NTNU

Norwegian University of
Science and Technology

Scaling Machine Learning Methods to Big Data Systems

Thor Martin Abrahamsen

Master of Science in Computer Science
Submission date: June 2017
Supervisor: Herindrasana Ramampiaro, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

As the world is becoming more digital, an increasing amount of data is generated that could provide industries with valuable insight to make their products or services better for customers as well as employees. To unlock this value, the data is captured and stored, and Machine Learning techniques are applied to extract the knowledge. With the emergence of Internet of Things, increasing quantities of data are available in *real-time*. The industry is missing a unified tool to exploit the relationship of historical and real-time data, and as a consequence, systems have been "glued" together for the two separate tasks. To address the issues of glueing multiple systems together, in this project we investigate how we can implement existing Machine Learning frameworks, such as WEKA, with existing Big Data Management System, e.g. AsterixDB. The main goal is to enable Machine Learning and Data Mining methods on a large scale, both for historical, stored data, and for streaming, real-time data. This unlocks the power to view historical data and real-time data in context of each other, and to continuously improve the Machine Learning techniques by training on the accumulated new arrival of data, and have the algorithms adapt as the data evolves. Experiments with 100 million records show that this implementation scales gracefully for historical data, and at the same time imposes minimal overhead to AsterixDB when processing streaming data. The solution achieves a processing throughput of up to 10'000 records a second, and scales well with the distribution of the cluster.

Sammendrag

I takt med at verden digitaliseres genereres også en økende mengde data som kan gi næringer og virksomheter verdifull innsikt som vil gjøre produktene og tjenestene deres bedre for både forbrukere og ansatte. For å kunne nyttiggjøre seg av denne verdien blir dataene fanget opp og lagret og maskinlæringsteknikker tas i bruk for å hente ut kunnskap. Med fremveksten av Tingenes Internett blir stadig mer data også tilgjengelig i sanntid. Bransjen mangler et helhetlig verktøy for å utnytte forholdet mellom historiske data og sanntidsdata og som en konsekvens av dette har systemer for de to ulike oppgavene blitt «limt» sammen. For å adressere problemet med å lime sammen flere systemer, undersøker vi i dette prosjektet hvordan vi kan integrere eksisterende maskinlæringsrammeverk, som WEKA, i eksisterende BigData systemer, som AsterixDB, for å muliggjøre maskinlæring og datagruvedrift i stor skala, både for historiske lagrede data og for strømmende sanntidsdata. Dette gjør det mulig å se historiske data og sanntidsdata i kontekst av hverandre og kontinuerlig forbedre maskinlæringsteknikker ved å trene modeller med nye data og la algoritmene tilpasse seg etterhvert som dataene utvikler seg. Eksperimenter med 100 millioner poster viser at denne implementasjonen skalerer godt for historiske data, samtidig og medfører minimal belastning på AsterixDB når datastrømmer håndteres. Løsningen oppnår en prosesseringskapasitet på opptil 10'000 poster i sekundet, og skalerer godt med antall noder i systemet.

Preface

This thesis is written by Thor Martin Abrahamsen as part of the Computer Science education at Norwegian University of Science and Technology (NTNU) in Trondheim. It is the final requirement for a degree in Master of Science (MSc) with specialization in Databases and Search, and was conducted during the spring semester of 2017.

Acknowledgements

The idea of the project was brought up by associate professor Heri Ramampiaro at the Department of Computer Science (IDI). He has been my supervisor for this project and I would like to express my sincere gratitude for his guidance throughout the last year. Allowing me to work independently, yet steering me in the right direction whenever I lost track. I would also like to thank the AsterixDB community for quickly and thoroughly responding to my inquiries, especially Xikui Wang, PhD candidate at University of California, Irvine, which went beyond expectations to provide me with the help I needed. Finally I would like to thank my partner, family and friends for feedback and input on this report and emotional support.

Thor Martin Abrahamsen
Trondheim, 10th June 2017

Contents

List of Figures

Listings

Acronyms

ADM Asterix Data Model. xi, 7, 9, 17

API Application Programming Interface. xi, 8, 11, 15, 23

AQL Asterix Query Language. xi, 7, 9, 17, 27, 28

BAD Big Active Data. xi, 14

BDMS Big Data Management System. i, xi, 7, 14, 41

CC Cluster Controller. xi, 7, 18

CFM Central Feed Manager. xi, 18

DM Data Mining. xi, 10, 11, 13

DSMS Data Stream Management System. xi, 14

FM Feed Manager. xi, 18

IoT Internet of Things. i, xi

ML Machine Learning. i, xi, 3–5, 7, 10, 11, 13, 15, 19, 23, 25, 41, 42

MLDM Machine Learning and Data Mining. i, xi, 3, 13

NC Node Controller. xi, 7, 18

RDD Resilient Distributed Dataset. xi, 8, 10

RS Recommender System. xi, 13

SPE Stream Processing Engine. xi, 14

UDF User Defined Function. xi, 7, 9, 10, 18–20, 23, 26, 37, 41

WEKA Waikato Environment for Knowledge Analysis. xi, 10

1 Introduction

This chapter first describes the background and motivation for this project, then defines the primary goal and associated research questions. The approach taken will be described and the contributions identified, before presenting an outline of the following chapters.

1.1 Background and Motivation

Today data is generated from everything. Every website you visit, every post you share in social media, every step you take with an activity tracker, every time you swipe your access card, shop on the Internet, or when you leave a blood sample at the hospital, data are gathered and stored. We know this data is valuable, and we are getting skilled at storing this amount of *Big Data*. Still, the potential hidden in this data is just barely explored (Elgendy and Elragal, 2014). A large portion of this data remains unexploited.

We are becoming quite sophisticated at extracting unknown information from data with the help of Machine Learning (ML). These approaches learn and train them selves to classify, find outliers and patterns, predict future outcome, and make suggestions.

Businesses and corporations have started to use these methods of ML on the increasing amount of Big Data, but common for a majority of them is that the tools are specialized for solving a specific purpose, or to work within a specific domain (Low et al., 2012). This motivates the implementation of a general platform for performing ML algorithms on Big Data.

In the jungle of frameworks and tools for Machine Learning and Data Mining (MLDM) on Big Data, there already exists a few approaches that have been adopted by the industry and has become *de facto* tools. However, as the world evolves new requirements arise and these tools struggle to keep up. Among the challenges are the need to support real-time queries with advanced capabilities. Specialized tools have emerged to cope with these challenges, but there remains a gap between processing of real-time data and stored data.

1.2 Goals and Research Questions

The main motivation of this project is to fill the gap between ML platforms designed for stored Big Data and those designed for streaming Big Data. The primary question we seek an answer for can be formulated as follows:

Can Machine Learning scale to both stored Big Data and streaming Big Data?

Based on this main research question, three subquestions can be derived to guide the research of the project, and define its scope.

Research Questions:

1. *Which Big Data platform can handle streaming data, and support parallel execution of user code?*
2. *How can Machine Learning be implemented into such a platform?*
3. *How does the implementation scale with increasing amount of data?*

1.3 Approach

1.3.1 Research Method

This study started with a literature review of existing systems and research for Big Data and methods of streaming data, to examine current *state-of-the-art* systems and approaches, with their possibilities and limitations. Further, the use of ML methods in, or in combination with, such systems were examined. A suitable ML library was identified and implemented in an existing Big Data platform. The solution was tested against a benchmark considered industry standard, and results of the comparative experiments were evaluated.

1.3.2 Scope and Limitations

This research develops a tool to investigate the use of the generic ML framework WEKA in an existing Big Data platform, called AsterixDB. The specific ML task of *sentiment analysis* on Twitter feeds will be conducted to be able to reason about the effect of the implemented methods. Because of resource limitations, experiments in this thesis are performed on a single computer only, simulating cluster nodes with processor cores.

1.4 Contributions

Current *state-of-the-art* approaches is a patchwork of specialized systems, glued together to solve the challenges of ML in real-time on Big Data. These assembled systems make it hard to reason about data consistency, scalability and fault-tolerance. The main contribution of this work is the study of integrating a ML framework with an existing Big Data system. This reduces the number of moving parts needed to handle streaming Big Data.

This research will also conduct a comparative study of the implemented solution to an industry standard of Spark and Cassandra, in terms of their ability to handle ML on large scale streaming data.

1.5 Thesis Structure

The remainder of this report is structured as follows: Chapter 2 describes the concepts of Big Data and ML, and introduces a specific ML task and domain. Related research and frameworks are investigated in Chapter 3. The architecture of AsterixDB data feeds and the implemented solution is presented in Chapter 4, and Chapter 5 summarizes the setup and execution of the experiments performed, along with their results. Chapter 6 gives an evaluation and discussion of the results and implementation, while Chapter 7 presents a conclusion and points out directions for future work.

2 Background Theory

This chapter describes the concepts of Big Data, streaming data and Machine Learning (ML), and introduces specific systems for each. Then a specific ML task and domain is presented.

2.1 Big Data

The term *Big Data* occurs in multiple industries with different definitions. One of the most acknowledged definitions is from Douglas (2001), which introduces 3 characteristics: **Volume**, **Velocity** and **Variety**, concerning the scale at which data is produced and stored, the speed of which it is generated and processed and the structure and nature of the data. Today we often include two more V's in the definition: **Variability** and **Value**, concerning changes in the datas structure and what values it brings the organization (Fan and Bifet, 2013).

The increasing value of Big Data and its adoption by companies and organization has inspired the emerge of different tools and frameworks to work with and manage Big Data. One of these, to which this project puts particular attention, is AsterixDB (Alsubaiee et al., 2014).

2.1.1 AsterixDB

Apache AsterixDB is a scalable, open source Big Data Management System (BDMS). It was initiated by a team of faculty, staff and students at UC Irvine and UC Riverside to fill a need in the market for an open source parallel database system, and is now developed as an Apache project. AsterixDB distinguishes it self from other Big Data analytic platforms by both storing and querying the data, and the projects mantra is *"One size fits a bunch"* (Alsubaiee et al., 2014), meaning that they are aiming for a wider range of use cases than other Big Data technologies address to today.

AsterixDB has a flexible NoSQL style data model based on JSON called Asterix Data Model (ADM), and an expressive and declarative query language based on SQL called Asterix Query Language (AQL). Its shared-nothing architecture involves a central Cluster Controller (CC) and a set of worker nodes referred to as Node Controllers (NCs).

It has built-in mechanisms to handle streaming data called Data Feeds, which allows continuous ingestion into datasets from external sources, and the feature User Defined Function (UDF) allows users to define their own processing of the data. This can be defined either alongside the query with AQL or as a library written in Java. The UDF is

then invoked as a function with the record to process as parameter (Grover and Carey, 2015).

2.1.2 Spark

Apache Spark is another tool for processing large-scale data. It started as a research project at UC Berkeley with the goal of designing a programming model to support iterative algorithms, interactive data mining and streaming applications. Spark offers an abstraction called Resilient Distributed Datasets (RDDs) which are datasets stored in memory between queries (Zaharia et al., 2012). These are exposed through Application Programming Interfaces (APIs) in Java, Scala and Python, and are instantiated by developers through transformations (e.g., map and filter). Actions are then applied to the RDD in order to return results (e.g., count, collect, save). RDDs are fault-tolerant without replication, because each RDD remembers how it was built to rebuild itself. Spark computes RDDs lazily and the optimal pipeline of the transformations applied can be found.

Spark is strictly a processing engine, and does not persist data for use outside of the current session. To be able to compare Spark to AsterixDB the data would need a separate system for offline persistence. A candidate for achieving this is *Cassandra*, described next.

2.1.3 Cassandra

Apache Cassandra is a distributed storage system for large amounts of data (Laksham and Prashant, 2010). It was developed at Facebook and designed to handle high write throughput. Benchmarking with other key-value stores shows that it scales linearly, and achieve the highest throughput of the systems tested (Rabl and Gómez-Villamor, 2012). This result makes Cassandra a good candidate for persistence after spark.

The Spark Cassandra connector[1] provides easy database access to Cassandra from Spark in Java.

2.2 Streaming Data

Streaming data is conceptually different from traditional stored data, and requires new models and algorithms to interact with. Streaming data arrives in tuples, often in such a rate that it is impractical to store the entire stream. This means that arriving data must be processed immediately or it is lost forever. Processing capacity is constrained by available physical storage (e.g., main memory) often referred to as the working space of the stream processor. In addition, stream processors may have larger offline storage for archival purposes, with much higher I/O-delay than the frequency of arriving data (Rajaraman and Ullman, 2011).

[1] `https://github.com/datastax/spark-cassandra-connector`

There are two main categories of queries in a streaming system: *standing queries* and *ad-hoc queries*. The former are questions that are asked for every incoming record in the system. Examples include "What is the maximum value seen so far?", or "Notify me when the temperature is above 30 degrees". The latter type of queries are questions asked once about the current state of the stream. Examples are "How many unique users have visited the website the last month?", or "What is the average temperature for the last 12 hours?". This requires that the stream processor keeps some state of previously seen records, e.g. the last moth of users or the last 12 hours of the temperature.

One approach for keeping a state of the stream is *sliding windows*. This means that the stream processor keeps the recent n records of the stream, or records arrived within the last t time units, in working space dropping any older records. To be of any value the stream processor must have a notion of what kind of ad-hoc queries it needs to handle beforehand. If the rate of arriving records is high the window will quickly exhaust the working space. To mitigate this it is common to either keep a sample of the stream or use an estimate to answer the query. This reduces the space requirements for the window and allows for grater values of n and t. An alternative to the sliding window is a *decaying window*, keeping more of the recent elements and less of the older, and a *tumbling window*, isolating sequential series of time or elements.

2.2.1 AsterixDB feeds

The AsterixDB Data Feed is a component to support continuous flow of data from an external source into persistent and indexed storage of AsterixDB (Grover and Carey, 2015). It works by instantiating a *Feed Adapter* using AQL in the AsterixDB interface, and connecting it to a dataset. There are different feed adapters for different datasources, and they are either pull-based or push-based. When started, the feed adapter connects to the data source, receives records, parses and translates them into ADM-objects and stores them incrementally. The feed adapters works as pluggable components in the system and allows for building cascade networks of feeds. The output of one stream can be used as the input to another, meaning that the system can fetch the data once, and perform multiple computations on it.

Data can arrive in the feed faster than what AsterixDB can process, and each feed has an associated *ingestion policy* describing the behavior in such case. The built-in policies are *basic*, *spill*, *discard*, *throttle* and *elastic*. The basic policy buffers any excess records in memory until the system is ready to process them, the spill policy saves excess records to disk while the discard policy simply drops the records altogether. The throttle policy will randomly filter out records to regulate the rate of arrival and the elastic policy will scale the AsterixDB cluster out or in to adapt to the rate of arrival.

When instantiating a feed adapter the user can provide custom code for preprocessing of the data by the means of an UDF. This function will be applied to each record prior to persistence. This allows users to interact with the stream in real-time.

2.2.2 Spark Streaming

The motivation behind Spark Streaming was to reduce costs of fault recovery and slow nodes in traditional distributed steam processing models (Zaharia et al., 2013). The core of Spark Streaming is the processing model called *discretized streams*, or D-Streams. It structures a streaming computation as a series of stateless, deterministic batch computations on small time intervals. Each of these batch computations, or tasks, may run on any node in the cluster, and their state is stored in memory using Sparks fault-tolerant data structure RDD. Each task can be recomputed deterministically and allows for parallel recovery. Spark Streaming is based on the Spark processing engine and consists of a *master*-component, which schedules tasks and track the D-Stream lineage, *worker nodes*, which receives data and execute tasks, and a *client library*, used to send streaming data into the system. Spark Streaming is now included in Spark as a library.

2.3 Machine Learning

Machine Learning (ML) is a field of study including construction of algorithms that can learn from and make predictions on data (Mitchell, 1997). This is used for Data Mining (DM), the process of automatically extracting meaningful information from data (Witten et al., 2011). The purpose of this is to unlock the values mentioned earlier.

ML can be *supervised* or *un-supervised*. With the supervised approach, a classifier is trained with annotated data, meaning that the documents in the training set is given a pre-defined category label. The unsupervised approach does not require this human intervention, and uses other means of classification, like clustering.

2.3.1 Open Source Machine Learning Libraries

There are many ML libraries available today[2]. To be able to use the library for the purpose of this project, two requirements emerge. First, it should be feasible to implement in Java. AsterixDB is based on Java and provides UDF for code written in Java. Second, it should be flexible. The intention of using ML in this setting is to provide general DM on Big Data. The library should not be specialized to one specific ML task.

Waikato Environment for Knowledge Analysis (WEKA): WEKA is a collection of machine learning algorithms for data mining tasks (Hall et al., 2009). It is an open-source project written in Java by a team at the Machine Learning Group at the University of Waikato. WEKA aims to provide a collection of ML algorithms and data preprocessing tools to researchers. It ships with a WEKA-client to explore data *ad hoc*, and an API to extend and use the toolkit in other applications.

[2]Gate, RapidMiner, MOA, Spark, R, KNIME and Hadoop (among others)

2.3.2 Spark Machine Learning Library

Spark MLlib started as a project to make ML more accessible to non-experts (Kraska et al., 2013). It quickly became a part of Spark as Spark lacked a suite of robust and scalable learning algorithms (Meng et al., 2016). Spark is designed with iterative computations in mind and enables efficient implementations of ML algorithms. MLlib ships with common ML algorithms, interfaces for feature transformation and pipeline construction and APIs for Java, Scala and Python.

2.4 Task and Domain

This section outlines a chosen task and domain to conduct the project within.

2.4.1 Sentiment Analysis

Sentiment analysis, or opinion mining, has been chosen as the area to perform DM in this project, specifically with the task of classifying the *polarity* of a given text to be either positive or negative. It is one of the most active research areas in natural language processing and DM, and interest in the subject has spread from computer science into other sciences and industries, which makes it a natural task to apply in this study.

Sentiment analysis is concerned with finding people's opinions towards entities such as products or services (Liu, 2012). The focus of this project is not sentiment analysis. This is only a chosen task of DM and will work as an aid for testing the implemented ML method.

2.4.2 Twitter

Twitter has been chosen as the domain for this project, and the ML methods will be performed on data from the Twitter universe. Twitter is one of the most popular microblogs today, with 313 million active users a month[3]. Each user can *follow* other users, and post *Tweets* for their followers to read. Each Tweet can be at most 140 characters, which makes them short and to the point, and therefore extensively used for sentiment analysis. Twitter makes portions of its data available for consumers, and is ideal for this research.

In the Twitter-domain there exists certain special characters that express informal metadata about the text and authors. These characters are # and @. The former are called a *hashtag*, and often denotes a topic of the tweet. The latter are called a *tag* and is used to mark other users. In addition to these, *links* are often present in Tweets and is identified simply by starting with `www` or `http`.

[3] As of November 30, 2016 according to `https://about.twitter.com/company`

3 Related Work

This chapter will investigate the state-of-the-art approaches towards Machine Learning (ML) and stream processing.

3.1 Related Research and Frameworks

ML techniques on large scale data is not a new phenomenon, and Amatriain (2013) points out Recommender Systems (RSs) as the mainstream application for this type of Data Mining (DM). This is how Spotify makes suggestions on what songs a user would probably enjoy, and LinkedIn recommends people for a user to connect with. Both Spotify and LinkedIn (and many other large scale companies like Facebook, Amazon, Twitter, and Yahoo!)[1] uses the **Hadoop**[2] ecosystem as the underlying platform for distributed computing, with different tools for DM **(SPARK, R, MAHOUT, MOA)**[3].

As the sophistication of Machine Learning and Data Mining (MLDM) techniques escalates, there is an increasing need to execute these algorithms in parallel. Low et al. (2012) identifies that different research groups repeatedly solves this issue for individual domains, and that there is a need for a general high-level abstraction. They focused on graph structured computation in parallel and extended the **GraphLab**[4] framework for this problem.

Research has advanced regarding the volume of Big Data, but the *velocity* of the data is an increasingly urgent issue. Data arrives at high speed and the analysis to be performed on the data are often time sensitive. Twitter experienced in 2013 that the *de facto* analysis platform Hadoop did not scale to meet their time requirements regarding the velocity of data (Mishne et al., 2013). They were building a "related query suggestion engine" to suggest trending queries to users in real-time. This required the system to detect a trending query within 10 minutes, and they started to implement the system using Hadoop. Twitter soon discovered that there were too many bottlenecks keeping Hadoop from fulfilling the time requirement. MapReduce-jobs could take tens of seconds only to start up on a large cluster.

A challenge when working with social media data, like microblogs, is that the queries often requires indexing of both real-time and historic data (Magdy, 2016). One approach to the real-time data problem is *continuous queries*, where a query is registered *a priori*,

[1] `https://wiki.apache.org/hadoop/PoweredBy`
[2] `https://hadoop.apache.org`
[3] Different Data Mining tools: `http://spark.apache.org`, `https://www.r-project.org`, `http://mahout.apache.org`, `http://moa.cms.waikato.ac.nz`
[4] GraphLab: `https://turi.com`

and incoming data is processed upon arrival. The query then returns incremental results as new records arrive in the stream. Several Data Stream Management Systems (DSMSs) and Stream Processing Engines (SPEs) provides this functionality, but they lack indexing and querying of historical data.

Big Data Management Systems (BDMSs) has emerged as a concept to handle the multiple challenges of Big Data. Alsubaiee et al. (2014) describes some of the criterions of a BDMS to include a semistructured data model with support for varying data types, a full query language with support for a range of different queries, an efficient parallel query runtime, automatic indexing, continuous data ingestion and to be scalable with increasing volume of data. **AsterixDB**[5] is categorized as such a system. **MYRIA**[6] is another BDMS with a distributed, shared-nothing architecture. It is also provided as a cloud service *(Big-Data-as-a-Service)*. Myria's goal is for users to upload their data and for the system to help them be self-sufficient data science experts on their data. To the best of our knowledge, Myria has no streaming support.

One promising approach to the problem of combining real-time and historical data is the work done with Big Active Data (BAD) by Carey et al. (2016). They are extending AsterixDB to become a platform for what they call *active* data, concerning the ever changing and arriving data. BAD can be thought of as a descendant of *Event-Condition-Action* rules and triggers. The system they are designing will leverage from BDMS with scalability, parallel data analysis, flexible data model, a declarative query language and include concepts from DSMS for real-time data analysis. They envision a *Publish/Subscribe*-pattern where there is a high number of sources of data, and an even higher number of subscribers who desire notification involving both incoming data of interest and changing state of historical data.

In an attempt to benefit from SPEs and BDMSs, several systems has been "glued" together to get the best of both worlds. Grover and Carey (2015) compared continuous data ingestion in AsterixDB to **Storm**[7] + **MongoDB**[8], a popular approach in the community. Storm is used for streaming and processing of streams, while MongoDB is used for persistence. In their evaluation, Grover and Carey (2015) concluded that AsterixDB outperformed the "glued" solution in terms of both performance and user-experience.

Another combination of systems considered *state-of-the-art* is **Spark Streaming**[9] + **Casandra**[10]. Here, Spark Streaming is a library for Spark used to process streams, and Apache Casandra is used for persistence. Pääkkönen (2016) compared this combination to AsterixDB for stream processing of semi-structured data and concluded with AsterixDB achieving higher throughput and lower latency than the "glued" solution.

[5] `https://asterixdb.apache.org`
[6] `http://myria.cs.washington.edu`
[7] `http://storm.apache.org`
[8] `https://www.mongodb.com`
[9] `http://spark.apache.org/streaming`
[10] `http://cassandra.apache.org`

3.2 Related Methods to Handle Streaming Data

The *MapReduce* programming model makes it possible to easily parallelize a number of common batch data processing tasks and operate in a large cluster without worrying about system issues like failover (Dean and Ghemawat, 2008). These systems typically operate on static data by scheduling batch jobs. In *stream computing* the paradigm is to have a stream of events that flow into the system at a given data rate over which the application have no control. The streaming paradigm dictates a very different architecture than the one used in batch processing. In an attempt to create the "Hadoop" for streaming data, several methods and systems have emerged.

S4[11] addresses the need for a general-purpose distributed stream computing platform (Neumeyer et al., 2010). It is based on the *Actors* model, a model of concurrent computations in distributed systems (Agha, 1986). Here actors is the universal primitive which messages are exchanged through. Upon receiving a message, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message it will receive. Actors can keep local states, but only communicate with each other through messages. In S4, the actors are called *Processing Elements* and the messages are *events*. All nodes in S4 are homogenous, there is no centralized control. S4 assumes nodes are not added or removed from a running cluster, and it accepts loss of data in case of a failure.

Another system based on the Actors model is **Apache Flink**[12]. It is built on the philosophy that real-time analytics, batch processing and iterative and interactive algorithms, like ML and graph processing, can be expressed and executed as pipelined fault-tolerant *dataflows* (Carbone et al., 2015). It achieves this by providing two Application Programming Interfaces (APIs) on top of the streaming dataflow engine, a *DataSet API* and a *DataStream API*. This allows for both early approximate results and delayed accurate results in the same operation. The distributed dataflow runtime supports *exactly-once* state consistency through checkpointing. The process model of Flink consists of a *Flink Client*, *Job Manager* and multiple *Task Managers*, and all remote procedure calls are asynchronous messages through the actor model.

A similar system is IMB's **Stream Processing Core** (Amini et al., 2006). This system is derived from the *subscription* model, and users can "subscribe" to existing streams of data, without recompiling the processing flow-graph. It allows for users to submit continuous inquiries, expressed as processing flow graphs evaluated over a stream of data, and results of the evaluation is streamed back to the user continuously.

One of the latest additions to stream processing is **Heron**[13], Twitters successor for Apache Storm. It was designed to resolve issues related to scalability, debug-ability, manageability, and sharing of cluster resources with Storm, and it does so by changing the underlying architecture, but keeping the same API for backwards compatibility (Kulkarni et al., 2015). Heron can run on top of several resource schedulers, and relies

[11] `http://incubator.apache.org/s4`
[12] `https://flink.apache.org`
[13] `https://twitter.github.io/heron`

on **ZooKeeper**[14] for coordination. Heron manages the lifecycle of stream-processing entities called *topologies* which are directed acyclic graphs of *spouts* and *bolts*. Spouts generate the input tuples and bolts do the computation.

While Storm used a thread-based computation model, Heron uses a process-based computation model. Each *Heron Instance* is a Java Virtual Machine process with two threads, a *task execution* thread responsible for carrying out the work of a `Spout` or `Bolt`, and a *gateway* thread for communicating data and results between the task execution thread and the associated *Stream Manager*. The Stream Manager is responsible for routing the tuples and lives in a container alongside multiple Heron Instances. Several containers can run in parallel and is controlled by a single *Topology Master*.

Aurora[15], **Borealis**[16] and **Telegraph**[17] were among the pioneers of streaming databases and concepts of streaming data, such as *windows* and *incremental operators*. They are designed with a *continuous operator* model. In this model the processing is performed by a pipeline of operators with an initial source- and a final sink-operator. In between are continuous operators which processes the streaming data one record at a time before passing it on to the next operator in the pipeline. The operators are preassigned to worker nodes in the cluster. Since the operators are statically assigned to worker nodes it makes it difficult to recover quickly from failing nodes, and static allocation can also result in uneven workloads and bottlenecks.

Borealis, which is the only distributed system of the three mentioned, solves the fault-tolerance through replication. The technique involves keeping a duplicate operator on another node in the cluster, and is a resource expensive approach. AsterixDB tackles the fault-tolerance problem with a combination of operator replication and *upstream backup*, which involves buffering and replaying records in the case of a consequent node failure. This is only necessary if *at-least-once* semantics are required. AsterixDB mitigate load balancing by having worker nodes report resource statistics regularly.

Spark Streaming[18] addresses the fault-tolerance issues with a different processing model called *discretized streams* (Zaharia et al., 2013). The stream is divided into batches and dynamically scheduled and executed on the worker nodes. This model allows for parallel recovery, since the batches are stateless and deterministic.

[14] `https://zookeeper.apache.org`
[15] `http://cs.brown.edu/research/aurora`
[16] `http://cs.brown.edu/research/borealis`
[17] `http://telegraph.cs.berkeley.edu`
[18] `http://spark.apache.org/streaming`

4 Feed Ingestion in AsterixDB with Machine Learning

This chapter presents the architecture of AsterixDB feeds and the implemented system. It begins with the feed components and concepts, then presents a high-level overview of the implemented system before describing the different components in further detail.

Listing 4.1: Query statements involved with initiating a feed

```
create dataset DATASET(type) primary key ID;
create feed FEED using ADAPTOR (
    ("parameter"="value")
);
connect feed FEED to dataset DATASET
using policy POLICY
apply function LIBRARY#FUNCTION;
start feed FEED;
```

4.1 The AsterixDB Data Feed Adapter

Initiating a feed is done through the AsterixDB web interface by a query. The Asterix Query Language (AQL) compiler fetches the involved components, `feed`, `adaptor`, `policy`, `function`, and target `dataset`, and translates the `connect feed`-statement into a Hyracks job (Grover and Carey, 2015). This results in a dataflow referred to as the *feed ingestion pipeline*. The pipeline involves *intake*-, *compute*- and *store*-steps, where each step is performed by a Hyracks *data operator* that can run in parallel across nodes in the AsterixDB cluster. The intake step creates an instance of the adaptor to ingest and transform incoming data to Asterix Data Model (ADM)-records. The compute step applies any library function to each record, and the store step places the records in the target dataset and updates indexes accordingly.

In addition to the data operators, there is a *data connector* component, which repartitions the output from the operators to make them available for consuming operators, and a *feed joint* component, which is placed on operators to provide access to data flowing along the pipeline. The joints are used to build cascading networks and allows data to be routed along multiple paths simultaneously.

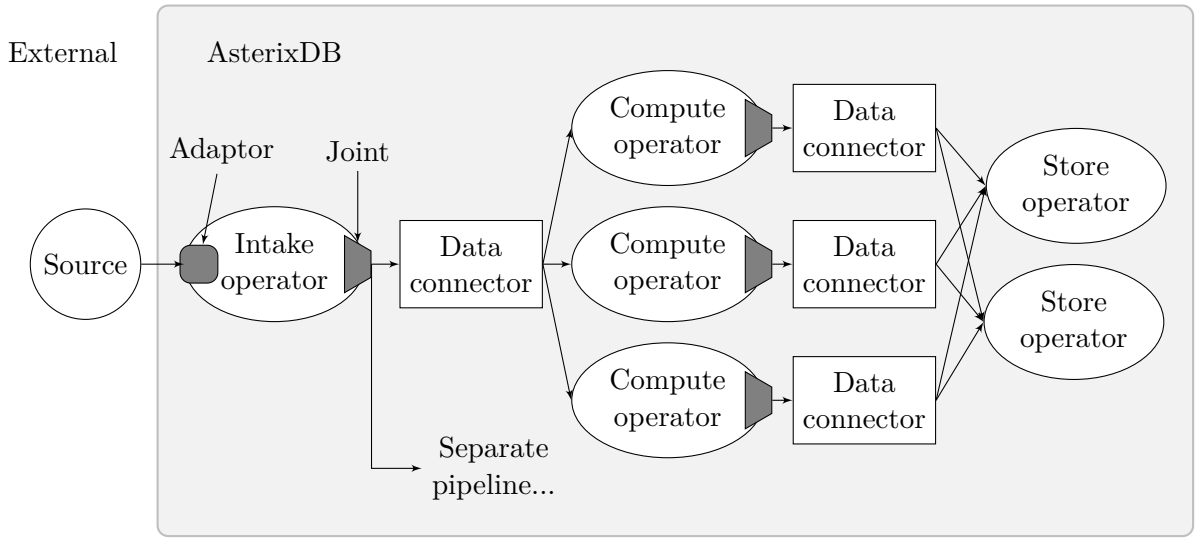

Figure 4.1: The components of the Feed Ingestion Pipeline

An AsterixDB cluster is composed of a manager node, the `Cluster Controller`, and a set of worker nodes, the (`Node Controllers`. The manager node hosts a `Central Feed Manager (CFM)` responsible for scheduling feed ingestion pipelines by mapping each of the operators to a worker node. Each worker node hosts a `Feed Managers (FMs)` that periodically reports back resource statistics to the CFM to monitor load distribution.

Each pipeline operator may define a degree of parallelism. For the intake operator, this is determined by the adaptor, meaning that multiple intake operators can run in parallel. The CFM will run a single intake operator instance if no constraints is provided. The compute operators do not provide any constraints, and the CFM will initially match the level of parallelism imposed by the store step. To cope with increasing arrival rate of data, new resources can be added to the cluster to scale out the system, and the feed ingestion pipeline may be re-structured, dependent on the ingestion policy. The parallelism of the store operators are predetermined by the nodegroup that holds the partitions of the dataset.

A pipeline can be congested in two ways: by an increasing arrival rate of data, and by an expensive User Defined Function (UDF). The system is congested when an operator lacks resources to process the data at a rate grater than, or equal to, the rate it is arriving at, hence causing a bottleneck in the pipeline. Local policies exists to determine how to handle queuing records, they may be buffered in memory or spilled to disk, for delayed processing, sampled or discarded altogether. This remains hidden from any upstream operators that will continue to receive and send data. A global strategy is to spawn multiple compute operators on idle nodes, or on nodes with low loads, to process expensive UDFs. This is made possible by the stateless and parallel nature of UDF itself, and activated in the *elastic* policy.

Runtime exceptions can occur in UDFs and could halt the entire dataflow. Every compute operator is wrapped by a *MetaFeed* operator, that acts as a sandbox environment. An exception thrown by the core of the compute operator is caught by the MetaFeed operator which removes the conflicting record from the input and resumes the core compute operator. In other words: the exception generating record is discarded.

When a node falls out of the cluster it is detected as "dead" through a heartbeat mechanism. Each operator in the pipeline receives a notification about the failure and saves its current state and contents of its buffers. The intake operator buffers all new incoming records and ceases to forward them upstream. A new pipeline is constructed using the same mechanism as when scaling due to congestion. New operators are placed alongside the old ones, and enter a *"hand-off"* from the old operators. Operators from the dead node may be scheduled to run on any other node, as all operators can co-exists on worker nodes in the cluster. The "in-flight" records of the dead node would be lost, as there is no hand-off from the dead node. To mitigate this, AsterixDB provides an optional *at-least-once* semantics, achieved by giving all records a *tracking ID* at intake and buffer them until an acknowledgment is returned from persisting the record. Upon timeout the record is replayed. The loss of an intake operator node will lead to lost records. Some sources allows retrieval of past records from a live feed, but if this is not possible multiple intake operator nodes can run in parallel to ensure higher availability of feed intake.

The loss of a store operator is equivalent to the loss of a data partition. AsterixDB currently does not support data replication, so this would lead to an early termination of the feed. If the store operator node rejoins the cluster it undergoes a log-based recovery, to ensure consistent state of all partitions, and the feed ingestion pipeline is rescheduled (Grover and Carey, 2014).

4.2 System Overview

The implemented solution of Machine Learning (ML) is designed to be installed in an AsterixDB cluster as an external library. It will then be used as a function processing all incoming data in an adapter, or applied on every record of a dataset. The model will be trained offline, in advance of installing the library.

4.2.1 Training the Model

The model is trained in an offline process. Given a dataset with annotated data, the `Text Analyzer` extracts the predefined features for each entry in the dataset. This is sent to the classifier, along with the associated class label, which builds and stores the model. The `Classifier` takes a desired classification algorithm as parameter, for example J48, Naive Bayes, or Support Vector Machine.

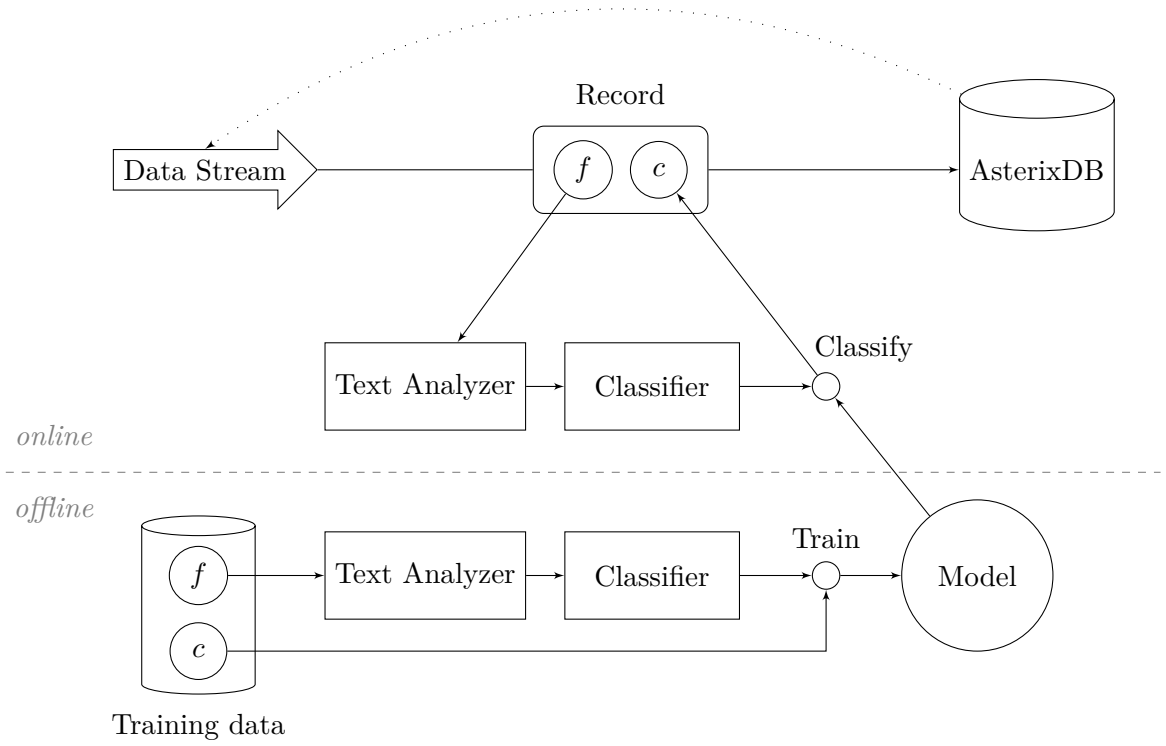

Figure 4.2: A high-level overview of the system

4.2.2 Using the Model

The library is packaged with the trained model and installed into the AsterixDB cluster. Each record in the feed or dataset is processed by the UDF where the `Text Analyzer` extracts the features and sends them to the `Classifier`. The `Classifier` then uses the model to classify each record and returns a new record with the class-attribute attached. The records must be of the same datatype as the model was trained with.

Listing 4.2: Example of an input, and accompanying output, record

```
input = { "text": "I love AsterixDB =)" }

output = { "text": "I love AsterixDB =)",
           "class": "positive" }
```

4.3 System Components

This section describes some of the components shown in Figure 4.2 in further detail.

4.3.1 Text Analyzer Component

Multiple steps are involved with classifying a text, and the first component the text meets in the system is the `Text Analyzer`. This components job is to prepare the raw text for the `Classifier`. It consists of multiple steps in a pipeline. The first step is a `Tokenizer` to turn the raw text into a vector of tokens, along with the frequency of each token. This will also remove any stop words. Next, stemming is applied to the tokens, through the `Stemmer`. This alternative is optional, and any chosen stemmer can be supplied. The final step is the `Feature Extractor`. Here the different features are retrieved from the tokens, and saved as numeric values.

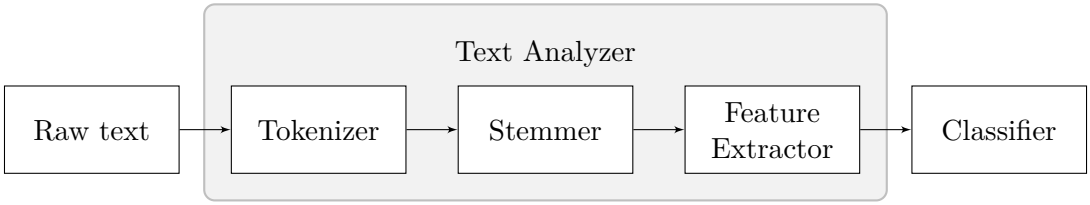

Figure 4.3: The steps of the `Text Analyzer` component

Features: The nature of metadata in Twitter described in Section 2.4.2 is used to extract features from the record. In addition to these, a simple sentiment lexicon lookup is used to give a coarse hint about the sentiment. All together, four different numeric values are used as features for building models and classifying a record. These are:

- Number of **topics**
- Number of **tags**
- Number of **links**
- A **sentiment score**

The topics, tags and links are simply counted in each record, and the sentiment score is calculated by occurrence of a token in a sentiment lexicon. The lexicon consists of two lists of words, positive and negative, where the words in each list are manually assembled to express positive or negative sentiment (Hu and Liu, 2004). If a token exists in the positive list, the sentiment score increases with 1, whilst if it exists in the negative list, the sentiment score decreases with 1. If the token is not present in any of the two lists, the sentiment score remains unchanged. When the token is categorized as a topic, meaning a preceding hashtag, the author wanted to emphasis this word (Efron, 2011), and the sentiment score calculation above can be scaled with any given factor.

4.3.2 Classifier Component

An `Instance Holder` is created with the same number of attributes as there are features. The value of these attributes are set according to the feature values extracted from the raw text. The `Instance Holder` is then used as input to the `Instance Classifier` to either train the model or to determine the class of the instance. When training, the `Instance Holder` also sets the class-value, and the `Classifier` takes the desired classification algorithm as input.

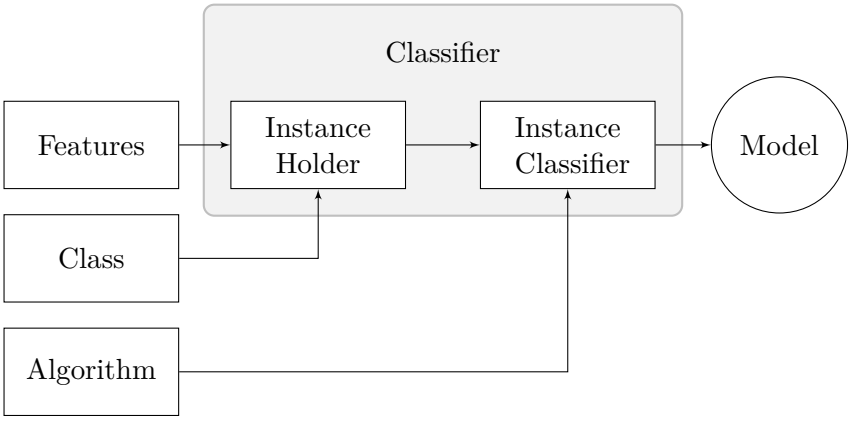

Figure 4.4: The steps of the `Classifier` component during training

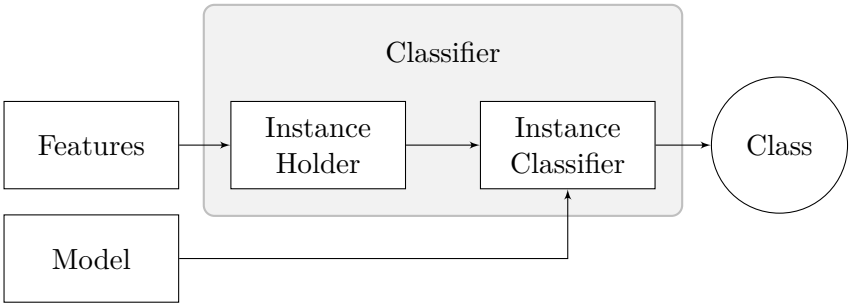

Figure 4.5: The steps of the `Classifier` component during classifying

5 Experiments and Results

This chapter consists of the planning, setup and results for the experiments conducted. It will present a baseline, along with the dataset, metrics and methodology used.

5.1 Goals with experiment

The goal stated in Chapter 1 asked if Machine Learning (ML) can scale to both stored and streaming Big Data. To determine this the experiments must be designed to measure how the implemented method handles increasing *volume* and *velocity* of data. The goal is to investigate if the implemented method can match an industry standard baseline in terms of high volumes and arrival rates of data.

It is worth noting that the accuracy of the classification is not the focus of this experiment, since the work done only concerns scaling ML to Big Data, not making the model and classification algorithms optimal for any given problem.

5.2 Baseline

Spark was chosen as the baseline for comparison. It has become an widely adopted tool for data analysis and robust libraries for both stream processing and ML has been developed by a large open-source community.

The baseline program was written using Spark's Java Application Programming Interface (API) and works simmilar to the AsterixDB solution. First, a `JavaStreamingContext` is created with a batch size of 1 second, and 0.5 seconds in turn. Next, the streaming context is connect to a socket to create a `JavaReceiverInputDStream`. Every input in this stream is transformed into a `dataframe` and passed through a series of transformation. The input is turned into tokens, stopwords are removed, features are extracted through a custom User Defined Function (UDF), and the dataframe is finally transformed by the model. The model is trained in an offline step, and stored for online classification. When the input has been classified it is written to Cassandra.

Spark version 2.1.1 is used in this evaluation, along with Spark Cassandra Connector version 2.0.0 and Cassandra version 3.10.

5.3 Evaluation Methodology

This section presents the dataset, metrics, and approach for the experimental evaluation.

5.3.1 Dataset

Training data: The model was trained with data from Sentiment140[1]. This dataset consists of 1'600'000 Tweets, annotated with positive and negative sentiment. It was gathered by Go et al. (2009) using the Twitter Search API[2], with queries for positive and negative emoticons. This distant supervision method automatically annotates large amount of data, without human labor, using emoticons as noisy labels, and is shown to achieve good accuracy. The data was stripped for anything but the content of the Tweet and stored in separate files for positive and negative. The model was then trained several times with a subset of these files to discover the minimum amount of training data required for sufficient accuracy. Testing the model revealed that the accuracy converged with 5000 positive and negative Tweets individually. The test data was manually collected by Go et al. (2009) and contains 77 negative Tweets and 182 positive Tweets. As mentioned earlier, the accuracy is not the main focus and will not be measured.

Test data: To perform the experiment with sentiment classification of Twitter statuses a dataset was collected using the Twitter Streaming API[3]. The API delivers a small percentage of all traffic passing through Twitter in realtime, and approximately 10 million unique Tweets were gathered. The records were stripped for anything but the text field, as shown in listing 5.1.

Listing 5.1: Format of the collected Tweets

```
{ "text": string }
```

Data Generator: To simulate a Twitter-stream, a dedicated `Data Generator`[4] was developed in Node.js[5] which will read the collected data and output the Tweets on a socket at a configurable rate of Tweets per second.

The `Data Generator` is set to stop emitting records if it discovers backpressure from the receiving socket, and will pause if there is more than one million outstanding records in the stream, giving the receiver time to catch up. This functionality is present to limit the memory consumption of the Data Generator.

The `Data Generator` can operate in two modes, as a client or as a server.

[1] `http://help.sentiment140.com`
[2] Read more about the Twitter search API here: `https://dev.twitter.com/rest/public`
[3] Read more about the Twitter Streaming API here: `https://dev.twitter.com/streaming/public`
[4] Data generator: `https://github.com/thormartin91/twitter-stream`
[5] Node.js: `https://nodejs.org/en/`

5.3.2 Evaluation Metrics

To investigate wether ML classification in the AsterixDB platform can scale to large quantities of Big Data, the classification algorithm will be applied to different amount of records in the dataset and the time it takes to classify them should be measured.

For each query performed in the AsterixDB browser interface, it returns the **execution time**. This would give the amount of time spent processing the query before returning the record, and thus ignoring the network delay between the server and the client and ignoring the browser loading time on the client. The experiment will also explore the **response time** from AsterixDB to the client.

To evaluate the scalability of the system, the experiment will measure the **maximum arrival-rate of records** the system can successfully process and store. The rate of incoming records will increase until the system is fully saturated. This will be repeated for an increasing number of nodes in the cluster to explore how the maximum arrival-rate evolves with a scaling cluster.

5.3.3 Experiments

Two categories of experiments will be performed to evaluate different aspects of the system. The first experiments investigates how the classification scales with increasing *volume* of offline, historical data, while the second experiments explores the streaming aspect and how the system handles increasing arrival rate, or *velocity* of data. The AsterixDB version 0.9.2-SNAPSHOT as of 19th of May was evaluated.

In the first set of experiments, each query were given 2 "warm-up" runs without recording any metrics, then they were executed 5 times each and the average execution time, or response time, was reported. The different amount of Tweets were achieved by loading smaller subsets of the total Tweets into the cluster.

These experiments were performed on a MacBook Air mid-2012 with the following hardware:

- 2 GHz Intel i7 with 2 physical cores and hyperthreading
- 8 GB DDR3 1600 MHz RAM

First, AsterixDB's user interface in the browser was used to create and load the dataset and perform all queries. Each query is packed in the `count()`-statement to keep the returned result to one printed line in the browser. The query is shown in listing 5.2.

Listing 5.2: Query performed on the different datasets

```
count(
  for $x in dataset Twitter
    return testlib#classifyText($x)
);
```

Next, an experiment was performed with an API-client called Postman[6]. In this experiment, two different requests was posted to the AsterixDB API and the response time was measured. The first query is identical to the query performed in the browser interface (listing 5.2) except that response time is measured instead of execution time. A typical response from the API for this query is shown in listing 5.3.

Listing 5.3: Response from the API when using `count()`

```
[ 10,000,000 ]
```

In the second query, the `count()`-statement was removed. This query was posted with a HTTP parameter called `mode=asynchronous`, which makes AsterixDB perform the classification, but return a *handle* to the results instead of the actual results. This is to keep the payload equal in size for all queries. A typical response is shown in listing 5.4. This handle can be used to pick up the results on a different endpoint of the API.

Listing 5.4: Response from the API when using `mode=asynchronous`

```
{ "handle": [51,0] }
```

In the next series of experiments the velocity of the data is of interest. These experiments will investigate the rate at which records can be processed by the system for varying rates of incoming data, and will make use of the `Data Generator` for data ingestion.

For different amount of nodes in the cluster, a given rate of Tweets per second will be ingested in the system for a duration of 5 minutes. The rate at which records are processed will be reported by a `Statistics` component in the UDF. The system will scale with 2, 4, 6 and 8 nodes and the ingestion rate will be 5'000, 10'000 and 15'000 records a second. Spark will run with one `Executeor`, scaling the available cores.

AsterixDB has different adapters for ingesting data, and can act as a socket server or a socket client. Both will be explored in the experiments.

These experiments were performed on a Dell OptiPlex 9020 with the following hardware:

- 3.4 GHz Intel i7 with 4 physical cores and hyperthreading
- 16 GB DDR3 1600 MHz RAM

[6] `https://www.getpostman.com`

The figures 5.1 and 5.2 show the timelines of communication between the involved components of both AsterixDB and Spark, and the listing 5.5 shows the Asterix Query Language (AQL) statement for the experiment. For each node and arrival rate, the experiments will be performed twice where the first run is a "warm-up" before datatypes, datasets, and the feed is dropped and recreated.

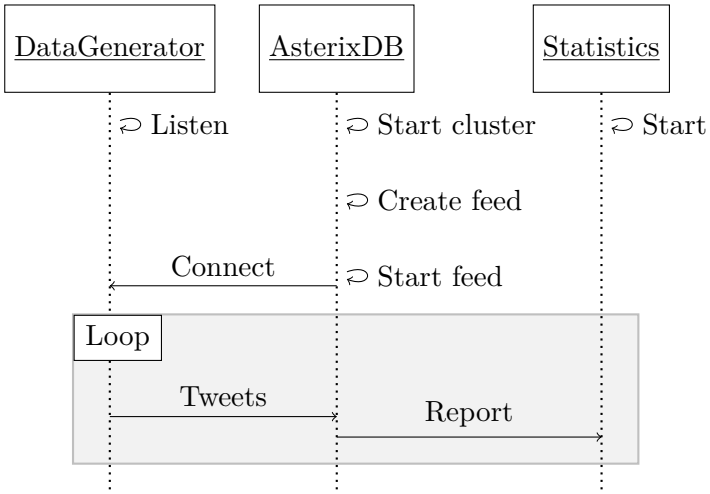

Figure 5.1: Communication timeline for AsterixDB experiments

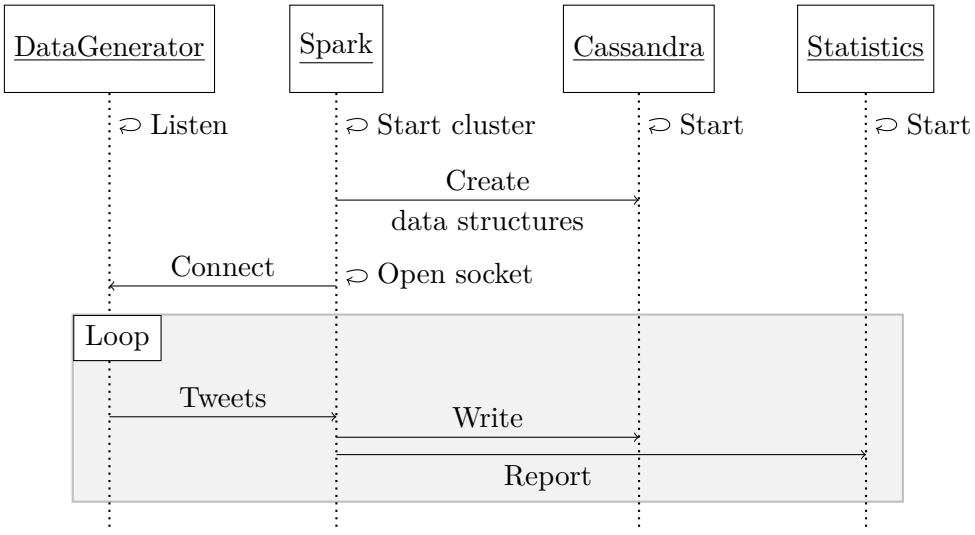

Figure 5.2: Communication timeline for Spark experiments

Listing 5.5: AQL for streaming data experiments.

```
use dataverse feeds;

drop feed SocketFeed if exists;
drop dataset TestDataset if exists;
drop type TextType if exists;

create type TextType as open { text: string };
create dataset TestDataset(TextType) primary key text;
create feed SocketFeed using socket_adapter (
  ("sockets"="127.0.0.1:10001"),
  ("address-type"="IP"),
  ("type-name"="TextType"),
  ("format"="adm")
);

connect feed SocketFeed to dataset TestDataset
apply function testlib#classifyText
using policy Basic;
start feed SocketFeed;
```

5.4 Results

This section presents the results of the experiments.

Figure 5.3 shows the result of running queries from the browser on the dataset from 1 million up to 10 million records. The 10 million records dataset took approximately 1.5 GB on disk. To see what happens when the dataset approaches the size of the main memory available on the machine, the dataset was duplicated to achieve greater size. This means that a record will appear multiple times in the dataset, but each record was given a unique ID. A 50 million records dataset was generated, which took approximately 8 GB on disk (the size of the available memory), and a 100 million records dataset was generated, which took approximately 16 GB on disk. The results are displayed in the figure 5.4. The red line is the regression line from the experiment with 10 million records, to compare with the trend of these results.

In figure 5.5 the response times from the the API client is shown. The blue line shows the the execution time for comparison. The red line shows the response time posting with the `count()`-statement. The green line shows the response time when posting without the `count()`-statement, and with the `mode=asynchronous` parameter.

Next, the results from the streaming data experiments are presented. This will show the reported Tweets per second, along with times for when backpressure occurred and when all Tweets were finally ingested.

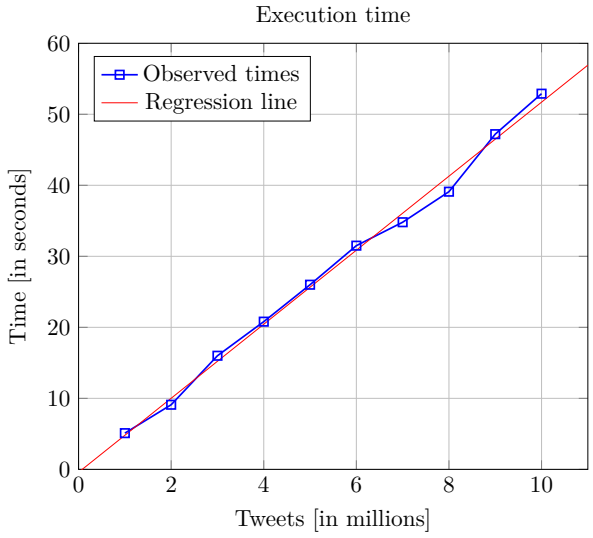

Figure 5.3: Execution time for classifying up to 10 million records

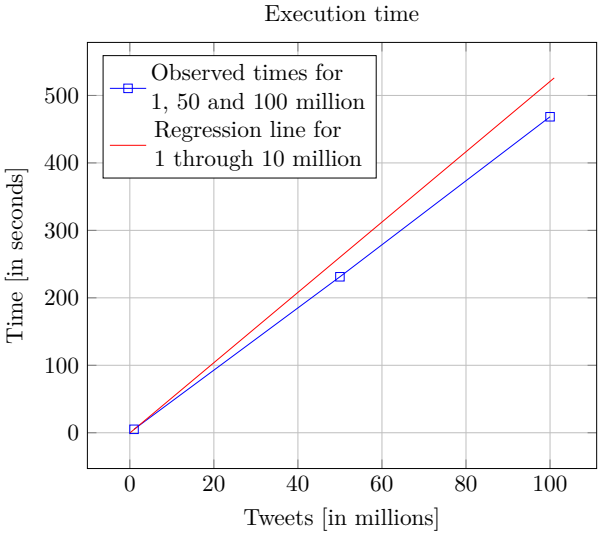

Figure 5.4: Execution time for classifying 50 and 100 million records

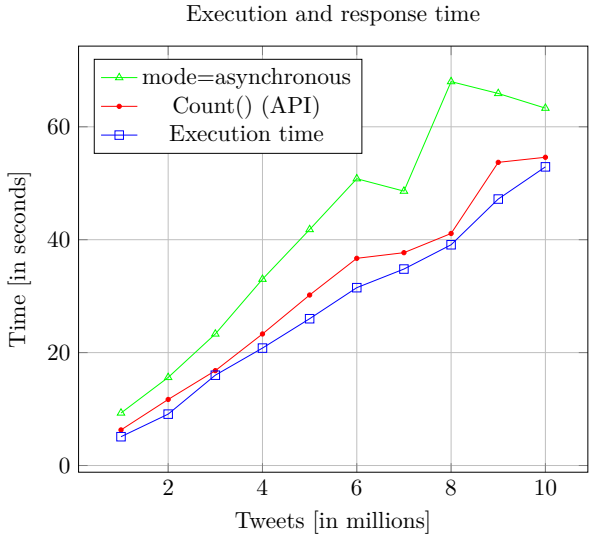

Figure 5.5: Execution time and response time for classification of historical data

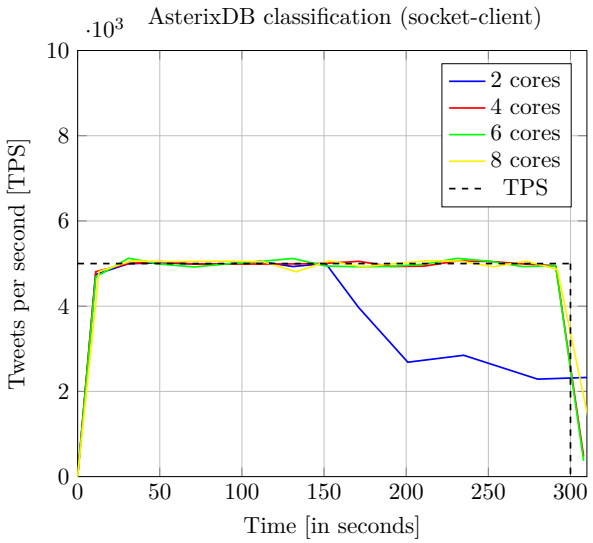

Figure 5.6: AsterixDB classification for 5'000 Tweets per second, using `socket-client` adapter.

Nodes #	*Backpressure after*	*Finished at*
2	212 s	487 s
4	None	308 s
6	None	308 s
8	None	316 s

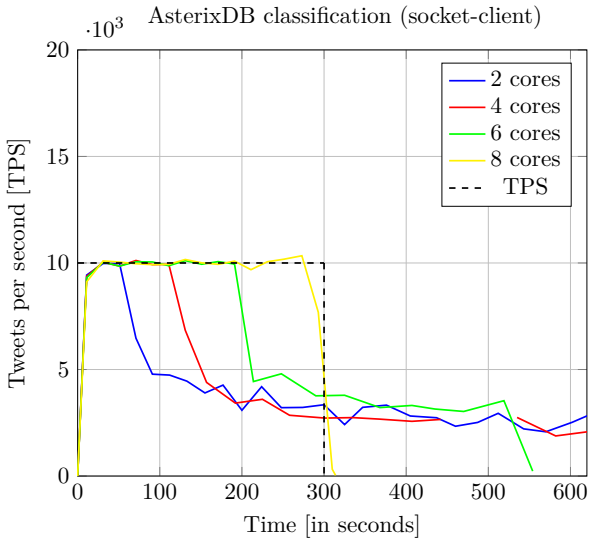

Figure 5.7: AsterixDB classification for 10'000 Tweets per second, using `socket-client` adapter.

Nodes #	*Backpressure after*	*Finished at*
2	85 s	970 s
4	149 s	979 s
6	224 s	554 s
8	None	310 s

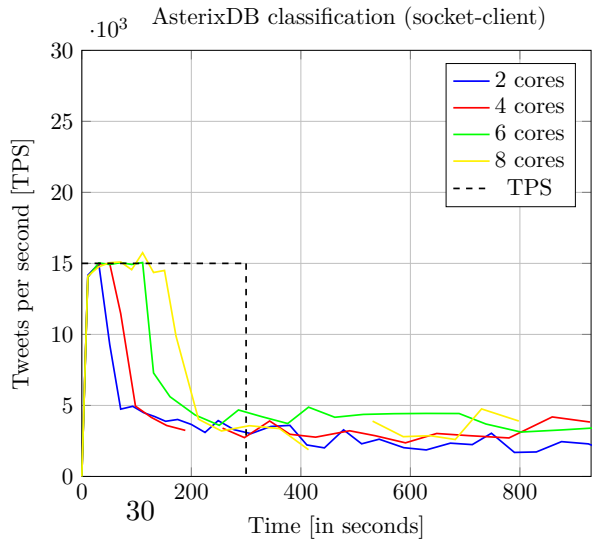

Figure 5.8: AsterixDB classification for 15'000 Tweets per second, using `socket-client` adapter.

Nodes #	*Backpressure after*	*Finished at*
2	59 s	2037 s
4	110 s	1862 s
6	138 s	1202 s
8	182 s	1089 s

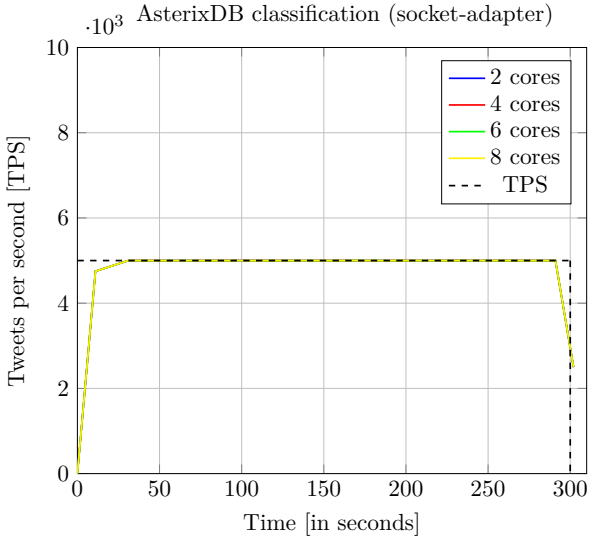

Figure 5.9: AsterixDB classification for 5'000 Tweets per second, using `socket-adapter`.

Nodes #	*Backpressure after*	*Finished at*
2	None	302 s
4	None	302 s
6	None	302 s
8	None	302 s

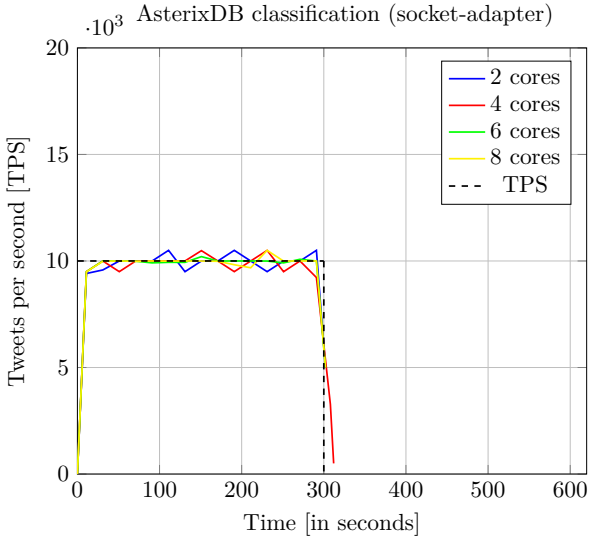

Figure 5.10: AsterixDB classification for 10'000 Tweets per second, using `socket-adapter`.

Nodes #	*Backpressure after*	*Finished at*
2	None	302 s
4	None	312 s
6	None	302 s
8	None	302 s

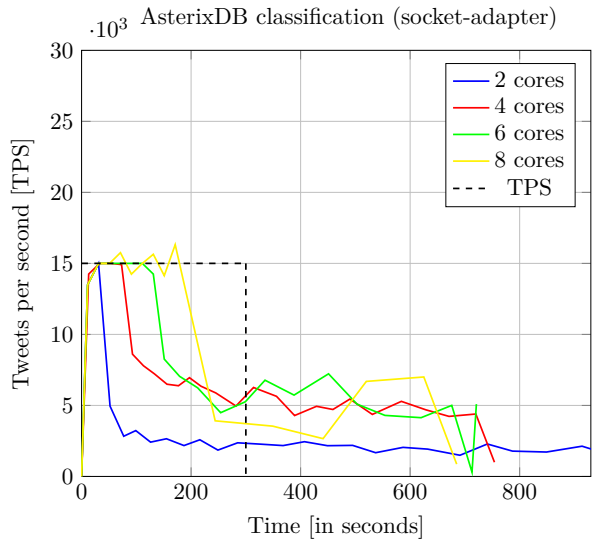

Figure 5.11: AsterixDB classification for 15'000 Tweets per second, using `socket-adapter`.

Nodes #	*Backpressure after*	*Finished at*
2	262 s	3957 s
4	100 s	754 s
6	156 s	721 s
8	190 s	685 s

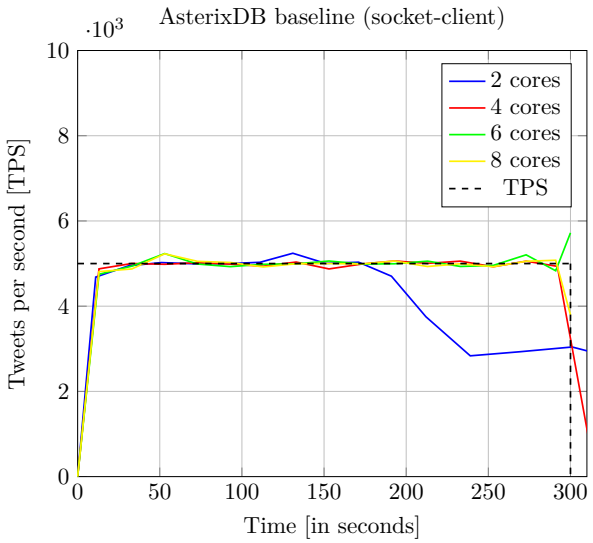

Figure 5.12: AsterixDB baseline for 5'000 Tweets per second, using `socket-client` adapter.

Nodes #	*Backpressure after*	*Finished at*
2	251 s	402 s
4	None	314 s
6	None	300 s
8	None	300 s

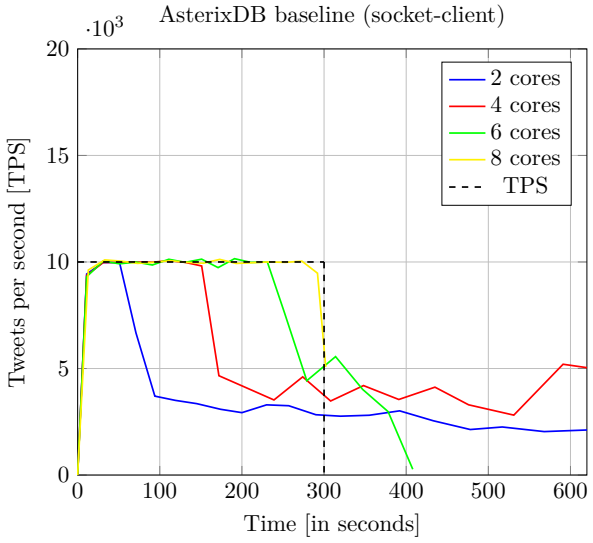

Figure 5.13: AsterixDB baseline for 10'000 Tweets per second, using `socket-client` adapter.

Nodes #	*Backpressure after*	*Finished at*
2	84 s	1290 s
4	180 s	688 s
6	270 s	408 s
8	None	302 s

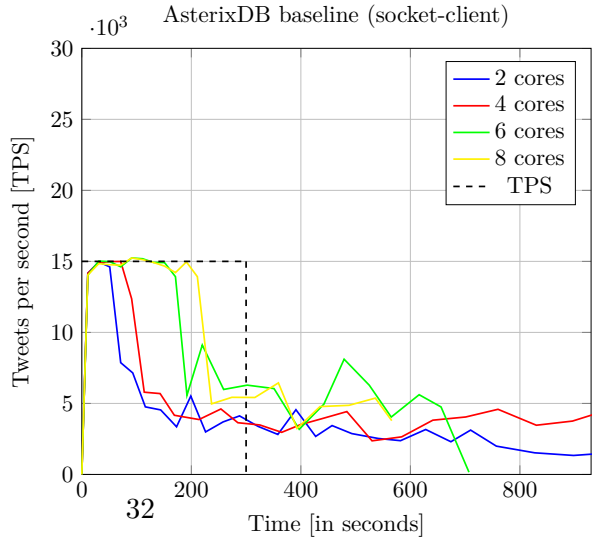

Figure 5.14: AsterixDB baseline for 15'000 Tweets per second, using `socket-client` adapter.

Nodes #	*Backpressure after*	*Finished at*
2	73 s	2177 s
4	107 s	1358 s
6	196 s	707 s
8	231 s	566 s

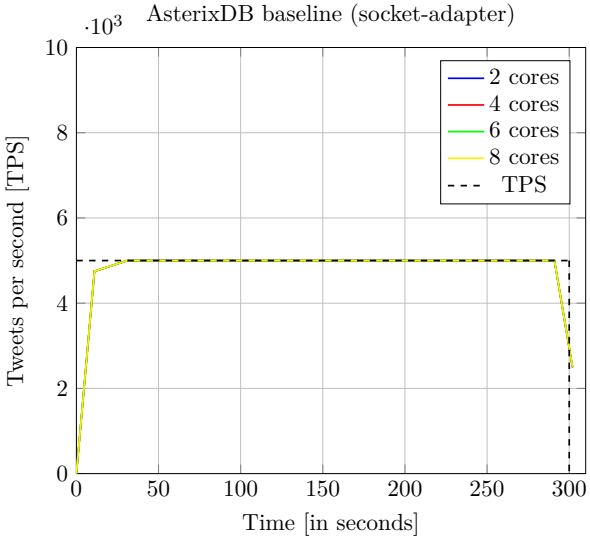

Figure 5.15: AsterixDB baseline for 5'000 Tweets per second, using `socket-adapter`.

Nodes #	*Backpressure after*	*Finished at*
2	None	302 s
4	None	302 s
6	None	302 s
8	None	302 s

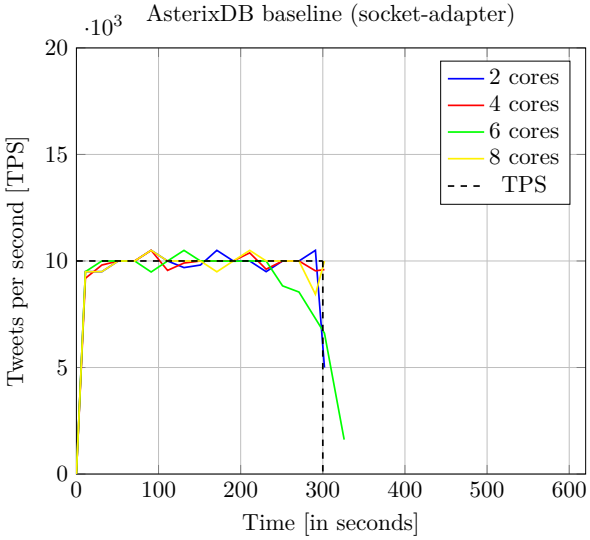

Figure 5.16: AsterixDB baseline for 10'000 Tweets per second, using `socket-adapter`.

Nodes #	*Backpressure after*	*Finished at*
2	None	302 s
4	None	302 s
6	302 s	326 s
8	None	302 s

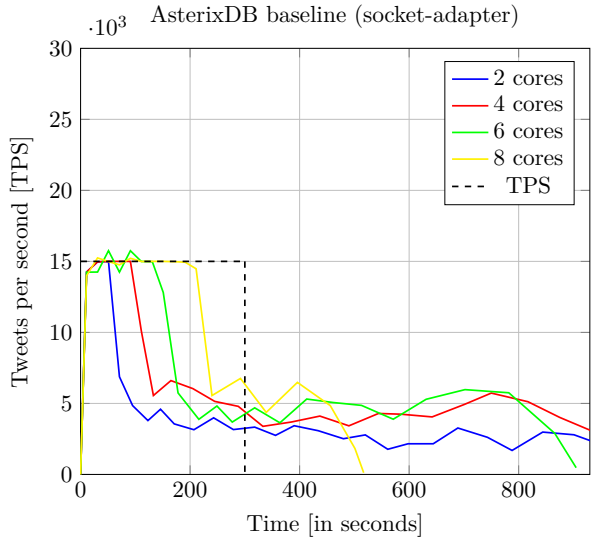

Figure 5.17: AsterixDB baseline for 15'000 Tweets per second, using `socket-adapter`.

Nodes #	*Backpressure after*	*Finished at*
2	74 s	2177 s
4	122 s	1203 s
6	170 s	905 s
8	232 s	517 s

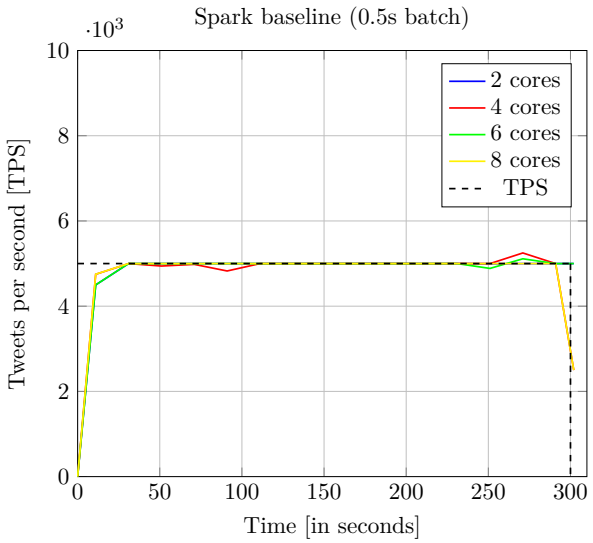

Figure 5.18: Spark classification for 5'000 Tweets per second with 0.5 second batch size.

Cores #	*Backpressure after*	*Finished at*
2	None	302 s
4	None	302 s
6	None	302 s
8	None	302 s

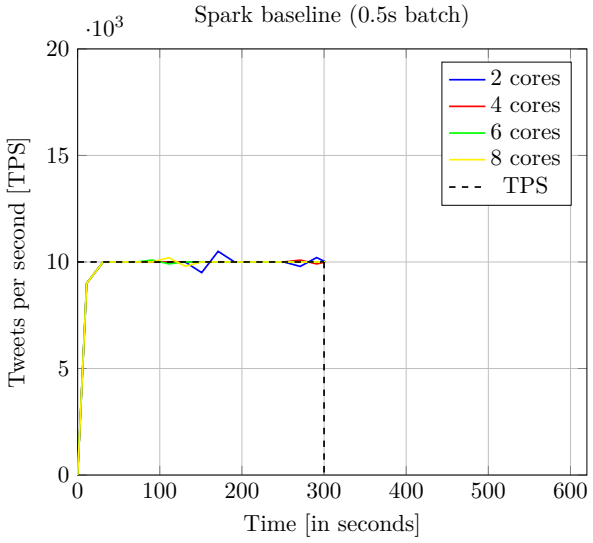

Figure 5.19: Spark classification for 10'000 Tweets per second with 0.5 second batch size.

Cores #	*Backpressure after*	*Finished at*
2	None	302 s
4	None	302 s
6	None	302 s
8	None	302 s

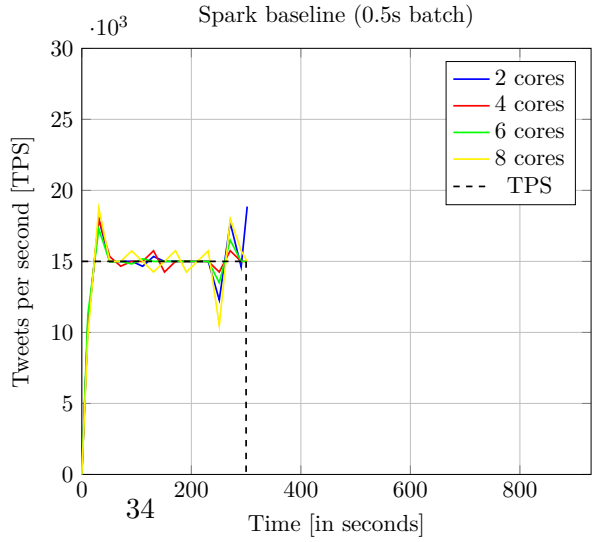

Figure 5.20: Spark classification for 15'000 Tweets per second with 0.5 second batch size.

Cores #	*Backpressure after*	*Finished at*
2	None	302 s
4	None	302 s
6	None	302 s
8	None	302 s

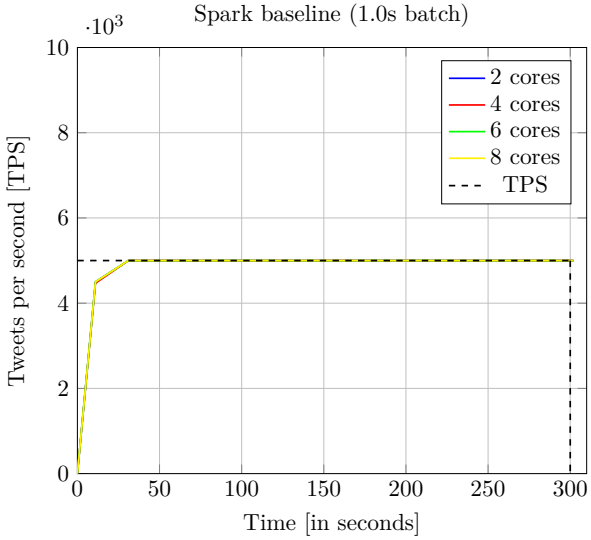

Figure 5.21: Spark classification for 5'000 Tweets per second with 1 second batch size.

Cores #	*Backpressure after*	*Finished at*
2	None	302 s
4	None	302 s
6	None	302 s
8	None	302 s

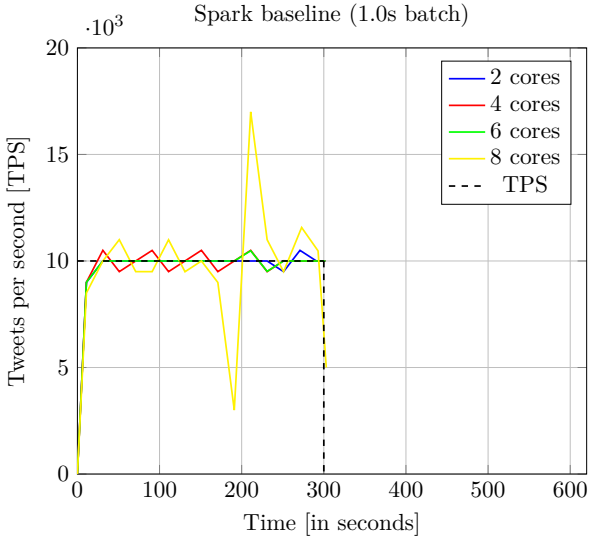

Figure 5.22: Spark classification for 10'000 Tweets per second with 1 second batch size.

Cores #	*Backpressure after*	*Finished at*
2	None	302 s
4	None	302 s
6	None	302 s
8	None	302 s

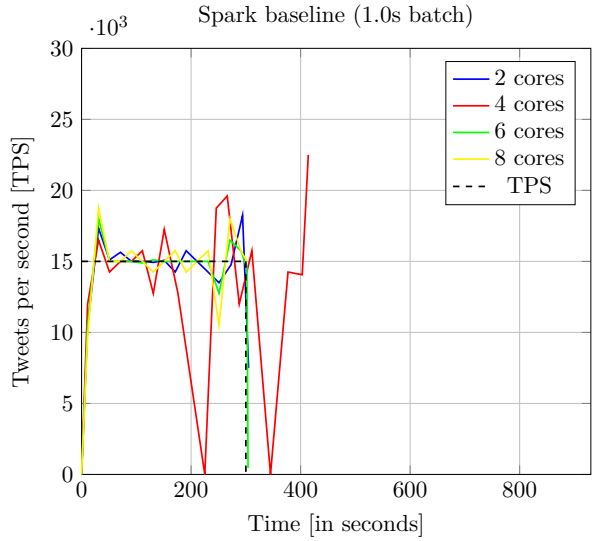

Figure 5.23: Spark classification for 15'000 Tweets per second with 1 second batch size.

Cores #	*Backpressure after*	*Finished at*
2	None	305 s
4	None	414 s
6	None	304 s
8	None	302 s

6 Evaluation

This chapter will evaluate the performed experiments and give a discussion of the results.

6.1 Evaluation

10 million records take about 1.5 gigabyte of storage and could easily fit in main memory of the cluster. To see how the the processing scales to Big Data, an experiment would have to be performed, where at least the dataset would not fit in main memory. This is shown in figure 5.4.

One query in the offline experiments, figure 5.3, wraps the result in a `count()`-statement. The reason for this is to keep the body of the returned results equally large for all queries and lightweight, and avoid printing 10 million records in the browser interface. The consecutive experiment was coined to see what impact the the `count()`-statement had on the execution time.

There are larger deviations for the query with the `mode=asynchronous` parameter, and it is not unlikely that they are caused by other processes on the machine. For this experiment, the client and AsterixDB executed on the same machine. Section 7.2 makes suggestions for isolating the experiments further.

Notice that in both the queries to Spark and to AsterixDB for sreaming data experiments, the processing rate is reported to the `Statistics` component without knowledge of the process of persisting the records. It is not possible to gain this insight without altering the source code of the underlying systems, and we can only report the rate as observed in the User Defined Function (UDF). For this reason persistence is treated as a "black-box" and the records is assumed to be persisted once submitted by the UDF.

The experiments were conducted with only one Cassandra instance. Scaling this number would require a layer between Spark and Cassandra to coordinate consistency, handle indexing and partitioning. It is likely that this would cause overhead regarding the throughput of persistence and section 7.2 makes suggestions for further investigation.

The `Data Generator` was executed on the same machine as both the AsterixDB cluster and the Spark cluster. This leads to less overhead in terms of network delay, but will also require it to share resources with the cluster. A backpressure mechanism was implemented into the `Data Generator` to avoid flooding the stream with outstanding records. The resource usage of the `Data Generator` was closely monitored and never exceeded 6 % memory consumption and 10 % CPU execution time. The average values of resource consumption for the `Data Generator` was 1.6 % memory and 4.8 % CPU.

6.2 Discussion of Results

As seen in figure 5.3, the time it takes to process the records increases with the amount of records to process. This is expected, and the rate at which the execution time increases is what is interesting. Although the line is not perfectly straight, it is evident that the increase is linear. It takes the same amount of time to process a record, independently of how large the dataset is, at least for a dataset of the size used. The points that deviate from the straight line is probably caused by other processes on the computer fighting for resources. Additional experiments are proposed in Section 7.2.

When looking at the regression line from the offline experiment on the larger dataset, figure 5.4, the execution time for 50 million and 100 million records indicate that the processing scales $\mathcal{O}(n)$.

In the response time experiment, figure 5.5, the same query posted through the API and the browser had almost identical results, with the one posted through the API slightly higher. A reason for this might be that the time recorded is the response time, and a few milliseconds is required to transfer the results, as opposed to the execution time in the browser query.

The results from the query without the `count()`-statement, and with the `mode= asynchronous` parameter is also slightly higher. An explanation for this might be that the payload in the response of this query is a few bytes larger than the other query, and therefore the response time increases. This results also deviates much more than the others, and before concluding with that being caused by processing overhead, further experiments will have to be performed.

The conclusion to be drawn from the offline experiments is that on the same hardware, scaling only the amount of data has no impact on the performance of the classification. The throughput remains the same.

As for the streaming data experiments it is evident that the classification does not add significant overhead to AsterixDB when comparing the plots for classification to the AsterixDB baseline. There is notable differences for the throughput of `socket-adapter` compared to `socket-client` adapter, both for the baseline and with classification. The `socket-adapter` has a steadier throughput for all size clusters when the arrival rate is low, and this adapter is being more actively developed of the two.

As the arrival rate increases records are pending in the stream for AsterixDB, both for the baseline and the classification implementation. However we observe that this occurs later when more nodes are present in the cluster, thus indicating scalable processing and that AsterixDB performance improves with distribution.

The experiments show that the implemented solution were able to handle arrival rates of up to 10'000 Tweets per second using the `socket-adapter`, in accordance with the results of Grover and Carey (2015).

Compared to Spark, AsterixDB does not keep up with the throughput as the arrival rate increases. This is true for both the classification implementation and the AsterixDB baseline. Spark's strategy of batching records proves efficient, although this strategy inherently adds a small latency and the records will be seen in *near* realtime as opposed

to closer to real-time in AsterixDB.

Spark also never caused backpressure to the `Data Generator` and any queueing records did not cause congestion in the stream. If Spark was prevented from processing in a timeframe it ingested twice as many records in the next batch.

The baseline system of Spark and Cassandra was designed to ingest and store data, and does not enable analytical queries of historical data. This is provided "out-of-the-box" with AsterixDB.

In previous experiments, AsterixDB outperformed Spark and Cassandra in the task of sentiment analysis with regards to throughput (Pääkkönen, 2016), which contradicts the results of this report. There are several reasons for this. Pääkkönen performed classification on records by reading them from Cassandra after they were persisted and write the result back. This yields twice as many write operations and an additional read operation for each record. Performing classification before persistence was suggested by Pääkkönen and is the way the experiments in this report are conducted. In addition, Pääkkönen evaluated Spark version 1.3.1, while the experiments in this report were performed with Spark version 2.1.1. Both Spark Streaming, Spark MLlib and the Spark Core has seen performance improvements between these versions[1].

To conclude the streaming data experiments, classification in AsterixDB imposes minimal overhead, and the solution scales gracefully with the number of nodes in the cluster. Nevertheless, Spark achieves greater throughput for high arrival rates.

[1] `https://spark.apache.org/releases/spark-release-2-0-0.html`
`https://spark.apache.org/releases/spark-release-2-1-0.html`

7 Conclusion

This chapter presents a conclusion to the work done in this project and points out directions for future work.

7.1 Conclusion

This project sat out to explore whether Machine Learning (ML) methods could scale to both stored Big Data and streaming Big Data. When investigating *state-of-the-art* Big Data platforms, AsterixDB proved promising for both handling streaming data with *Data Feeds* and allowing parallel execution of user code with *User Defined Function (UDF)*. For a ML library to fit with this task it must to be general, extensible and feasible to implement in the same programming language as AsterixDB, namely Java. The WEKA-library fulfills all of these requirements. WEKA has been implemented into the Big Data Management System (BDMS) AsterixDB for the purpose of ML. Specifically, classification was performed on Twitter data to determine the sentiment of a Tweet. Experiments with 100 million records show that the classification scaled $\mathcal{O}(n)$ for historical data on a single node setup, and imposes minimal overhead while classifying real time streaming data. The system handled throughputs of up to 10'000 Tweets per second, and scales gracefully with distribution of the cluster.

Experiments revealed that the implementation of Spark and Cassandra achieved greater throughput than that of ML in AsterixDB, but the batching of records in Spark will only provide *near* real-time processing. Also, Spark is not designed for data storage, and the tasks of handling partitioning, indexing and replication need to be incorporated in a Spark + Cassandra technology to achieve functionality that AsterixDB provides by default. In addition, historical queries were not implemented or evaluated in Spark and Cassandra.

One of the benefits of running ML on a system for both stored and streaming Big Data is the ability to run queries on real-time and historical data simultaneously, revealing information one might not have discovered looking at them in isolation. The historical data can also be used to improve classification. As data arrives in the system and is accumulated with the historical data, a classification model can be trained on this data periodically and continuously adapt to the varying data.

An argument for using modular systems like Spark + Cassandra is that tools are specialized to solve individual tasks and while this might be a more performance efficient strategy than a "one-size-fits-a-bunch" approach, it requires developers to acquire knowledge of a range of technologies in order to maintain the system.

7.2 Future Work

To extend the experiments performed in this report, one can consider isolating the experiment further by running client and cluster on different nodes. This will leave all processing power on the cluster node to perform ML, avoiding competition on resource usage.

Notice also that the experiments simulated cluster nodes with processor cores. An extended experiment could investigate the effect of having separate, physical machines for each node in the cluster. One could also vertically scale up the hardware on each node to see how this impacts the processing.

The experiments were conducted with only one Cassandra instance, and the effect of scaling the number of these instances could be examined to inspect how the rate of throughput evolves.

The implementation of Spark + Cassandra did not involve the ability to run analytical queries on historical data. Since this combination of technologies proved efficient with regards to throughput of streaming data it would be interesting to see how the inclusion of query functionality affects the performance, compared to that of AsterixDB.

Finally, studying the accuracy of sentiment analysis is beyond the scope of this study. Achieving high performing sentiment analysis in the AsterixDB platform is left for future work.

Bibliography

Gul A. Agha. ACTORS: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, and Till Westmann. AsterixDB: a scalable, open source BDMS. Proceedings of the VLDB Endowment, 7(14):1905–1916, 2014. URL http://dl.acm.org/citation.cfm?id=2733085.2733096.

Xavier Amatriain. Mining Large Streams of User Data for Personalized Recommendations. SIGKDD Explor. Newsl., 14(2):37–48, 2013. ISSN 1931-0145. doi: 10.1145/2481244.2481250. URL http://doi.acm.org/10.1145/2481244.2481250{\%}5Cnhttp://dl.acm.org/citation.cfm?doid=2481244.2481250.

Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. SPC: A Distributed, Scalable Platform for Data Mining. In Proceedings of the 4th international workshop on Data mining standards, services and platforms, pages 27–37, 2006. ISBN 159593443X. doi: 10.1145/1289612.1289615. URL http://dl.acm.org/citation.cfm?id=1289615.

Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache Flink: Unified Stream and Batch Processing in a Single Engine. Data Engineering, 36:28–38, 2015.

Michael J Carey, Steven Jacobs, and Vassilis J Tsotras. Breaking BAD: A data serving vision for big active data. In DEBS 2016 - Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems, pages 181–186, 2016. ISBN 9781450340212. doi: 10.1145/2933267.2933313. URL http://dx.doi.org/10.1145/2933267.2933313.

J Dean and S Ghemawat. MapReduce : Simplified Data Processing on Large Clusters. Communications of the ACM, 51(1):1–13, 2008. ISSN 00010782. doi: 10.1145/1327452.1327492. URL http://www.usenix.org/events/osdi04/tech/full{_}papers/dean/dean{_}html/.

Laney Douglas. 3d data management: Controlling data volume, velocity and variety. Application Delivery Strategies, 6:4, 2001. ISSN 09505849. doi: 10.1016/j.infsof.2008.09.005.

Miles Efron. Information search and retrieval in microblogs. Journal of the American Society for Information Science and Technology, 62(6):996–1008, jun 2011.

Nada Elgendy and Ahmed Elragal. Big Data Analytics: A Literature Review Paper. Advances in Data Mining. Applications and Theoretical Aspects, 8557:214–227, 2014. ISSN 0302-9743. doi: 10.1007/978-3-319-08976-8_16. URL `http://link.springer.com/10.1007/978-3-319-08976-8{\_}16`.

Wei Fan and Albert Bifet. Mining Big Data : Current Status , and Forecast to the Future. ACM SIGKDD Explorations Newsletter, 14(2):1–5, 2013. ISSN 19310145. doi: 10.1145/2481244.2481246.

Alec Go, Richa Bhayani, and Lei Huang. Twitter Sentiment Classification using Distant Supervision. Processing, 150(12):1–6, 2009. ISSN 00370738. doi: 10.1016/j.sedgeo.2006.07.004. URL `http://www.stanford.edu/{~}alecmgo/papers/TwitterDistantSupervision09.pdf`.

Raman Grover and Michael J. Carey. Scalable Fault-Tolerant Data Feeds in AsterixDB. CoRR, 2014. URL `http://arxiv.org/abs/1405.1705`.

Raman Grover and Michael J Carey. Data Ingestion in AsterixDB. Edbt, pages 605–616, 2015. doi: 10.5441/002/edbt.2015.61.

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian Witten. The WEKA data mining software: An update. SIGKDD Explorations, 11(1):10–18, 2009. ISSN 19310145. doi: 10.1145/1656274.1656278.

Minqing Hu and Bing Liu. Mining and summarizing customer reviews. Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining KDD 04, 04:168, 2004. URL `http://portal.acm.org/citation.cfm?doid=1014052.1014073`.

T Kraska, A Talwalkar, J Duchi, R Griffith, M Franklin, and M Jordan. MLbase : A Distributed Machine-learning System. 6th Biennial Conference on Innovative Data Systems Research (CIDR'13), 2013.

Sanjeev Kulkarni, Nikunj Bhagat, Masong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream Processing at Scale. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15, pages 239–250, 2015. ISSN 07308078. doi: 10.1145/2723372.2742788. URL `http://dl.acm.org/citation.cfm?id=2723372.2742788`.

Avinash Laksham and Malik Prashant. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, pages 1–6, 2010. ISSN 01635980. doi: 10.1145/1773912.1773922. URL `http://dl.acm.org/citation.cfm?id=1773922`.

Bing Liu. Sentiment Analysis and Opinion Mining. Synthesis Lectures on Human Language Technologies, 5(1):1–167, 2012. ISSN 1947-4040. doi: 10.2200/S00416ED1V01Y201204HLT016.

Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab. Proceedings of the VLDB Endowment, 5(8):716–727, 2012. ISSN 21508097. doi: 10.14778/2212351.2212354. URL http://arxiv.org/abs/1204.6078{\%}5Cnhttp://dl.acm.org/citation.cfm?id=2212354{\%}5Cnhttp://dl.acm.org/citation.cfm?doid=2212351.2212354.

Amr Magdy. Scalable Microblogs Data Management. In SIGMOD'16 PhD Proceedings of the 2016 on SIGMOD'16 PhD Symposium, pages 32–36, 2016. ISBN 9781450341929. doi: 10.1145/2926693.2929898.

Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. [seminal] MLlib: Machine Learning in Apache Spark. Journal of Machine Learning Research, 17:1–7, 2016. ISSN 1532-4435. URL http://arxiv.org/abs/1505.06807.

Gilad Mishne, Jeff Dalton, Zhenghua Li, Aneesh Sharma, and Jimmy Lin. Fast data in the era of big data: Twitter's real-time related query suggestion architecture. Proceedings of the 2013 ACM SIGMOD International Conference on management of data, pages 1147–1158, 2013. ISSN 07308078. doi: 10.1145/2463676.2465290.

Tom M Mitchell. Machine Learning. 1997. ISBN 0071154671. doi: 10.1145/242224.242229. URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20{\&}path=ASIN/0070428077.

Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In Proceedings - IEEE International Conference on Data Mining, ICDM, pages 170–177, 2010.

Pekka Pääkkönen. Feasibility analysis of AsterixDB and Spark streaming with Cassandra for stream-based processing. Journal of Big Data, 3(1):6, 2016. ISSN 2196-1115. doi: 10.1186/s40537-016-0041-8. URL http://journalofbigdata.springeropen.com/articles/10.1186/s40537-016-0041-8.

Tilmann Rabl and S Gómez-Villamor. Solving big data challenges for enterprise application performance management. Proceedings of the VLDB Endowment, 5(12):1724–1735, 2012. ISSN 21508097. doi: 10.14778/2367502.2367512. URL http://dl.acm.org/citation.cfm?id=2367512.

Anand Rajaraman and Jeffrey D Ullman. Mining of Massive Datasets. Lecture Notes for Stanford CS345A Web Mining, 67:328, 2011. ISSN 01420615. doi: 10.1017/CBO9781139058452. URL http://ebooks.cambridge.org/ref/id/CBO9781139058452.

Ian H Witten, Eibe Frank, and Mark a Hall. Data Mining: Practical Machine Learning Tools and Techniques, volume 54. 2011. ISBN 9780123748560. doi: 10.1002/1521-3773(20010316)40:6<9823::AID-ANIE9823>3.3.CO;2-C. URL http://www.cs.waikato.ac.nz/{~}ml/weka/book.html{\%}5Cnhttp://www.amazon.com/Data-Mining-Practical-Techniques-Management/dp/0123748569.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, and Ankur Dave. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI, pages 2–2, 2012. ISSN 00221112. doi: 10.1111/j.1095-8649.2005.00662.x. URL https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf.

Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. Sosp, (1):423–438, 2013. doi: 10.1145/2517349.2522737. URL http://dx.doi.org/10.1145/2517349.2522737.