold of the first place in the ranking list, consecutively sending a request for deleting all records in the dataset. Closer inspection of the results implies that the time spent maintaining the top-k ranking, i.e. scoring `Threads` and updating the `TreeMap`, is stable. As the `TreeMap` offers $\mathcal{O}(logn)$ complexity, this is a possible explanation for the stable results. Additionally, none of the data structures (`TreeMap` or `Threads`) holds a very large amount of data at any point during the run. Top-k has been set to $k = 5$ in this study, and the window size to $w = 2000$. This restricts the sizes of both the `TreeMap` and the number of `Threads` in memory. The number of `Threads` is upper-bound by w, as it can not be created more clusters than the number of tweets in the window. The number of `Threads` to maintain in the ranking list is upper bound by k.

When the top-ranked thread is replaced by another one, the system performs a HTTP request to AsterixDB's API, deleting old records from storage. A reasonable expectation to have is that the high latency related to sending a HTTP request could "clog" or create a bottleneck in the system, piling up tweets waiting in line to be processed and consequently slowing down the pipeline. This is because the API request had to be handled by the Cluster Controller process, which the UDF also drains resources from. However, during the second after deleting the records, in $s = 92$, the system spent an average of 0.0305 ms processing each tweet. This implies that the system does not seem to be affected by the process of deleting records from storage. This is important, as performing a HTTP request was the only solution found during this project to change records *after* their arrival time in AsterixDB. UDFs are built for modifying arriving data, not already persisted data.

The plot found in Figure 5.8 contains only one outlier, which is connected to the maintenance of the *persisted* records. This implies that the *median* execution time would serve as a better measure than the *average* as a means of evaluating the cost of scoring and updating the top-k ranking in-memory. With a median value of 0.0596 ms, the ranking maintenance lies far within the timing constraints required for real-time analysis.

It must however be specified that average processing time of 10.958 ms during $s = 91$, when the old records in the dataset is removed, is still not overstepping the time constraints of real-time analysis. This finding has important implications for implementing ranking using AsterixDB, as it is performed by a HTTP request to AsterixDB's API. If a triggering functionality was implemented as a core feature in AsterixDB, the time to update records could be reduced.

Overall, these results indicate that maintaining a ranking list of the elements in a data stream is inexpensive in the proposed system. Some cost is related to updating the persisted records, which happened once during the experiment.

6.1.4 Processing Time of Clustering Tweets

The result of this experiment is shown in Figure 5.10. When clustering, tweets are added to the sliding window, their most similar – if existing – `Thread` is found, `Threads`' centroids are updated as tweets are added or removed from them, and lastly, tweets are removed from the window. As the result implies, the average execution time for this process is quite stable even if the number of threads held in memory increases, and the results show no relation between the number of `Threads` and the execution time. As the `Threads` are maintained in a `HashMap`, and the time complexity of puts and lookups in `HashMaps` are in theory $\mathcal{O}(1)$, which is most likely the reason for the stable processing time, and means that the observations matches the theory. It can therefore be assumed that whether the number of `Threads` held in memory are 10, 50, 100 000, or even more, the system should be able to quickly cluster a tweet. By looking at the central tendency in the dataset, the median of the clustering time is roughly 1.5 ms, as shown in Table 5.5. This should be regarded as accepted for real-time processing.

Contrary to expectations, there are however some outliers with high execution time amongst the observation, creating the high error rates at 39 and 18 `Threads`. As the `HashMap` used for maintaining the `Threads` have $\mathcal{O}(1)$ time complexity, one would expect all execution times to be fairly similar. A possible explanation for these observations may be the fact that an external library is used for implementing parts of the clustering, namely *deeplearning4j*. As a consequence of it being external, it is unmanageable and for all practical purposes acts as a black box system in the context of this thesis, and is a possible source to inconsistent results.

The processing time for clustering tweets are higher than the average, total processing time of tweets in the system reported in Experiment 1 in Section 5.3.2. This is because this experiment, as opposed to Experiment 1, measures the time to cluster tweets which have actually entered the clustering phase by passing the filter. This is also why the maximum number of tweets in the window during the experiment were 361, even though the window had a size of 2000. On the other hand, Experiment 1 aims to mimic the entire process. In a typical scenario, a large amount of tweets in the data stream will be irrelevant for the user query, and therefore disregarded in the filtering phase, lowering the average processing time.

These findings imply that that the incremental clustering approach adopted is suitable for online, in-memory clustering. The reported median value for the processing time justifies the fact that clustering has been adopted to reduce the search space and limit the amount of maintenance required for the ranking list.

6.1.5 Ranking Threads of Tweets In Real-Time

Using the Old Ingestion Framework

The purpose if this experiment was to demonstrate real-time ranking of clustered tweets in-memory from a data stream. Tweets are evaluated at arrival, and those which are allocated to what is considered the topmost ranked `Thread` with regards to the user query are consecutively persisted. Tweets are clustered by utilizing the pre-trained model for calculating semantic similarity. This model is maintained in-memory as long as the *DataFeed* is connected to the Twitter API. By taking advantage of several of the features offered by AsterixDB, every tweet in the data stream has been evaluated by a customizable UDF.

Recall the set of requirements for the behaviour of the ranking component, which were stated in Section 4.3.2. A list of the top-k ranked items must be preserved at every time instant. This requirement is satisfied, as a `TreeMap` in the UDF holds the top-k `Threads` in-memory throughout the entire time the *DataFeed* is connected and started. If a new element arrives, the top-k elements should be updated if necessary, potentially updating the records stored in AsterixDB if they are found to not be relevant anymore. The system behaviour also complies with this requirement, as only tweets which resides within the current top-ranked `Thread` are inserted into storage. If the top-ranked `Thread` is replaced, the system also deletes the old records in the dataset, ensuring that only tweets which resides in the top-ranked `Thread` at every instant are persisted.

Updating the top-k elements should not be too time consuming, nor be too computational heavy, as it must be done in-memory and in real-time. By utilizing data structures with $\mathcal{O}(1)$ and $\mathcal{O}(logn)$ look up time complexity for `Threads` and for the top-k ranking respectively, and also by maintaining these in-memory in the UDF, this requirement has been met. Additionally, based on the results presented in Table 5.4 from Experiment 3, the processing time is stable: The processing times averages at roughly 0.11 ms per tweet, and the median is 0.0596 ms per tweet. Considering these processing times and the data structure complexities, the requirement of time consumption and computational complexity has been fulfilled.

To limit the amount of items to rank, the scoring function should be applied after tweets have passed the filtering function. As the filter works as a continuous user-defined query, continuously asking for all tweets which matches a given number of keywords from the query, all other tweets are disregarded. This phase happens before the tweets are clustered and ranking applied, which means that this requirement has been met as well.

The scoring function should be query-dependent, by considering a user-defined query. A user query is inserted in a dataset in AsterixDB prior to the *DataFeed* being connected to the Twitter API. When the *DataFeed* is started, the UDF attached to it begins its initialization phase. During this phase, the user query is obtained from storage by sending a HTTP request to AsterixDB's API, and stored as a class variable in the UDF. This query can then be used when calculating the score of the `Threads` in memory. The calculated score expresses the semantic similarity between the user query and the content of a `Thread`. This requirement has therefore been met.

Item freshness should be considered. By keeping all tweets which have passed the filter in a count-based sliding window, only tweets in this window are subject to being persisted. By removing tweets from the window as new ones arrive following a FIFO queuing strategy, item freshness is taken into consideration. Additionally, as old tweets are removed from storage when the topmost ranked `Thread` changes, this also help to conform with the item freshness requirement. Overall, the proposed solution takes item freshness into account.

The old ingestion framework is not built to be sensitive to changes in reference data. If the user query changes while the system is running, the system would not be able to reflect this change by using the new query for evaluation.

Using the New Ingestion Framework

As the batch size must the set using this framework, the initialization phase of the UDF will be invoked once per batch. If the batch size is too small, the initialization phase will be called often, reading the model into memory just as often. The garbage collection in Java is not able to release the memory used by the model during the previous batch, so the number of models read into memory increases with the number of batches processed, which is not scalable, and was also the reason for the system crashing when trying to perform Experiment 5. Additionally, the initialization phase last for about 2 minutes on the machine used for performing experiments. This slows down the pipeline considerably. However, when a batch is finished, the system will read the user query again, making it sensitive to possible changes made to the user query. This is an important feature, and was the main reason for putting in effort in trying to make the system work using the new framework. However, this feature does not justify the increase in memory usage and the absence of some form of garbage collection being performed between batches.

6.2 Discussion

This study have investigated how to maintain a continuous user-defined query, use this query for filtering tweets in a data stream, cluster tweets by semantics, and – most importantly – rank these clusters of tweets with respect to the user query. The unified BDMS AsterixDB has been used as a use case for the task. In this section, the strengths and weaknesses of the final implemented system will be discussed.

6.2.1 Solving The Use Case with AsterixDB

The issue of overloading users with information can be prevented by filtering and ranking tweets in AsterixDB. In the suggested solution, one UDF is applied as a query predicate to the *DataFeed* connected to the Twitter API. This UDF is responsible for filtering out tweets which does not match any of the keywords in the user query, and to rank items from the data stream. In the case of no filtering or ranking by a user query being applied to the data stream items, every arriving item in the stream would be stored in AsterixDB. This would both require extensive storage space, and it would retrieve too many tweets for the user to handle – which essentially is what we are trying to avoid. Since there would be no filtering of which tweets to retrieve if this functionality was not added, the information need of the user would not be met in a precise manner.

When using AsterixDB to process items in a data stream, incoming items can only be evaluated and inserted at their arrival. This is handled in the proposed system by using a single-pass approach instead. For instance, given a tweet t which arrived at t_0. In the current system, t will be filtered, clustered into cluster c_1 and persisted to storage if found to reside in the top ranked cluster. If, however, this cluster was *not* ranked as top at time t_0, t would never be stored in AsterixDB, even though c_0 became the top ranked cluster at the next time instance t_1. This is a limitation of the current approach. Being able to insert items into storage in an efficient way later than at their arrival would ensure that tweets that become relevant right after their arrival time is not lost.

Babu and Widom [9] discusses the concept of triggering mechanisms, which could be used to perform certain activities when specific events occur. As AsterixDB does not offer a triggering mechanism, there is no straightforward way to insert items if they are found relevant *after* the time of arrival, which is possible in multi-pass approaches [34]. If this functionality was offered, records of tweets which are not relevant at their arrival, but may become so at a later point in time, could be maintained in-memory, and a triggering mechanism could persist the records when they became relevant. This would however require methods for only choosing the candidate tweets, as it is not feasible to maintain *all* tweets in-memory.

Attaching a query predicate to the data stream enables the system to discard tweets as soon as they are found to not contain any relevant keywords for the user query. As shown in Experiment 2 in Section 5.3.3, the Word2Vec model also allows for finding the n most semantic similar words to the keywords in the user query. This can be used to perform query expansion. When the implemented filtering function only matches on the original keywords without considering query expansion, tweets which use different words with similar meaning than those in the user query will be lost.

As discussed in the implementation phases in Section 4.3, there were some challenges with AsterixDB which were discovered during development that have affected the solution. The first issue was the case of stateful versus stateless UDFs. As it was found important for the use case in this thesis to make the system sensitive to changes in the user query, a lot of time was spent implementing the solution using the new, decoupled ingestion framework presented in Section 2.5.1. This framework is new, and had not been tested on many users. Exploring if this new, decoupled framework could be used in this use case resulted in a total of five different versions being sent to us from one of the developers at AsterixDB, listed in Table C.1.

When using the old ingestion framework, the UDF is initialized once. During the initialization phase, the system retrieves the indexed user query. As this phase is only ran once in the old ingestion framework, the system is unable to detect changes to the user query. However, initializing the UDF only once would in turn only require the model to be read into memory *once*, which is advantageous.

The system will behave differently if adopting the new ingestion framework rather than the current/original ingestion framework. When using the decoupled framework, the batch size must be set before the *DataFeed* is started. As the intake job is invoked once per new batch, this implies that the initialization phase of the UDF will be ran just as often. Choosing the batch size is therefore important with respect to both efficiency and sensitivity to referenced data, such as the user query, as it impacts how many records are processed using the same reference data. When tuning the size, one can decide whether ingestion performance or the awareness of changes to the referenced data is most important. When using a low batch size, the initialization phase will be ran often, and the system can then consider updates to the referenced data. Being sensitive to change in reference data is important if changes in the user query (information need) ought to be taken into account. If the number of records in a batch is large, the records will not be sensitive to changes in the referenced data [56]. However, running the UDF initialization phase often would also require reading the Word2Vec model into memory often. Depending on the runtime environment, this can take between 2 and 15 minutes.

In the experiments performed in this thesis, the user interface provided by AsterixDB has been used to insert user queries. This is not an optimal approach, as it requires the user both to be able to start a local instance of AsterixDB on their computer, as well as possess knowledge of the semantics and syntax of SQL++ statements.

6.2.2 Continuous Top-k Query

When implementing ranking together with continuous queries, items in the data stream are both filtered and ranked. This must be done continuously, so the solution is required to keep track of the top ranked items at every instant. By utilizing a data structure with fast lookups, the system can efficiently decide whether the score of the `Thread` it is evaluating calls for the ranking list to be updated. This evaluation is performed for every tweet that arrives, but the number of updates to the ranking list is limited by clustering tweets, and letting the cluster's centroid score be evaluated against the ranking list rather than each tweet's score. Adding tweets to clusters updates the centroids, but a threshold defines how similar the two must be for the tweet to be added to it. Adding a tweet to a cluster of already similar tweets will not impact the centroid query score by a large amount, and will therefore decrease the chance that the cluster's placement in the ranking list changes on a per-tweet basis.

Tweets which are evaluated to belong to the top-ranked `Thread` are continuously being inserted into a dataset in AsterixDB. However, the system does not delete persisted tweets before the similarity score of another `Thread` has surpassed that of the current top-ranked `Thread`. This means that if the highest ranked `Thread` remains on top for a long time, tweets which are stored may become old. But, as the `Thread` centroid is calculated by the tweets currently in it, which is a subset of all tweets in the sliding window, the `Thread` will have no content to match the user query if all tweets which was earlier in it have been removed from the window. The `Thread` will naturally decrease in score if all its tweets are removed from the window and no other tweets are added to it. This would then make another `Thread` the top-ranked one, which in turn would trigger the updating of the records stored in AsterixDB.

As mentioned in Section 1.4, building indexes on queries are suggested to improve continuous queries. In this project, an index has not been built on the query, as this would require to perform alternations to the source code of AsterixDB. This project is going to show how AsterixDB *as is* works for the given use case, and therefore this was not performed. Making additions to the source code is also not desireable, as AsterixDB should be a BDMS with features that normal users can adopt for their use cases.

Data Structure

A TreeMap was implemented to maintain the top-ranked threads for the specified user query at every time instant. This is a red-black balanced tree structure, which employs a natural ordering of its keys. It offers $\mathcal{O}(nlogn)$ complexity for inserts and $\mathcal{O}(logn)$ complexity for lookups. This means that finding the lowest member in the ranking list, when using similarity scores as keys, can be done with $\mathcal{O}(logn)$ complexity. In this project, n is a constant defined by k, the number of elements to hold in the ranking list, meaning $n = k$.

A limitation of the TreeMap data structure is that it can not hold two equal keys. This implies that if two or more threads have the exact same score, they can not exist in the ranking list at the same time. In the scenario of two threads having the exact same similarity, the current thread would simply be updated by the newest thread with the same score. This means that the one of the two threads which were most recently updated will be given the spot in the ranking list.

6.2.3 Word Embeddings

As the results showed in ,the clustering algorithm was able to group tweets which discussed the same topic into the same threads. This is achieved by using a pre-trained model built on news. Imagine the possibilities word embeddings can give when for instance adopting a model trained on Twitter data. Another strength of representing data using word embeddings, is that the Word2Vec model can be used to perform query extension. This is beneficial due to the short length of tweets, as extending the query with semantic similar words could results in more precise matches during the filtering phase. Adopting a AvgWord2Vec representation for Threads' centroids in this project has also worked well. A limitation of using the pre-trained Word2Vec model is that it is unable to consider words which it has never seen before. Due to the noisy text in tweet, there are many words in tweets which the model is unable to create a word embedding for, as it has never seen the words before. It should therefore be considered to a model trained on Twitter data, as will be suggested in Section 7.2.

6.2.4 Information Freshness Using Sliding Window

Only tweets which have passed the filtering function will be maintained in the sliding window. A cluster's score is updated as often as tweets arrive or are removed from the sliding window. However, there is one notable weakness of only holding tweets which have passed the filter function in a sliding window: If the user query consists of keywords which rarely matches the arriving tweets, the sliding window will increase in size slowly, which in turn means that it may take some time before the system starts to remove tweets from the window.

6.2.5 Comparisons to Related Work

The work of Norrhall [38] presented in Section 3.2 can be compared to the proposed solution in this thesis. In that solution, all tweets that match the query are stored. The proposed ranking function in that system continuously ranks and updates the score of every arriving tweets and the stored tweets based on a continuous user-defined query. To combat stale data, a decay function is adopted, meaning that older tweets will be considered less relevant. This is in contrast to the solution proposed in this thesis, which updates the storage by removing old records, and by only storing tweets which belong to the highest ranked cluster, thus limiting the number of retrieved items. However, Norrhall [38] is able to achieve a high, stable throughput of up to 15 000 tweets/second. In the approach proposed in [38], all arriving tweets are stored and indexed in Elasticsearch. The suggested solution in this project maintains a list of the top-k most relevant clusters, and only persists tweets belonging to the highest ranked cluster, instead of all arriving tweets.

BAD, as presented in Section 3.1, is as extension of AsterixDB. It uses the Publish/Subscribe pattern, letting users subscribe to different topics. Due to it adopting this pattern, BAD does not perform ranking of the data stream elements. The solution presented in this thesis, the data stream items are filtered by a query defined by a user. Additionally, clusters of data stream elements are ranked before persisted in AsterixDB. Thus, only data which the user has specified its interest in will be indexed. This solution does not notify users, but continuously updates the element which are retrieved, and by deleting old data. BAD, however, stores all data from the specified data publishers.

6.2.6 Limitations

In this project, the scope was set to maintain *one* user query as a proof-of-concept. This means that a multi-user aspect is not considered. The implemented solution is not suitable to maintain several queries as is, as it would be time consuming to update the ranking list of all queries. This is because the query is not maintained in an efficient data structure built to handle updating the ranking list of several queries, such as proposed by others [54][62].

Chapter 7

Conclusion and Future Work

This chapter finalizes the thesis, and gives a conclusion to the work conducted. The answers to the research question will be given sequentially, followed by a list of future research.

7.1 Conclusion

The task undertaken in this project was to investigate whether textual streaming data in the form of tweets could be continuously ranked in real-time by using a unified BDMS. A solution to this problem has been implemented using the features offered by AsterixDB. The issue of information overload towards users has been dealt with by filtering the data stream with a continuous user-defined query, and by ranking clusters of the remaining elements.

The first phase of this project explored the state-of-the-art, and implemented the required components for performing ranking on streaming data. During the second phase, all the components were combined, creating a solution which could perform real-time ranking of tweets based on a continuous user-defined query. The experiment phase showed that the system had a linear time complexity for the processing an increasing amount of tweets. It also showed that the system struggled with handling a throughput equal to the arrival rate, and that the filtering function had linear time complexity. Time spent maintaining the top-k clusters is stable, with a median execution time to maintain ranking list of 0.0596 ms/tweet, and an average of 0.1104 ms/tweet. The median execution time for clustering was 1.484 ms/tweet.

A summary of the most important conclusions that can be drawn from the conducted work are as follows:

- Ranking of real-time Twitter data can be performed using the unified big data management system AsterixDB, but streaming elements can only be inserted at their arrival.
- If a triggering mechanism was available in AsterixDB, this mechanism could monitor specific events, such as changes in the ranking list, and delete persisted records or insert streaming elements stored in memory when such events occurred. This would eliminate the need for making API calls from the UDF which slows down the pipeline more than necessary.
- Maintaining the top-k items for a single user query in a `TreeMap` is not computational heavy. By ranking clusters instead of each tweet, there are also less frequent updates required.
- Ranking items in the data stream with a continuous query shows a significant decrease in the amount of data retrieved compared to retrieving every tweet in the stream, overcoming the issue of information overload.
- Experiments show that the system is only able to achieve a throughput of approximately 2/3 of the arrival rate.
- The old/current ingestion framework is stable when running the real-time experiment. Contrary to the old ingestion framework, the new framework would make the ranking component able to respond to changes in the user query, but at the same time introduce issues when keeping a large model in memory, making the system unable to run for longer than two batch sizes due to memory restrictions.
- The system is able to retrieve relevant tweets by only utilizing word embeddings from a pre-built model, without using deep learning in real-time, nor having to learn continuously as the system is running.
- Ranking maintenance for one user query is within the timing constraints for real-time analysis. Updating the storage required sending an API request from the UDF at runtime, but the time was low (k restricts the size of the data structure maintaining the ranking list.)
- The incremental clustering approach adopted for reducing the search space is suitable for online, in-memory clustering. The reported median processing time justifies the adoption of clustering to reduce the search space and limit the amount of maintenance required for the ranking list.

7.1.1 Goal Achievements

RQ 1: *How can relevant tweets be detected in a Twitter stream based on a continuous query defined by a user?*
This research question sought to answer how to set up a pipeline for retrieving elements from a data stream of tweets in real-time, and how to filter and only pass along those relevant to a continuous user-defined query. In the suggested solution implemented using AsterixDB, a *DataFeed* was created and configured to use the push-based Twitter adapter,

enabling it to receive and parse tweets by only sending *one* initial request. Furthermore, it had to be connected to a *Dataset*.A user query was inserted into a dataset using the user interface offered by AsterixDB. To make this query into a continuous query, the query first had to be obtained by sending a request to AsterixDB's API during the initialization phase of a UDF. This was found to be the only solution when implementing the UDF as a query predicate, but a triggering mechanism would better this part of the solution. By attaching the UDF to the *DataFeed* as a query predicate, and using the query content to match the content of the arriving tweets, all tweets in the data stream are filtered. This made the query available throughout the whole lifespan of the *DataFeed*. As all arriving tweets are processed by the UDF query predicate, which filters based on the user query content, only items matching the use query are retrieved in this phase. By filtering the arriving data as first step in UDF, irrelevant tweets are quickly discarded, and are thus not inserted into storage. Experiment 2 in Section 5.3.3 also showed that query expansion could be used by adopted in the system by utilizing the Word2Vec model. This is possible since it was chosen to maintain both the filtering and ranking within the same UDF, and these phases can therefore share the UDF if desired. If another of the approaches was taken in *1.* and *4.* in Section 4.3.2, this would not have been possible.

As a pipeline which can continuously retrieve data from Twitter has been set up, and that a continuous user-defined query is used to filter the data stream, this research question is considered answered.

RQ 2: *How can clusters of Tweets be ranked as relevant with respect to a user defined query?*
To answer this research question, a way for expressing whether a cluster centroid was relevant for a user query was needed. Relevance was defined to be the semantic similarity between a cluster with recent tweets and the user query. The ranking functionality was implemented in the same query predicate UDF as the filtering discussed above. To ensure the ranking takes the user query into account, the user query made available during the intialization phase is used when evaluating the `Threads`. Tweets which have passed the filter are maintained in a count-based sliding window and are further clustered by semantic similarity. This ensured that both of the relevance aspects was considered. The scoring function used the word embedded representations if cluster centroids and the user query to calculate their semantic similarity. Word2Vec implementation of word embeddings was used to obtain a low dimensional representation of tweets, cluster centroids and the query.

A `Thread`'s score is updated whenever a tweet is added or removed from it, which happens when a tweet is being added or removed from the sliding window. Each time the window shifts, the `Thread` which the tweet resides in are evaluated against the current ranking list. A `Thread` will never live longer than w shifts of the sliding window if it never is updated after its creation.

To continuously maintain the top-k ranked `Threads`, an in-memory data structure with fast look up of the lowest score and for insertion in the ranking list was as used. The data structure is updated if the arrival of a tweet caused the positioning of the current top-k items to change.

To examine the final behaviour of the ranking component, and to see whether it conformed with the requirements set in Section 4.3.2, an experiment was performed on real-time Twitter data.

RQ 3: *How do continuous queries and ranking affect the retrieved results?* In the proposed solution, a continuous query is implemented as a filter on the data stream, only retrieving the tweets found to be relevant in terms of matching words in the user query, or the expanded version of the query. Continuous queries requires the system to update the ranking list on the fly at the arrival of new items. As this is to be performed in real-time, continuous queries requires the system to operate at high speed. Applying a scoring function further helps reduce the information overload effect, as only the highest ranked items are retrieved and persisted. But, if the continuous query is used for filtering by only matching words from the user query without performing query expansion, relevant tweets can be lost.

When ranking with a continuous query is applied to the data stream, the search space is reduced. When applying ranking with continuous queries on the data stream, the system is able retrieve relevant tweets, as shown in Experiment 5 in Section 5.3.6. The number of retrieved tweets are reduced from the entire size of the data stream to only a small set of tweets. As also shown in the same experiment, maintaing a ranking list ensures that old records stored in AsterixDB can be switched out as time passes by. This means that a user will receive up-to-date tweets.

Main RQ: *How to make ranking of tweets in real-time efficient with continuous queries?* All of the abovementioned research questions have been used to guide the research and have therefore looked into how to answer this main research question.

Project Goal: *Investigate how ranking can be implemented to retrieve the most relevant set of tweets from Twitter with respect to a user-defined query in an efficient manner.* Experiment 5 presents a set of tweets which the system has been able to retrieve from the real-time Twitter stream, by performing ranking with a continuous user-defined query. AsterixDB was used as the underlying technology for being able to implement this solution. The project goal must be seen in context of the scope and limitations of this project introduced in Section 1.4. With the scope in mind, the project goal is considered to be met.

7.2 Future Work

The system currently only maintains one user query, and there is little cost related to updating the ranking list of this single query. However, an optimal system should be able to scale to any given number of user queries, and do so efficiently. To achieve this, it should be possible to build indexes on user queries instead of records in AsterixDB. For future research, it should therefore be investigated how to maintain more than one user query in an efficient way in AsterixDB. To do so, indexes should be built on queries, and efficient in-memory data structures should be created, such as shown in [54].

Considering that experiments are only performed on a single, local instance of AsterixDB, it could be interesting to distribute the processing by scaling the number of nodes in the cluster and compare the results with the single node solution.

Tweets which at arrival time are clustered into the highest ranked `Thread` are stored regardless of it being a re-tweet of another tweet already persisted. When considering a user's information need, it is not necessary to store the exact same tweets, as the information it provides will already have been given. This should be improved by considering other fields in the tweet object which indicate whether the tweet is a re-tweet or not, or simply check for "RT" in the tweet content when processing it in the UDF, to find out whether it is a re-tweet or not.

In this project, query expansion was only performed in Experiment 2 while observing how the system handled an increasing query size. It would be interesting to investigate the effect on the retrieved result when query expansion is utilized and the arriving tweets are matched against the expanded query in comparison to only matching against the original query. There exists a model[1] built on tweets as well. Comparing which tweets this system retrieves when using the different models is also subject for further investigation.

Tokenization is the task of dividing the text into paragraphs, sentences or words [31]. Additionally, some characters such as punctuation marks and commas are discarded. Texts consist of many frequent words such as "I, am, to, in", and these words do not contribute to differentiating the content of texts and are called *stop words*. Because of this, it is common to remove these word both from queries and texts. This have not been done in this thesis, but it would be interesting to investigate which effect this have on the generated clusters. Another important step is Named Entity Recognition (NER) tagging. The aim of using NER is to determine if the content in a tweet refers to an entity, such as a person, an organization or a location. Detecting if a tweet contains a location is desirable in order to use this information to detect if the content of a tweet revolves around a relevant location, as specified by a pre-defined user query.

[1]https://github.com/loretoparisi/word2vec-twitter

Bibliography

[1] Abdelhaq, H., Sengstock, C., Gertz, M., 2013. Eventweet: Online localized event detection from twitter. Proc. VLDB Endow. 6 (12), 1326–1329.

[2] Abrahamson, T. M., 2017. Scaling machine learning methods to big data systems. Master's thesis, NTNU.

[3] Aggarwal, C. C., 2015. Data Mining: The Textbook. Springer Publishing Company, Incorporated.

[4] Alkowaileet, W., Alsubaiee, S., Carey, M. J., Li, C., Ramampiaro, H., Sinthong, P., Xang, x., 2018. End-to-end machine learning with apache asterixdb. In: Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning. DEEM'18. ACM, pp. 6:1–6:10.

[5] AsterixDB, 2018. About apache asterixdb. `https://asterixdb.apache.org/about.html`, online; accessed 30.11.2018.

[6] AsterixDB, 2018. Data ingestion with feeds. `https://ci.apache.org/projects/asterixdb/feeds.html`, online; accessed 10.11.2018.

[7] AsterixDB, 2018. User-defined functions. `https://ci.apache.org/projects/asterixdb/udf.html`, online; accessed 21.10.2018.

[8] AWS, ???? What is streaming data? `https://aws.amazon.com/streaming-data/`, online; accessed 08.04.2019.

[9] Babu, S., Widom, J., 2001. Continuous queries over data streams. SIGMOD Record 30 (3).

[10] Baeza-Yates, R. A., Ribeiro-Neto, B., 1999. Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc.

[11] Bojanowski, P., Grave, E., Joulin, A., Mikolov, T., 2017. Enriching word vectors with subword information. TACL 5, 135–146.

[12] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K., 2015. Apache flink^{TM}: Stream and batch processing in a single engine. IEEE Data Eng. Bull. 38, 28–38.

[13] Carey, M. J., Jacobs, S., Tsotras, V. J., 2016. Breaking bad: A data serving vision for big active data. In: Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems. DEBS '16. ACM, New York, NY, USA, pp. 181–186.
URL `http://doi.acm.org/10.1145/2933267.2933313`

[14] Chen, L., Cong, G., Cao, X., Tan, K., 2015. Temporal spatial-keyword top-k publish/subscribe. In: 2015 IEEE 31st International Conference on Data Engineering. pp. 255–266.

[15] Dean, J., Ghemawat, S., 2008. Mapreduce: Simplified data processing on large clusters. Communications of the ACM 51 (10), 107–113.

[16] Elmasri, R., Navathe, S. B., 2017. Fundamentals of Database Systems 7th edition. PEARSON.

[17] Farzindar, A., Khreich, W., 2015. A survey of techniques for event detection in twitter. Computational Intelligence 31, 132–164.

[18] Grover, R., Carey, M. J., 2014. Asterixdb: A scalable, open source bdms. Proceedings of the VLDB Endowment 7 (17), 1905–1916.

[19] Grover, R., Carey, M. J., 2015. Data ingestion in asterixdb, 605–616.

[20] Guo, J., Fan, Y., Ai, Q., Croft, W. B., 2017. A deep relevance matching model for ad-hoc retrieval. CoRR abs/1711.08611.

[21] Hawkins, D. M., 1980. Identification of outliers. Chapman and Hall.

[22] Høybakken, R., Skarshaug, S., 2018. Filtering and clustering of tweets in asterixdb.

[23] Institute, T. M. G., 2011. Big data: The next frontier for innovation, competition, and productivity. `https://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/big-data-the-next-frontier-for-innovation`, online; accessed 29.10.2018.

[24] Kanhabua, N., Blanco, R., Nørvåg, K., et al., 2015. Temporal information retrieval. Foundations and Trends® in Information Retrieval 9 (2), 91–208.

[25] Kim, Y., 2014. Convolutional neural networks for sentence classification. CoRR vol. abs/1408.5882.

[26] Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J., Ramasamy, K., Taneja, S., 2015. Twitter heron: Stream processing at scale. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 239–250.

[27] Le, Q. V., Mikolov, T., 2014. Distributed representations of sentences and documents. CoRR abs/1405.4053.

[28] Leskovec, J., Rajaraman, A., Ullman, J. D., 2014. Mining of Massive Datasets. Cambridge University Press.

[29] Liang, S., Ren, Z., Weerkamp, W., Meij, E., de Rijke, M., 2014. Time-aware rank aggregation for microblog search. In: CIKM.

[30] Liu, T.-Y., Mar. 2009. Learning to rank for information retrieval. Found. Trends Inf. Retr. 3 (3), 225–331.
URL `http://dx.doi.org/10.1561/1500000016`

[31] Manning, C. D., Raghavan, P., Schtze, H., 2008. Introduction to Information Retrieval. Cambridge University Press.

[32] Middleton, S. E., Middleton, L., Modafferi, S., Mar 2014. Real-time crisis mapping of natural disasters using social media. IEEE Intelligent Systems 29 (2), 9–17.

[33] Mikolov, T., Corrado, G., Chen, K., Dean, J., 2013. Efficient estimation of word representations in vector space. CoRR abs/1301.3781, 1–12.

[34] Mouratidis, K., Bakiras, S., Papadias, D., 2006. Continuous monitoring of top-k queries over sliding windows. In: Proceedings of the 2006 ACM SIGMOD international conference on Management of data. ACM, pp. 635–646.

[35] Mouratidis, K., Pang, H., 2011. Efficient evaluation of continuous text search queries. IEEE Transactions on Knowledge and Data Engineering 23 (10), 1469–1482.

[36] Naveed, N., Gottron, T., Kunegis, J., Che Alhadi, A., 10 2011. Searching microblogs: Coping with sparsity and document quality. pp. 183–188.

[37] Nguyen, H., Woon, Y., Ng, W. K., 2014. A survey on data stream clustering and classification. Knowledge and Information Systems 45.

[38] Norrhall, S. P. K., 2018. Continuous queries on streaming data. Master's thesis, NTNU.

[39] Oates, B., 2005. Researching Information Systems and Computing. Sage.

[40] Ozdikis, O., Karagoz, P., Oğuztüzün, H., 12 2017. Incremental clustering with vector expansion for online event detection in microblogs. Social Network Analysis and Mining 7.

[41] Pääkkönen, P., 2016. Feasibility analysis of asterixdb and spark streaming with cassandra for stream based processing. Journal of Big Data 3 (1), 6.

[42] Pang, L., Lan, Y., Guo, J., Xu, J., Xu, J., Cheng, X., 2017. Deeprank: A new deep architecture for relevance ranking in information retrieval. CoRR abs/1710.05649.

[43] Park, J., Hong, B., Ban, C., 2008. A query index for continuous queries on rfid streaming data. Science in China Series F: Information Sciences 51 (12), 2047–2061.

[44] Pennington, J., Socher, R., Manning, C. D., 2014. Glove: Global vectors for word representation. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, 1532–1543.

[45] Petrovic, S., Osborne, M., Lavrenko, V., 2010. Streaming first story detection with application to twitter. Computational Linguistics (June), 181–189.

[46] Pripužić, K., Žarko, I., Aberer, K., 01 2014. Top-k/w publish/subscribe: A publish/subscribe model for continuous top-k processing over data streams. Information Systems 39, 256–276.

[47] Repp, Ø., Ramampiaro, H., 2018. Extracting news events from microblogs. Journal of Statistics and Management Systems 21 (3), 694–723.

[48] Salton, G., Wong, A., Yang, C. S., Nov. 1975. A vector space model for automatic indexing. Commun. ACM 18 (11), 613–620.
URL http://doi.acm.org/10.1145/361219.361220

[49] Schilder, F., Habel, C., 2003. Temporal information extraction for temporal question answering. In: New Directions in Question Answering. pp. 35–44.

[50] Shou, L., Wang, Z., Chen, K., Chen, G., 2013. Sumblr: continuous summarization of evolving tweet streams. In: SIGIR.

[51] Spangenberg, M., Roth, M., Franczyk, B., 2015. Evaluating new approaches of big data analytics frameworks. Business Information Systems, 28–37.

[52] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., Ryaboy, D., 2014. Storm@twitter. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. SIGMOD '14. ACM, pp. 147–156.

[53] Twitter, 2016. Open sourcing twitter heron. https://blog.twitter.com/engineering/en_us/topics/open-source/2016/open-sourcing-twitter-heron.html, online; accessed 07.12.2018.

[54] Vouzoukidou, D., 2015. Continuous top-k queries over real-time web streams. Ph.D. thesis, Université Pierre et Marie Curie - Paris VI.

[55] Wang, H., Can, D., Kazemzadeh, A., Bar, F., Narayanan, S., 2012. A system for real-time twitter sentiment analysis of 2012 us presidential election cycle. In: Proceedings of the ACL 2012 System Demonstrations. Association for Computational Linguistics, pp. 115–120.

[56] Wang, X., Carey, M. J., 2019. An idea: An ingestion framework for data enrichment in asterixdb. arXiv preprint arXiv:1902.08271.

[57] Yin, J., Lampert, A., Cameron, M., Robinson, B., Power, R., 2012. Using social media to enhance emergency situation awareness. IEEE Intelligent Systems 27 (6), 52–59.

[58] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I., Oct. 2016. Apache spark: A unified engine for big data processing. Commun. ACM 59 (11), 56–65.
URL http://doi.acm.org/10.1145/2934664

[59] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., et al., 2016. Apache spark: a unified engine for big data processing. Communications of the ACM 59 (11), 56–65.

[60] Zhang, C., Lei, D., Yuan, Q., Zhuang, H., Kaplan, L., Wang, S., Han, J., Jan. 2018. Geoburst+: Effective and real-time local event detection in geo-tagged tweet streams. ACM Trans. Intell. Syst. Technol. 9 (3), 34:1–34:24.
URL `http://doi.acm.org/10.1145/3066166`

[61] Zhang, C., Zhou, G., Yuan, Q., Zhuang, H., Zheng, Y., Kaplan, L., Wang, S., Han, J., 2016. Geoburst: Real-time local event detection in geo-tagged tweet streams. In: Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval. SIGIR '16. ACM, New York, NY, USA, pp. 513–522.
URL `http://doi.acm.org/10.1145/2911451.2911519`

[62] Zhang, J., Mouratidis, K., Li, Y., et al., 2017. Continuous top-k monitoring on document streams. IEEE Transactions on Knowledge and Data Engineering 29 (5), 991–1003.

[63] Zhou, Y., Kanhabua, N., Cristea, A. I., 2016. Real-time timeline summarisation for high-impact events in twitter. In: Proceedings of the Twenty-second European Conference on Artificial Intelligence. IOS Press, pp. 1158–1166.

[64] Zhu, R., Wang, B., Yang, X., Zheng, B., Wang, G., 2017. Sap: Improving continuous top-k queries over streaming data. IEEE Transactions on Knowledge and Data Engineering 29 (6), 1310–1328.

[65] Zicar, R. V., 2014. Asterixdb: Better than hadoop? interview with mike carey. `http://www.odbms.org/blog/2014/10/asterixdb-hadoop-interview-mike-carey/`, online; accessed 07.12.2018.

Appendices

Appendix A

Listings

```java
public class RelevanceDetecterFunction implements IExternalScalarFunction {
    private List<String> functionParameters = new ArrayList<>();
    private Query query;
    private Clustering clustering;

    public void initialize(IFunctionHelper iFunctionHelper) {
        functionParameters = iFunctionHelper.getParameters();
        query = createQuery();
        clustering = createClusteringFromModel();
        API.clearDataset();
    }

    private Query createQuery() {
        try {
            String response = API.getQuery();
            return new Query(response.toLowerCase(), LocalDateTime.now());
        } catch (Exception e) {
            throw new IllegalStateException("Could not set query from API.");
        }
    }

    private Clustering createClusteringFromModel() {
        if (functionParameters.size() == 0) {
            throw new IllegalArgumentException("Expected a parameter, got 0.");
        }

        File model = new File(functionParameters.get(0));
        Word2Vec w2vModel = WordVectorSerializer.readWord2VecModel(model, false);
        return new Clustering(w2vModel);
    }

    public void evaluate(IFunctionHelper functionHelper) throws Exception {
        JString text = (JString) functionHelper.getArgument(0);
        Tweet tweet = new Tweet(text.getValue().toLowerCase());
        JBoolean output = (JBoolean) functionHelper.getResultObject();

        int commonTerms = Score.commonTermsScore(query, tweet);
        boolean shouldUpdateStorage = false;

        if (commonTerms >= 1) {
            Integer threadId = clustering.clusterTweet(tweet.getContent());
            if (threadId != null) {
                double score = Score.queryThreadSim(query, clustering, threadId);
                shouldUpdateStorage = Ranking.update(clustering.getThread(threadId), score);
            }
        }

        output.setValue(shouldUpdateStorage);
        functionHelper.setResult(output);
    }
}
```

Listing A.1: Java UDF implementation of query predicate.

```java
public class API {
    private static class ApiRequest {
        private final HttpPost request = new HttpPost("http://localhost:19002/query/service");
        private final HttpClient httpClient = HttpClientBuilder.create().build();
        private final String successMessage;
        private final boolean asJson;

        private ApiRequest(String payload, String successMessage, boolean asJson) {
            this.successMessage = successMessage;
            this.asJson = asJson;
            request.setEntity(new StringEntity(payload, ContentType.APPLICATION_FORM_URLENCODED));
        }

        private String execute() {
            try {
                long start = System.currentTimeMillis();
                HttpResponse response = httpClient.execute(request);
                if (asJson) {
                    JSONObject jsonObject = new JSONObject(EntityUtils.toString(response.getEntity()));
                    return jsonObject.get("results").toString().replace("[\"", "").replace("\"]", "");
                } else {
                    return response.toString();
                }
            } catch (IOException ex) {
                ex.printStackTrace();
                return null;
            }
        }
    }

    public static String getQuery() {
        String result = new ApiRequest(
                "statement=select value q.query from relevance.UserQueries q WHERE q.id=1;",
                "Finished retrieving user query",
                true
        ).execute();

        if (result != null) { return result; } else { return ""; }
    }
}
```

Listing A.2: The implemented API class which is responsible for communicating with AsterixDB's.

```java
public class Thread {
    private List<String> tweets = new ArrayList<>();
    private INDArray centroid;
    private int id;

    public Thread(int id) { this.id = id; }

    public int getId() { return id; }

    public List<String> getTweets() { return tweets; }

    public void addTweet(String tweet) { tweets.add(tweet); }

    public void setCentroid(INDArray centroid) { this.centroid = centroid; }

    public INDArray getCentroid() { return centroid; }

    @Override
    public String toString() { return "Thread{id=" + id + '}'; }
}
```

Listing A.3: The class representing a Thread.

```java
private void setCentroid(int threadid, Word2Vec vector) {
    Thread thread = getThread(threadid);

    if (thread.getTweets().isEmpty()) {
        thread.setCentroid(vector.getWordVectorsMean(Arrays.asList(" ")));
    } else {
        LOGGER.info("Thread was not empty, calculating mean vector for thread..");
        String threadWords = String.join(" ", thread.getTweets());
        Collection<String> labels = Splitter.on(' ').splitToList(threadWords);
        thread.setCentroid(vector.getWordVectorsMean(labels));
    }
}
```

Listing A.4: Setting a Thread's centroid based on the tweets within it.

Appendix B

Figures

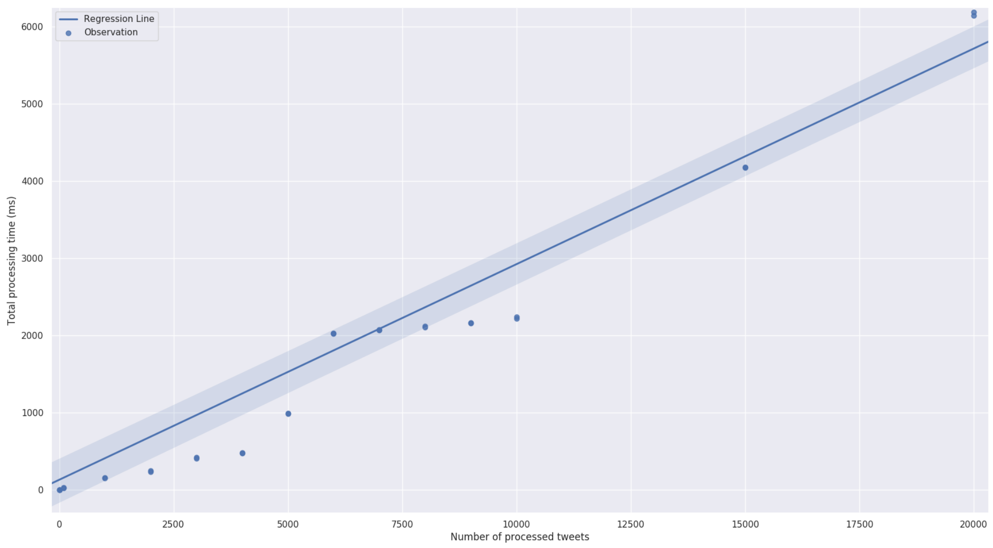

Figure B.1: The total time spent processing tweets found from performing two runs of the experiment, zoomed in on the lower x values.

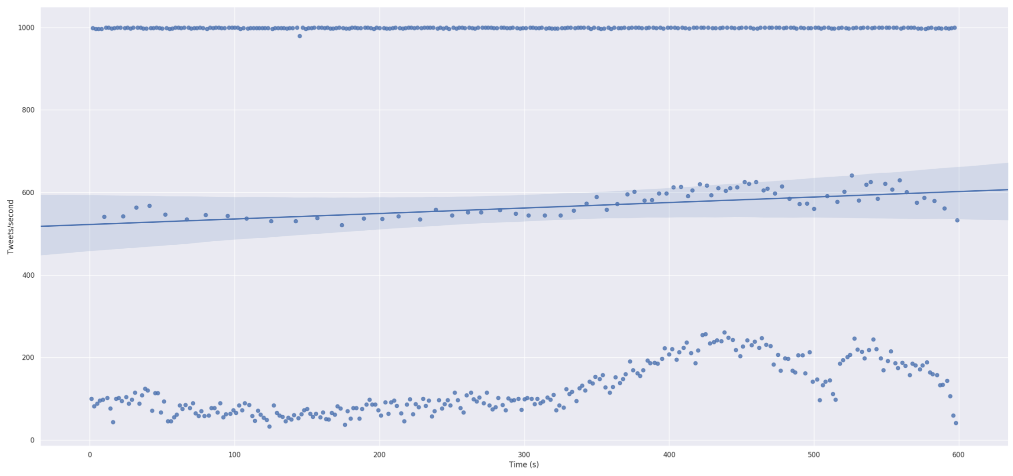

Figure B.2: The number of tweets actually processed each second when the arrival-rate was set to **1000** tweets/second in the first run, using w = 2000.

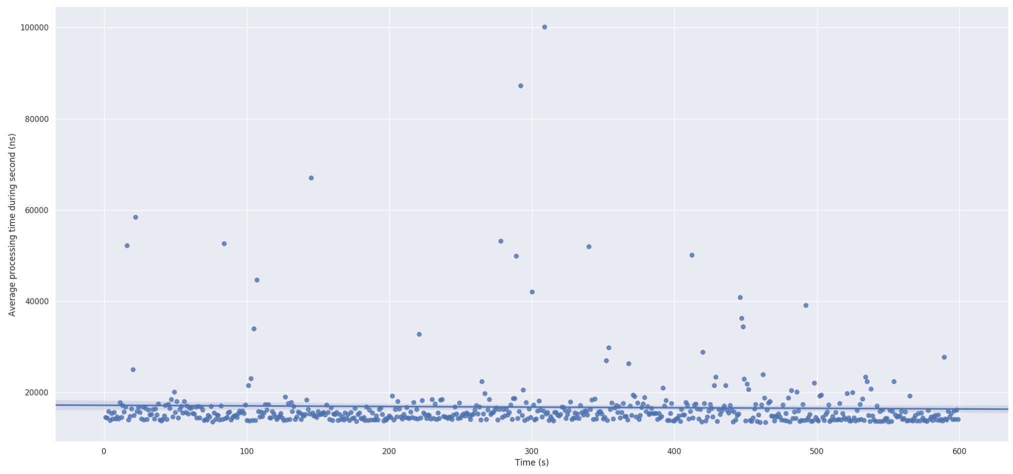

Figure B.3: First run: Average processing time of tweets during each second while sending **1000** tweets/second.

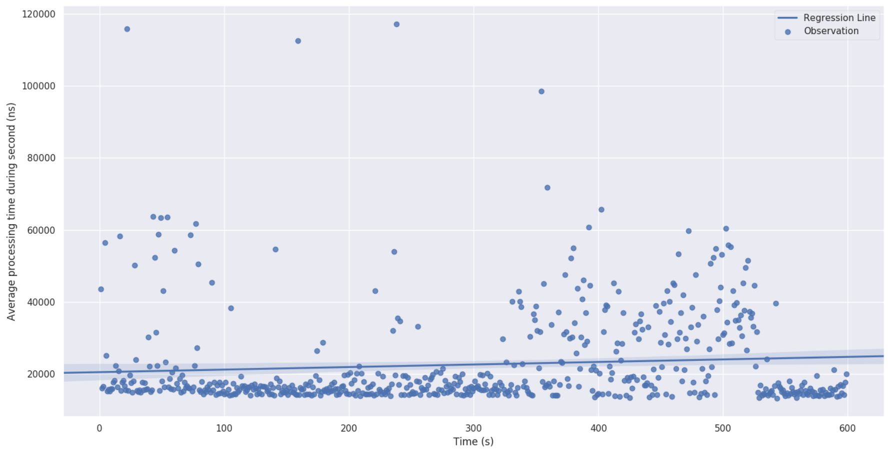

Figure B.4: Second run: Average processing time of tweets during each second while sending **1000** tweets/second.

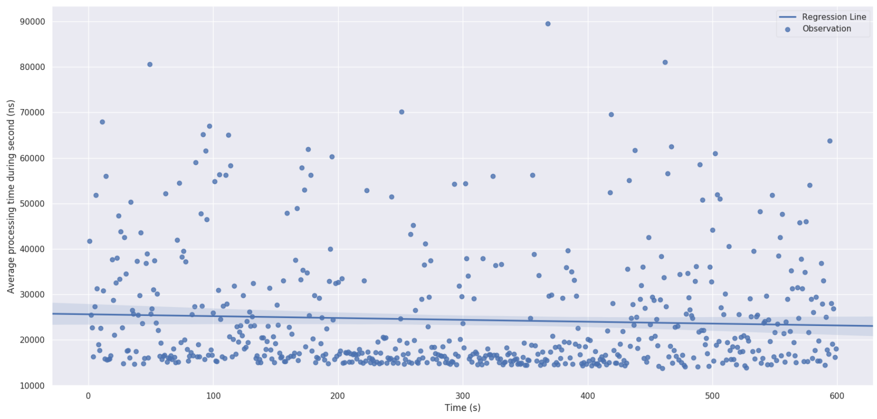

Figure B.5: Third run: Average processing time of tweets during each second while sending **1000** tweets/second.

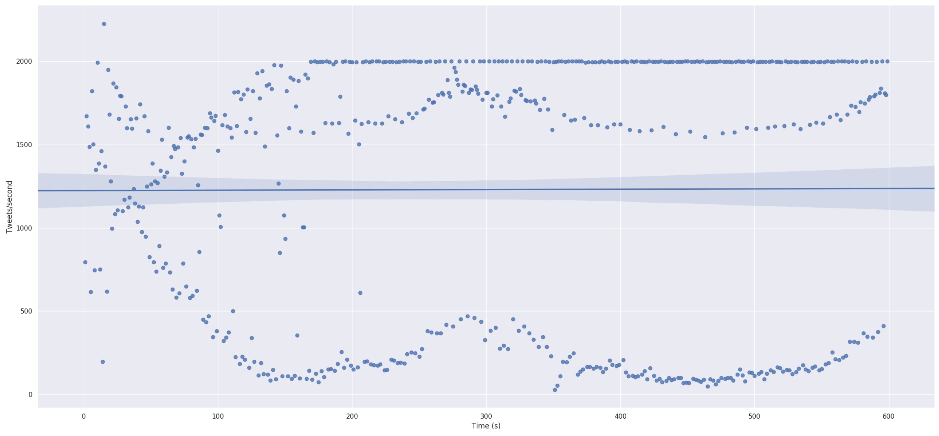

Figure B.6: The number of tweets actually processed each second when the arrival-rate was set to **2000** tweets/second in the first run, using w = 2000.

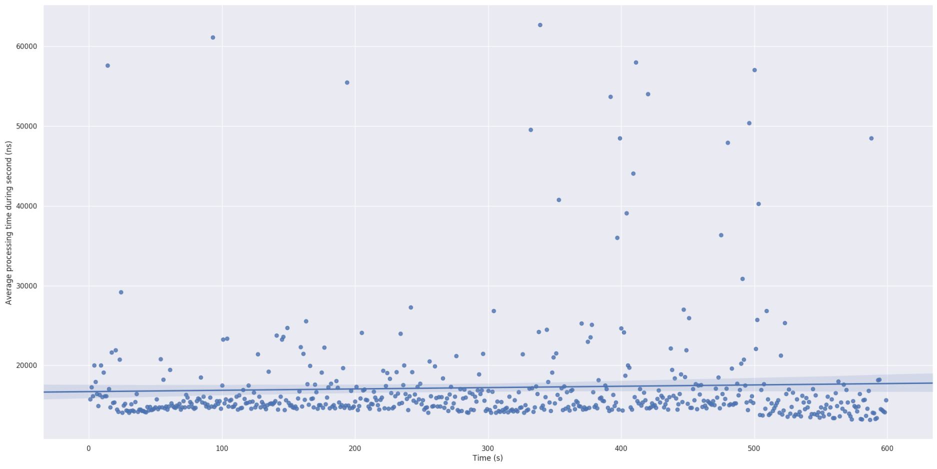

Figure B.7: First run: Average processing time of tweets during each second while sending **2000** tweets/second.

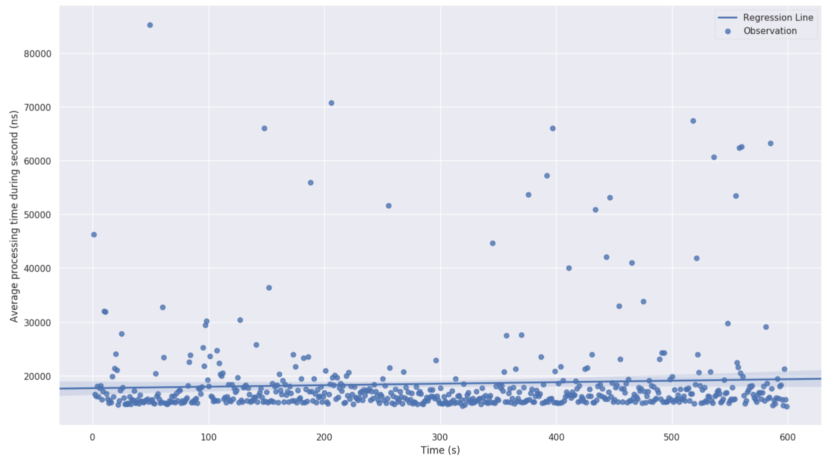

Figure B.8: Second run: Average processing time of tweets during each second while sending **2000** tweets/second.

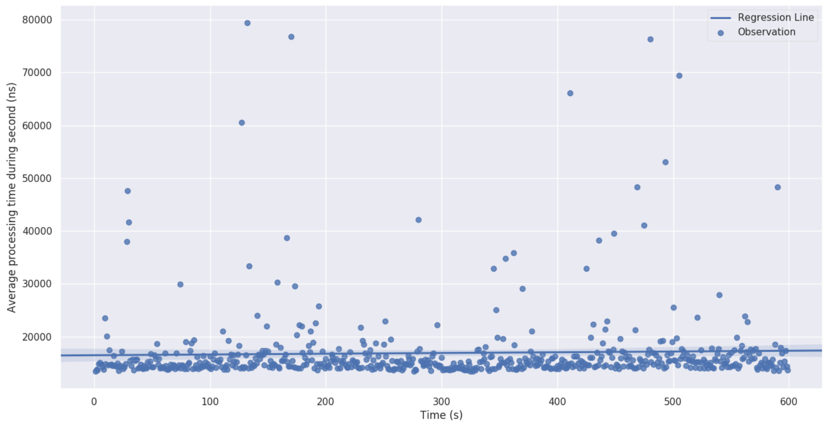

Figure B.9: Third run: Average processing time of tweets during each second while sending **2000** tweets/second.

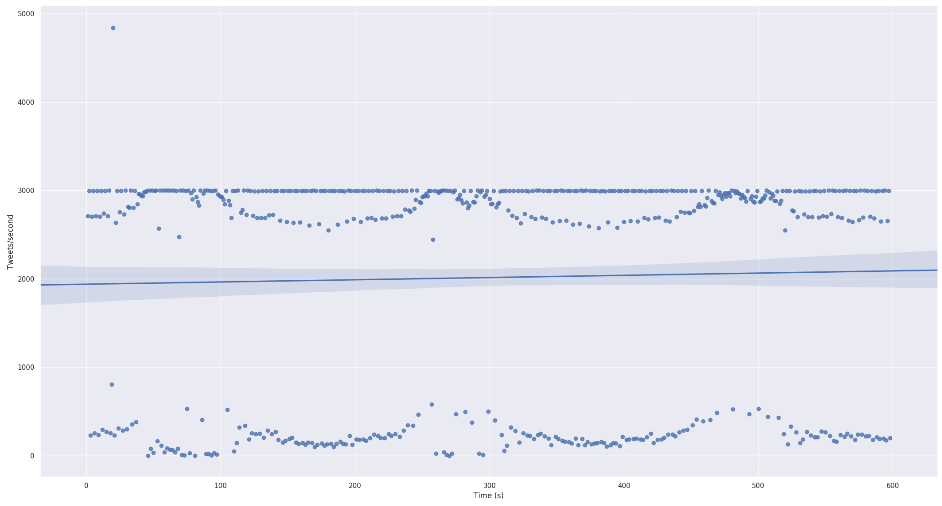

Figure B.10: The number of tweets actually processed each second when the arrival-rate was set to **3000** tweets/second in the first run, using w = 2000.

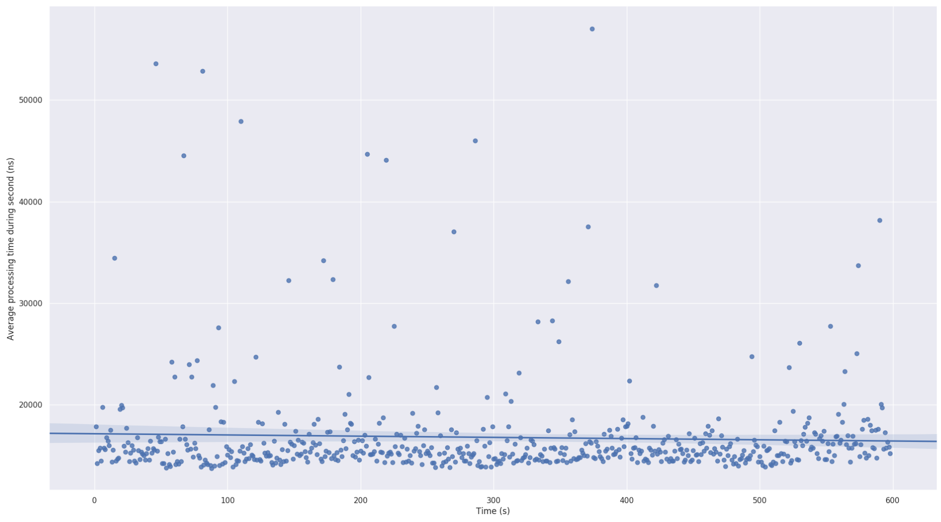

Figure B.11: First run: Average processing time of tweets during each second while sending **3000** tweets/second.

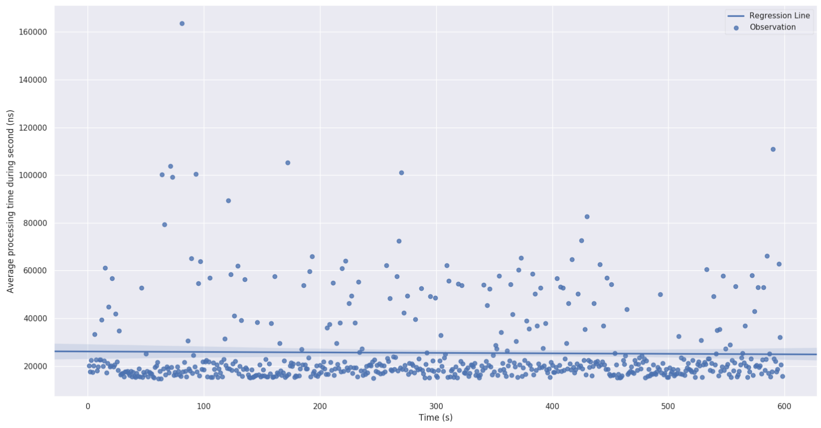

Figure B.12: Second run: Average processing time of tweets during each second while sending **3000** tweets/second.

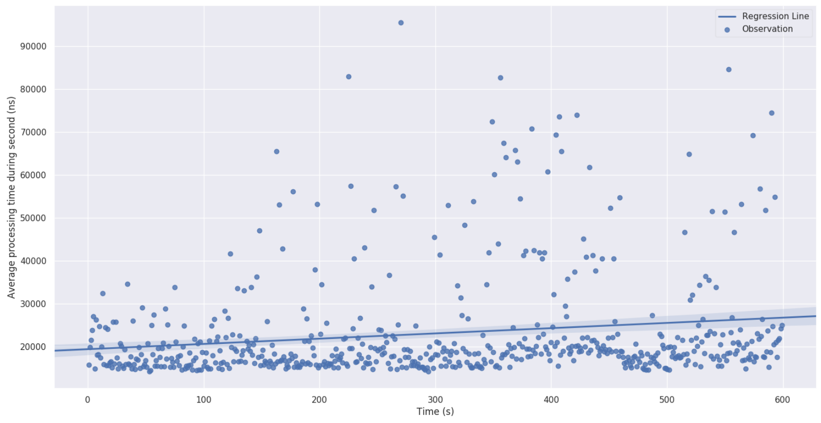

Figure B.13: Third run: Average processing time of tweets during each second while sending **3000** tweets/second.

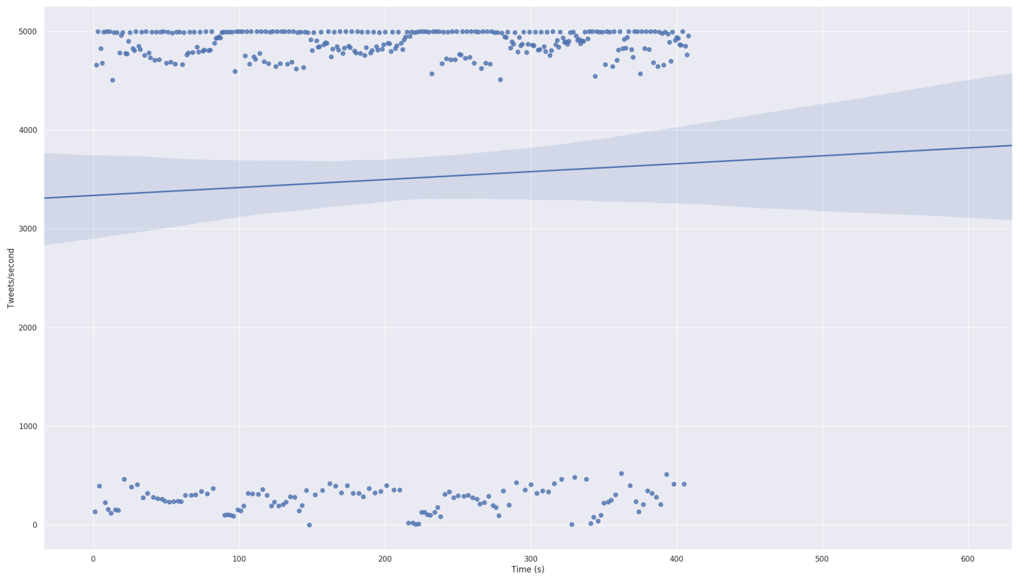

Figure B.14: First run: Actual number of tweets processed pe second when the arrival-rate was set to **5000** tweets/second, using w = 2000.

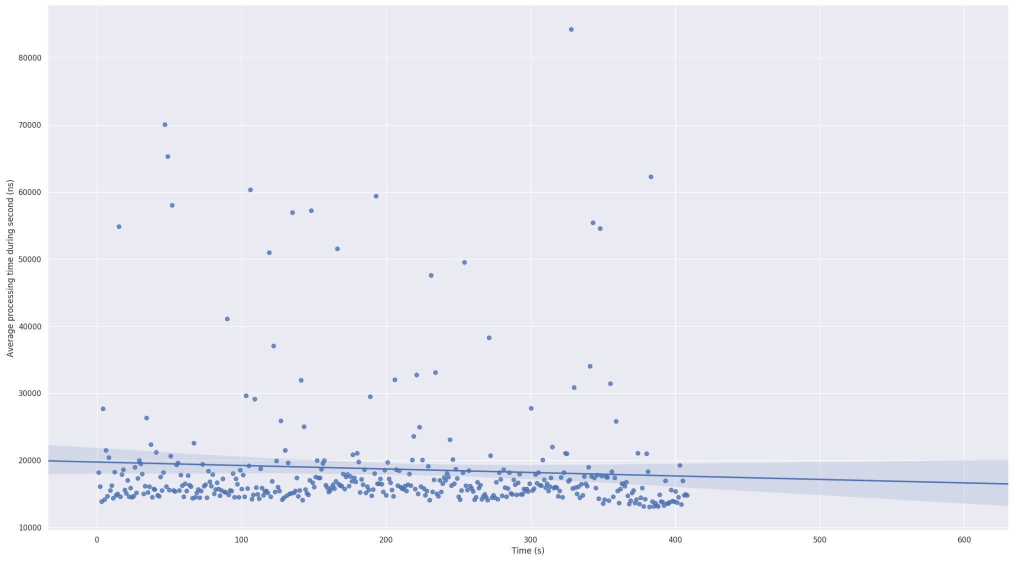

Figure B.15: First run: Average processing time of tweets during each second while processing **5000** tweets/second.

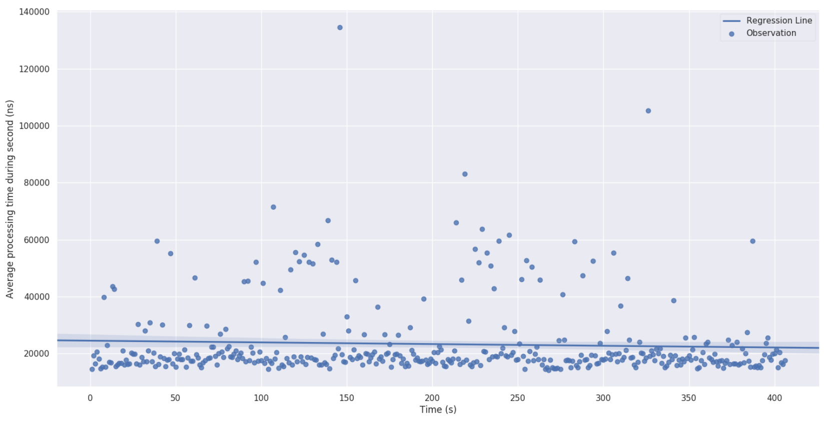

Figure B.16: Second run: Average processing time of tweets during each second while sending **5000** tweets/second.

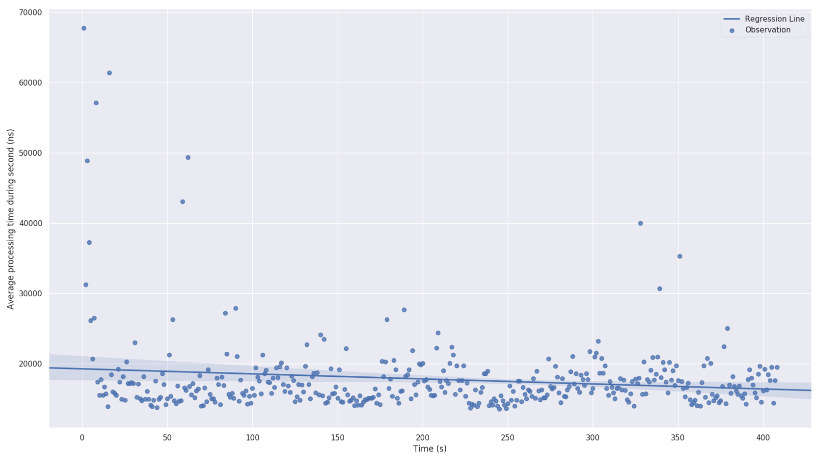

Figure B.17: Third run: Average processing time of tweets during each second while sending **5000** tweets/second.

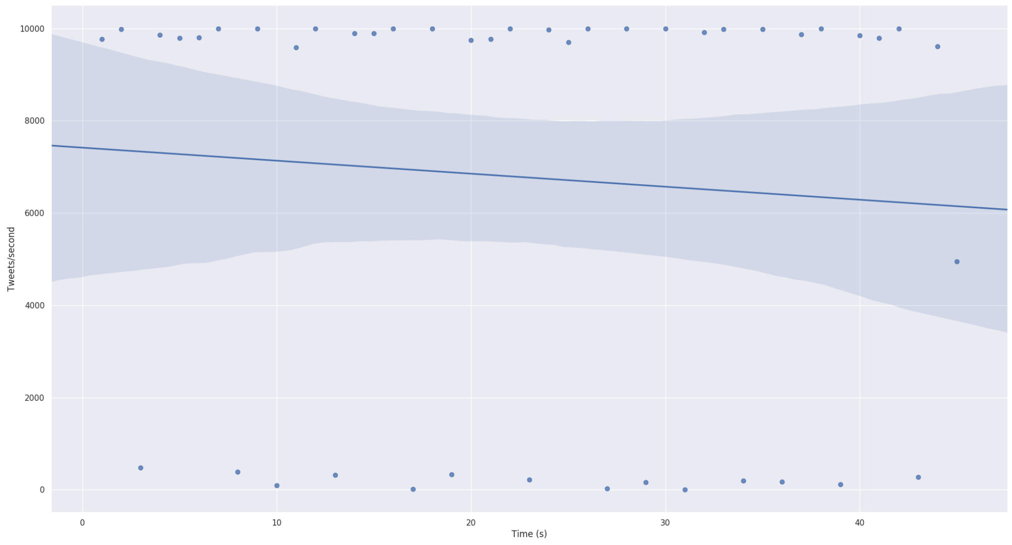

Figure B.18: The number of tweets actually processed each second when the arrival-rate was set to **10 000** tweets/second in the first run, using w = 2000.

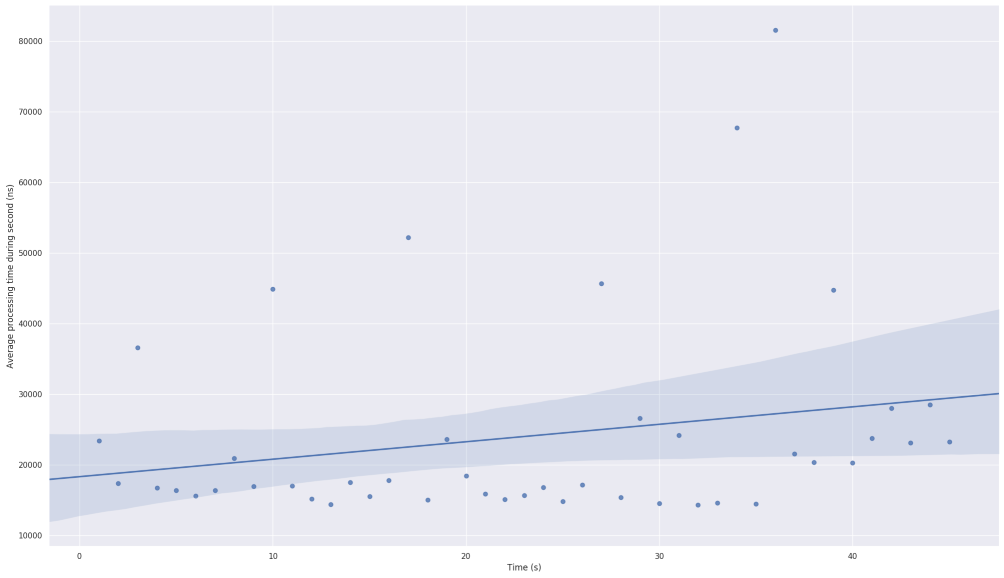

Figure B.19: First run: Average processing time of tweets during each second while processing **10 000** tweets/second.

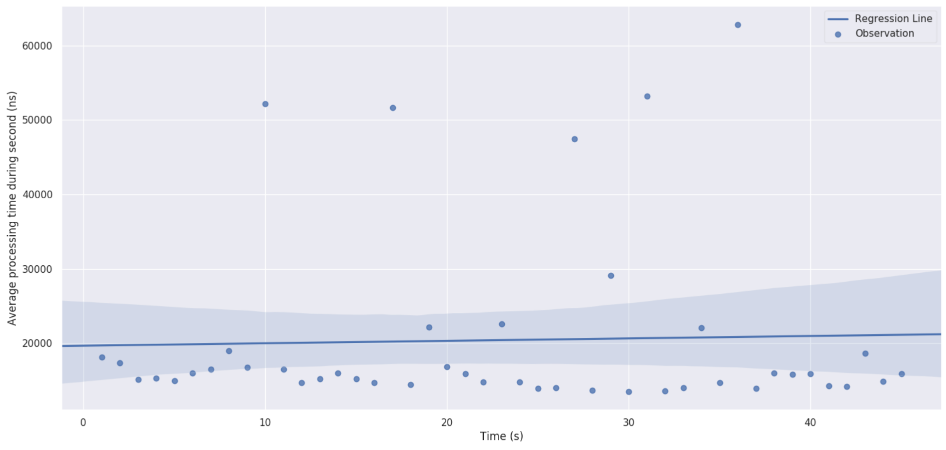

Figure B.20: Second run: Average processing time of tweets during each second while sending **10 000** tweets/second.

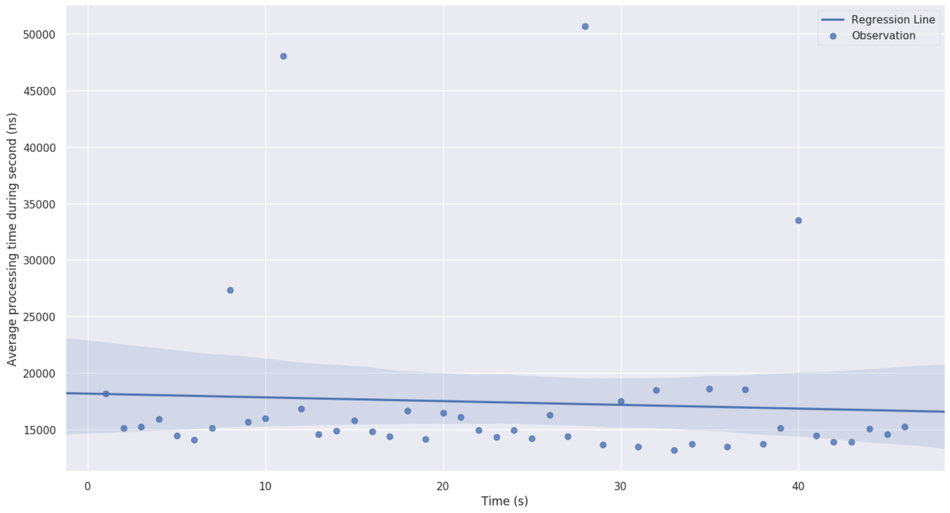

Figure B.21: Third run: Average processing time of tweets during each second while sending **10 000** tweets/second.

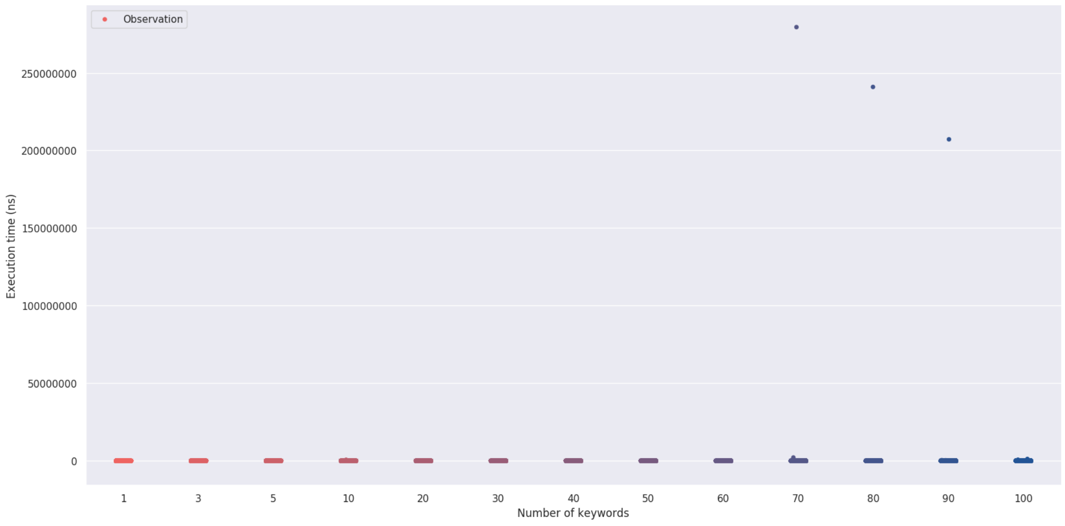

Figure B.22: The execution time of the filtering function for 10 000 observations per keyword.

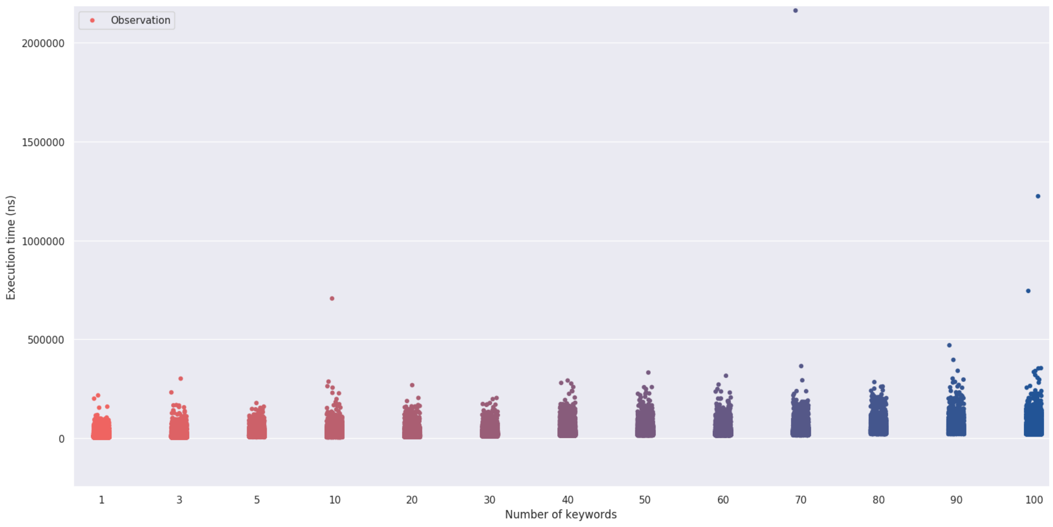

Figure B.23: The execution time of the filtering function for 10 000 observations per keyword, zoomed in on the lower execution times.

Appendix C

Tables

Version	**Solves**	**Retrieved**
First version	The new ingestion framework, makes it possible to see changes in referenced data	7 March 2019
Second version	Bug fix for UDF data type interference system	8 April 2019
Third version	Bug fix for halting queries due to a compilation issue related to the MISS-ING data type	30 April 19
Fourth version	Bug fix for runtime parallelism	3 May 2019
Fifth version	Parsing issue when Twitter API used together with Java UDF	24 May 2019

Table C.1: Different versions of AsterixDB with the new, decoupled feed framework which have been retrieved during this study.

Average throughput:	562.15 tweets/second
Average processing time	0.0168 ms

Table C.2: Results from sending **1000** tweets/second in the first run.

Average throughput:	1229.794 tweets/second
Average processing time	0,017210935 ms

Table C.3: Results from sending **2000** tweets/second in first run.

Average throughput:	2014.3277 tweets/second
Average processing time	0.01679 ms

Table C.4: Results from sending **3000** tweets/second.

Average throughput:	3501,0465 tweets/second
Average processing time	0.018704 ms

Table C.5: Results from sending **5000** tweets/second.

Average throughput:	6767.7555 tweets/second
Average processing time	0.024026 ms

Table C.6: Results from sending **10 000** tweets/second.