Pääkkönen *J Big Data (2016) 3:6*
DOI 10.1186/s40537-016-0041-8

Journal of Big Data

RESEARCH Open Access

Feasibility analysis of AsterixDB and Spark streaming with Cassandra for stream-based processing

Pekka Pääkkönen*

*Correspondence: pekka.paakkonen@vtt.fi
VTT Technical Research Centre of Finland, Oulu, Finland

Abstract
For getting up-to-date insight into online services, extracted data has to be processed in near real time. For example, major big data companies (Facebook, LinkedIn, Twitter) analyse streaming data for development of new services. Several technologies have been developed, which could be selected for implementation of stream processing functionalities. The contribution of this paper is feasibility analysis of technologies for stream-based processing of semi-structured data. Particularly, feasibility of a Big Data management system for semi-structured data (AsterixDB) will be compared to Spark streaming, which has been integrated with Cassandra NoSQL database for persistence. The study focuses on stream processing in a simulated social media use case (tweet analysis), which has been implemented to Eucalyptus cloud computing environment on a distributed shared memory multiprocessor platform. The results indicate that AsterixDB is able to provide significantly better performance both in terms of throughput and latency, when data feed functionality of AsterixDB is used, and stream processing has been implemented with Java. AsterixDB also scaled on the same level or better, when the amount of nodes on the cloud platform was increased. However, stream processing in AsterixDB was delayed by batching of data, when tweets were streamed into the database with data feeds.

Keywords: Sentiment, Tweet, Word count, AsterixDB, Spark, Performance, Eucalyptus, Cassandra

Introduction
Streaming data is increasingly important for online services. Facebook [1] and LinkedIn [2] have analysed event related data for understanding usage in their ecosystems. Twitter has created a big data streaming architecture, which is able to serve and process thousands of tweets in a second [3–5]. Also, several systems [6], methods [7, 8], and benchmarking tools [9–11] have been created for facilitating implementation of tweet related processing and analysis by 3rd parties. Especially, new stream processing technologies (Spark [12], AsterixDB [13]) have been created, which could be selected for implementation of stream extraction, storage and analysis functionalities. Although stream processing performance has been studied [12, 14], comparative feasibility analysis of the technologies has not been extensively performed in the context of semi-structured data processing.

This article focuses on performance analysis of Spark streaming, Cassandra, and AsterixDB technologies for stream processing of semi-structured social media data (tweets). Especially, Spark streaming has been integrated with Cassandra for data persistence, which has been compared to AsterixDB on Eucalyptus cloud environment on a DSM multiprocessor platform. The results indicated that AsterixDB achieved significantly higher throughput and lower latency, when data feeds were utilized and stream processing was implemented with Java. Performance of AsterixDB also scaled better, when the amount of nodes on the cloud platform was increased. However, stream processing in AsterixDB was delayed by batching of streamed tweets.

The document is structured as follows. First, background and literature review are presented, which is followed by comparison of AsterixDB and Spark streaming technologies. Then, research design and methodology is provided, and experiments are described. Subsequently, results are presented, analysed, discussed, and concluded. In the final chapter the research method is described in detail.

Background and literature review

As mentioned above, semi-structured data has been processed in several commercial use cases. On a more detailed level, Facebook has collected log data with Scribe, and performed analysis on Hive-Hadoop clusters [1]. LinkedIn has logged activity data from users with Kafka, which has been analysed with batch-based analysis tools [2]. Twitter processes tweets for the creation of new end user services [3]. Twitter's big data system architecture aims at real time analysis by tokenization, annotation, filtering, and inverted indexing of tweets with their EarlyBird search engine [4]. Twitter's latest development is Heron, which has replaced Storm as their stream processing technology [5]. Tweets have been utilized for creating insights in several businesses. Examples include competitive analysis among US pizza chains [15] and retail companies [16], and insights created for supply chain practises [17].

Methods and algorithms for social big data can be categorized into network analytics, community detection algorithms, text analytics, information diffusion models and methods, and information fusion [18]. Sentiment analysis [19] can be considered as part of text-based analytics, and it has been extensively studied [8]. Sentiment analysis is a classification problem, where sentiment can be analysed on document-level, sentence-level or aspect-level [19]. Classification techniques for sentiment analysis can be broadly classified into lexicon-based and machine learning approaches [20]. In the lexicon-based approach a sentiment/opinion lexicon is utilized for sentiment analysis. Machine learning can be divided into supervised or un-supervised learning approaches [20]. In the supervised approach a classification model is typically trained based on labelled training documents. Training data is not needed in the unsupervised approach, where sentiment categories are learned based on text-based corpus.

Several tools have been developed for sentiment analysis of tweets. 20 different commercial and open source sentiment analysis tools have been tested in different test beds [21]. It was discovered that Sentiment140 achieved best overall performance (accuracy >61–71 %) across the test beds among stand-alone tools. In the SemEval competition overall sentiment of tweets has been evaluated with an accuracy of ~64 % [22]. Almost all teams utilized supervised learning in the algorithms. Also, many web services are available for sentiment analysis [23]. Experiments with a Twitter data set indicated that AlchemyAPI achieved best accuracy (62.5 %).

SentiWordNet was selected as an open lexical resource for experimentation of this work, because an earlier study indicated good coverage and comparable prediction performance to alternatives [24]. SentiWordNet uses a synset of three numerical scores to describe polarity (objectivity, negativity, and positivity) of terms in the synset [25]. Each synset contains one or more words, which are associated with polarity scores. SentiWordNet 3.0 improves accuracy by ~20 %, when compared to the original solution [26].

Several tools and systems have been developed for extraction and analysis of tweets. Previous literature on data collection, data management, and languages for querying and analysing of tweets has been reviewed, of which only some are reported here [6]. Twarql encodes extracted tweets in a structured format [resource description framework (RDF)], and enables querying of tweets with SPARQL protocol and RDF query language (SPARQL) [27]. A large-scale distributed system for real time sentiment analysis has been developed [28]. The system consists of a lexicon builder, and a sentiment classifier, which are executed with MapReduce, and a distributed database system (HBase). The system scales in terms of processing time, when data size and number of instances are increased. Taghreed is a system for scalable querying and visualization of geotagged tweets/microblogs [29]. The system is able to manage, query, and visualize billions of tweets with main memory indexes, an optimized query engine, a recovery manager, and an interactive visualizer. Nimbus offers tunable filters for Twitter streams [30]. It has integrated Spark, web server, and MySQL for storing and filtering of tweets. A sentiment analysis process using part-of-speech tagging, and identification of negation phrases has been proposed, which has been validated with a product review data set from Amazon.com [31].

In order to get instant insight into analysed results, tweets have to be processed in near real time. Several technologies have been developed, which could be selected for serving such purpose [32]. In this paper Spark streaming and AsterixDB technologies have been focused on. Spark divides a data stream into batches, which are stored in memory, and processed via parallel operations [12]. Spark streaming architecture is comprised of a master, worker nodes, and a client library. The master schedules tasks based on data, which is received from the client library. Worker nodes receive data from the master, and execute tasks on it. Spark offers a distributed memory abstraction to the programmer with resilient distributed datasets (RDD) [33]. Additionally, Spark offers Java, Scala, and Python programming APIs for creation of stream processing tasks. Spark can also be connected to databases, for example with adapters [34].

Spark has achieved high per-node throughput, sub-second latency, and fault recovery [12]. Shark [35], an enhanced version of Spark, provides 100× faster queries than Apache Hive, and machine learning 100× faster than Hadoop. Nowadays, development of Shark has been continued as part of the Spark project [36]. The on-going work of Spark has focused on improvement of usability and performance with visualization, expansion of data science APIs, and memory management outside of Java virtual machine (JVM) (Tungsten) [37, 38]. Benchmarking of Spark against Apache Flink indicated that graph processing, data mining, and relational queries were processed faster with Flink [39]. Spark was faster in batch processing algorithms. Spark performance bottlenecks have been analysed with SQL benchmarks [40]. It was discovered that CPU was often the bottleneck (as opposed to I/O), improvement of network performance led only to a small improvement in the total execution time of jobs, and many straggler causes could be identified and fixed.

Spark's performance has also been studied in the context of machine learning, particularly with Spark's machine learning library (MLlib) [41]. MLlib's linear regression has been used for predicting power load based on weather data and power load data [42]. A model (MLlib's logistic regression) has been trained based on context specific grammars for predicting sentiment polarity of internet movie database (IMDb) reviews [43], which achieved ~64 % accuracy. Another model has been created for sentiment analysis of tweets, which was trained based on hashtags and emoticons [44]. Running time of tweet classification decreased, when the amount of nodes was increased in the Spark cluster.

AsterixDB may also be used for processing of streaming data. One of the main design goals of AsterixDB has been to create a 'one size fits a bunch' architecture, where one technology can be utilized in multiple use cases without gluing multiple technologies together [45]. AsterixDB has a flexible data model and query language (AQL) for describing, querying, and analysing semi-structured data [13]. Algebricks is a data model-agnostic layer in AsterixDB for parallel query processing and optimization [46]. It pushes jobs into AsterixDB's Hyracks run time as distributed acyclic graphs (DAG). The implementation architecture of AsterixDB consists of a cluster controller, metadata controller, and node controllers [45]. The cluster controller accepts AQL statements pushed from clients (over HTTP), which are distributed as job descriptions to Hyracks data flow engine [47], and node controllers [45].

The first performance results of data ingestion tests indicated that MongoDB had a lower latency with simple write operations, but AsterixDB outperformed MongoDB, when multiple elements were saved in a batch over representational state transfer (REST) API [45]. Recently, data feed management has been integrated as a new architectural component to AsterixDB [14]. The feature enables fault tolerant push/pull-based streaming and pre-processing of semi-structured data from different sources (e.g. Twitter, TCP socket) with user defined functions (UDF) [48]. UDFs enable programming with other languages (e.g. Java) than AQL. Performance results indicate that an integrated solution (Storm + MongoDB) provided lower ingestion performance in comparison to AsterixDB [14]. Performance of AsterixDB has been compared to other big data management systems with a micro-benchmark [49]. Results indicated that AsterixDB achieved in most cases comparable performance to the fastest system.

As streaming technologies have been developed, new tools are needed for their evaluation. OLTP-Bench has been introduced for benchmarking DBMSs, and database-as-a-service systems [50]. It is bundled with different workloads including derivations from other benchmarks [such as Yahoo! cloud serving benchmark (YCSB)]. It also includes a synthetic workload generator for simulating Twitter. Linked Data Benchmark Council (LDBC) has developed a benchmark for simulating social networks [51]. It provides interactive workloads for benchmarking graph-based databases, RDF databases, and relational DBMSs. Big Data generator suite (BDGS) has been developed for generating scalable big data while employing data models derived from real data [10]. The tool includes text, graph, and table generators for simulating real world datasets from

Wikipedia, Facebook, Amazon Movie reviews etc. BDGS has been used in performance testing of DataMPI, Spark, and Hadoop [11]. IoTABench has been developed for evaluating big data systems for internet of things (IoT) [52]. The tool uses a Markov-chain based synthetic data model for data generation. SparkBench covers machine learning, graph computation, SQL queries, and streaming applications for benchmarking Spark [53]. StreamBench is a messaging system between streaming data generation and consumption [9]. It uses web log processing and network traffic monitoring data sets as streaming data sources. Streaming data is transmitted with Apache Kafka to the target technologies under study. A prototype has been developed for benchmarking of platforms in the context of online social networks [54]. The benchmark crawls data from Twitter, and provides several analytical workloads for different data sources.

For this article, Spark streaming has been integrated with Cassandra for persistence, and its performance has been compared to AsterixDB. Processing of tweets has been experimented by simulating Twitter as a social media data source, and by performing analytical queries to the streamed tweets in near real time. The analytical Twitter-related algorithms were content analysis (word count), and sentiment analysis of tweets. Especially, SentiWordNet 3.0 has been implemented with both technologies as a lexicon-based sentiment analysis method [26]. A new test client has been developed for benchmarking of the streaming technologies, because existing tools were either not applicable [10, 50, 51], not publicly available [9, 52, 54], or technology-specific [53]. Also, scalability of the technologies has been analysed on Eucalyptus cloud platform. Similar feasibility analysis of the streaming technologies has not been performed earlier in this context, according to the author's best knowledge.

AsterixDB and Spark streaming for stream processing

AsterixDB and Spark streaming have been compared in Table 1 in terms of stream processing characteristics. Data can be streamed into AsterixDB via REST API or with data feeds from different sources (Twitter, RSS, TCP socket) [14]. Spark streams can be implemented with TCP sockets. Also, built-in adapters are available for other data sources (e.g. Kafka, Twitter etc.). Both technologies provide support for a variety of programming languages for processing of streaming data. Asterix query language (AQL) may be used (over REST API) for data definition and manipulation. Alternatively, other programming languages (e.g. Java) can be used for data processing with UDFs. The main programming language of Spark is Scala, but support is available also for other programming languages (Java, Python, R).

Table 1 Comparison of AsterixDB and Spark streaming for stream processing [29, 34]

	Streaming interfaces	Programming languages	Scalability	Data model	Data access
AsterixDB	REST API, data feeds (tweet, RSS)	AQL, UDF	master–slave cluster	Semi-structured (Asterix data model)	AQL, XQuery, HiveQL
Spark	TCP sockets, Kafka, Flume, Twitter, ZeroMQ, Kinesis	Scala, Java, Python, R	Stand-alone, Apache Mesos, Hadoop YARN	RDD	Spark SQL/data frames. Internally: database APIs or adapters

Different clustering techniques can be used for distributing computing across several nodes. AsterixDB utilizes a master–slave model, whereas Spark has several clustering possibilities. AsterixDB has JavaScript object notation (JSON)-based Asterix data model (ADM) for semi-structured data. Data in Spark is handled as parallelizable RDDs, which may also be persisted in-memory. Data in AsterixDB can be accessed with AQL, whereas Spark provides access with Spark SQL/data frames. Spark doesn't have persistent storage (besides persisting in-memory), but a database may be accessed with native drivers or APIs. One example is Spark Cassandra connector, which provides database access to Cassandra from Spark [34].

Research design and methodology

This work has been motivated by the following factors: (a) Business needs for data increasingly include requirements for near real time analysis [3, 55]. (b) Stream-based data is often semi-structured (e.g. loosely coupled structure such as JSON) instead of following a strict database model [32]. (c) Different technologies have been developed for stream processing of data [32]. Technology selection for a business case may require extensive evaluation, before an optimal decision can be made [56, 57]. This study aims to facilitate decision making process of technology selection by providing new information regarding feasibility. (d) Performance in terms of scalability on a cloud computing platform has not been studied in the context of this study.

Spark streaming and AsterixDB were focused on in this study based on the following rationale. AsterixDB has been developed for processing of semi-structured stream-based data [13]. However, few performance studies are available, where performance of AsterixDB would have been compared with other technologies (an exception is comparison to Storm + MongoDB [14]). On the other hand, Spark has had excellent stream processing performance. However, earlier performance tests [12, 37, 39] have not considered integration between Spark and a database, which may affect performance. In this work, Cassandra has been integrated with Spark for persistence, because it has been linearly scalable [58], and an adapter has been developed for facilitating integration with Spark [34]. Also, performance and scalability of Spark streaming has not been compared against AsterixDB in earlier literature according to the author's best knowledge. The following research questions are posed:

RQ 1: How AsterixDB and Spark streaming + Cassandra perform relative to each other, when semi-structured data is processed as a stream?

RQ 1.1: How the technologies perform, when content analysis and/or sentiment analysis of tweets is performed as a stream?

RQ 1.2: How the technologies perform, when the number of processing nodes is increased on Eucalyptus cloud platform?

This study is based on quantitative research methods. Particularly, processing latency and throughput in stream processing has been measured and compared, when semi-structured data is processed with the technologies under study. See "Methods" chapter for a detailed description of the test procedures.

Test setup

The tests were performed on Eucalyptus v3.4.2 cloud platform (HP Helion Eucalyptus). The platform has been installed on a HW rack (Dell PowerEdge R820), which has 512 GB RAM, 4 Intel Xeon E5 4620 processors (8 cores/processor), and eight hard drives (15 K) configured with redundant array of inexpensive disks (RAID) 0 (striping). The HW could be characterized as a distributed shared memory (DSM) multiprocessor system, where processors' cores communicate via Intel's QuickPath interconnect (QPI) to get access to DSM, and are controlled by an operating system (CentOS 6.5 of Eucalyptus). Performance of the HW for write-intensive services has been tested earlier on Eucalyptus cloud platform [59].

Test setup for experiments has been described in Fig. 1. The setup consisted of a test client, and the technologies under study. The test client was executed in one virtual machine (VM) (80 GB RAM, 12 virtual CPUs). AsterixDB, Spark streaming, and Cassandra were executed on dedicated VMs (40 GB RAM, 6 virtual CPUs). Scalability was experimented up to eight VMs. AsterixDB cluster had one master, and multiple slave nodes. Spark experiments were executed in local and cluster modes. Spark cluster had one Spark master node, and multiple slaves. Spark algorithms were uploaded to Spark

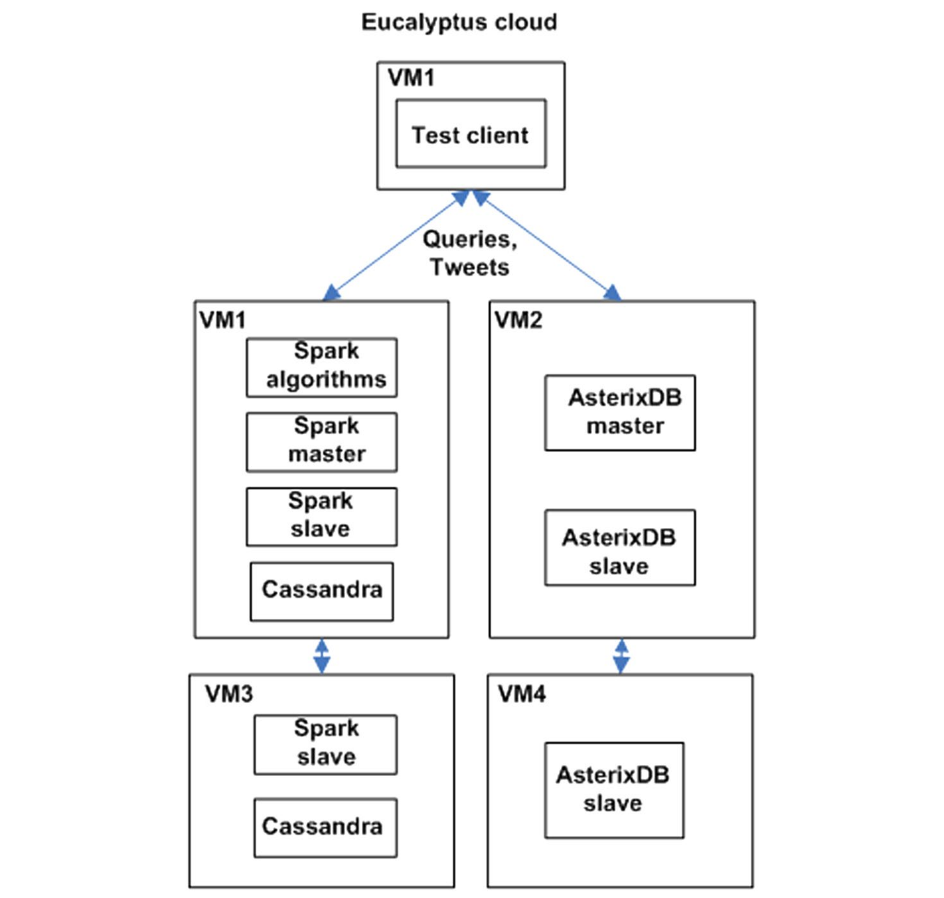

Fig. 1 Test setup for execution of experiments

master for execution of jobs. Cassandra was executed on the same VM with a Spark master/slave.

Algorithms and implementation

Word count and sentiment analysis algorithms were implemented for simulating stream processing of tweets. Word count algorithm calculated frequency of words from the past 50 tweets. Sentiment analysis algorithm was implemented based on SentiWordNet 3.0 [26] as follows. First, words of a stored tweet were extracted. Then, for each word a score (positive, negative) based on SentiWordNet was calculated. For each matching entry in SentiWordNet, the score was summed, and finally the sum was divided by the number of matching entries to get a final score for the word. The final scores of individual words in a tweet were summed together to provide an overall estimation of sentiment in a tweet.

Two implementations of SentiWordNet 3.0 were created. In the 1st version of the algorithm words and associated scores (of SentiWordNet 3.0) were stored into memory as a table [e.g. (words, SentimentScores)]. In the 2nd version of the algorithm, an inverted index was created. An inverted index improves search performance by finding data quickly based on indexed words [60]. The inverted index was comprised of two tables. The first table contained for each word indexed positions pointing to the other table [e.g. (word, index_positions)]. The second table contained sentiment scores for each indexed position [e.g. (index, SentimentScores)]. Thus, sentiment scores could be quickly found based on indexed words of the first table, and indexes of the second table instead of iterating each word in SentiWordNet (1st version of algorithm). The inverted index was created before execution of an experiment.

The word count algorithm and the first version of sentiment analysis algorithm were implemented to AsterixDB with AQL. An inverted index for sentiment analysis was implemented to AsterixDB with Java as a UDF, which was installed with a command line tool of the database. All algorithms were executed based on analytical AQL queries, which were received from the test client. See Additional files 1 and 2 for detailed description of algorithms for AsterixDB (Additional files 1 and 2).

Spark's algorithms were implemented mainly with Scala, but Java was used for connections to Cassandra. Implementation of word count and sentiment analysis algorithms, and tweet loading process were executed in separate Spark processes. In each process, a socket received tweets/query commands from the test client. Tweets were loaded into Cassandra or the requested algorithm was executed based on the received content. See Additional files 3 and 4 for detailed description of algorithms for Spark + Cassandra (Additional files 3 and 4).

Clustering and database access

AsterixDB master–slave architecture was experimented for clustering. With Spark streaming the algorithms were executed both in local mode, and cluster mode (Fig. 2). In the local mode, tweet receiver and both algorithms were executed in Spark driver. In the cluster mode a few modifications were needed for execution of analytical queries. First, a pool was created for storing connections to Cassandra. The driver managed the connection pool, and provided one connection for each worker. Second, sentiment analysis data structures were cached in the driver, in order to avoid costly serialization/

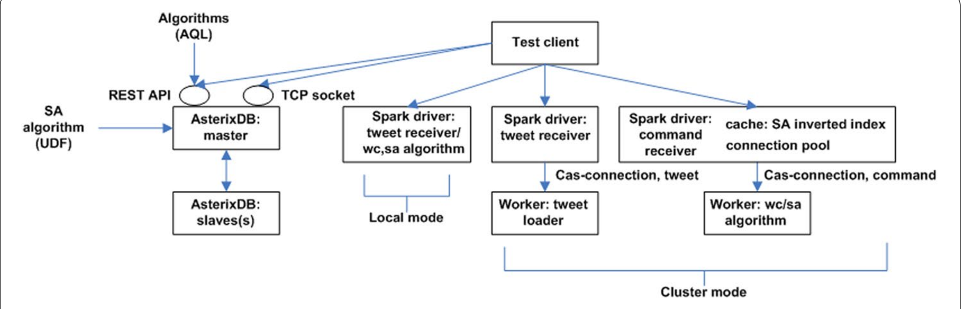

Fig. 2 Implementation architecture of algorithms with AsterixDB and Spark

deserialization between the Spark driver and workers. In practice, caching was implemented with Spark's broadcast variables.

The following fields of a received tweet were saved into a database:

- TwitterUserType: screen-name, language, friends_count, statuses_count, name, followers_count.
- TweetMessageType: tweet_id, sender_location, send_time, message_text, twitter_user_type.

The structure was created for tweets based on Twitter search API [61]. In both databases tweets were indexed based on tweet_id, which was used as the primary key. A tweet data set was created based on TweetMessageType to AsterixDB. One table was created into Cassandra with Spark streaming, which was used for storing of tweets.

Experiments

Several tests were performed in a cloud environment in order to find answers to the research questions. Initially, experiments were conducted separately for streaming and analysis of tweets without stream processing. Also, performance of Cassandra access from Spark was experimented. Subsequently, streaming and analysis of tweets was performed in near real time. The experiments were conducted initially on one cloud node. Finally, the amount of nodes was increased (up to 8 nodes) for studying scalability of the technologies. In the following the experiments are shortly described (Configuration of the experiments is provided in "Test configuration " section).

Preliminary experiment: Tweet loading

The purpose was to find out how fast tweets can be streamed into the cloud platform without processing of tweets. AsterixDB's REST API and data feeds were experimented for data ingestion. Respectively, loading of tweets was implemented to Spark with TCP sockets.

Preliminary experiment: reading from Cassandra with Spark

The purpose was to study reading performance of tweets from Cassandra with Spark. Particularly, Spark access to Cassandra with a native Java API and Spark Cassandra connector [34] were experimented.

Preliminary experiment: Tweet analysis

The purpose was to study performance of analytical processing queries without stream processing. In the experiments performance of word count and sentiment analysis algorithms was tested.

Experiment: stream processing (RQ 1.1)

The purpose was to study performance of tweet ingestion and analytical processing queries in near real time. Also, the effect of Cassandra was studied, by executing stream processing with Spark streaming without a database. In the experiments analytical queries were transmitted adaptively from the test client based on the rate tweets were written into the database.

Experiment: scalability (RQ 1.2)

The purpose of the test was to find out how performance of a technology scales, when the amount of processing nodes in the cloud infrastructure is increased.

Results and analysis

Preliminary experiment: Tweet loading

First, preliminary experiments were executed for evaluation of different APIs/drivers and algorithms (Fig. 3). AsterixDB achieves 11× throughput in comparison to Spark (with Cassandra'a Java driver), when data feeds are utilized for loading of tweets. Spark's performance was significantly lower, when Cassandra Spark connector was used. Lower performance was most probably caused by transformation of tweets (serialization) between Spark/Scala and Cassandra, when the tweets were saved into Cassandra [62]. AsterixDB has low throughput, when tweets are ingested using the REST API.

Preliminary experiment: reading from Cassandra with Spark

Prior to the execution of analytical query experiments, Cassandra Spark connector and Cassandra's Java driver were tested for reading performance (Figs. 4, 5). In the tests data had to be brought from Cassandra to Spark driver for analysis. With Cassandra Spark

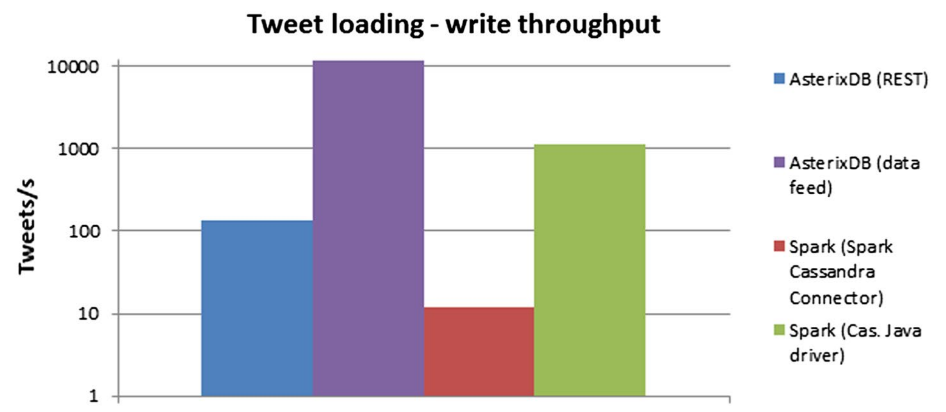

Fig. 3 Tweet loading results. The figure presents throughput, when tweets are loaded to AsterixDB or to Cassandra with Spark

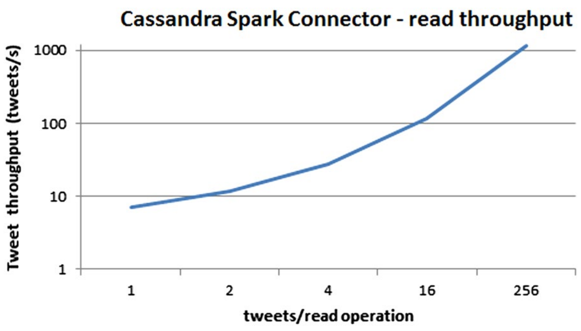

Fig. 4 Reading throughput with Cassandra Spark connector. The amount of tweets per reading operation was varied

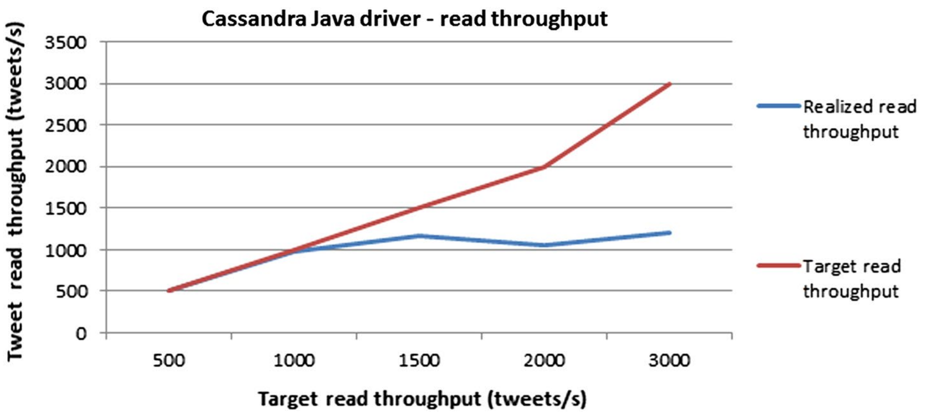

Fig. 5 Reading throughput with Cassandra's Java driver. The target reading throughput was varied

connector data from Cassandra's rows had to be serialized back to Spark's driver, which eventually led to low reading throughput. However, when multiple tweets were fetched with one database read operation (with Cassandra Spark Connector), throughput increased significantly (up to 1000 tweets/s). When data was read directly in Spark with Cassandra's Java driver, serialization wasn't needed, and more than 1100 tweets could be read in a second.

Preliminary experiment: Tweet analysis (word count)

Subsequently, analytical queries were tested. When Cassandra Spark connector was used for calculation of word count among tweets, Spark had lower performance than AsterixDB (REST) (Fig. 6). However, Cassandra's Java driver increased throughput higher in comparison to AsterixDB (with REST API).

Preliminary experiment: Tweet analysis (sentiment analysis)

When 1st version of sentiment analysis was experimented (Fig. 7), Spark achieved ~3× throughput (with Cassandra's Java-driver) in comparison to AsterixDB AQL. When sentiment analysis with an inverted index was used with Spark, throughput was ~1100 tweets/s, which is close to the reading performance of tweets. However, when sentiment

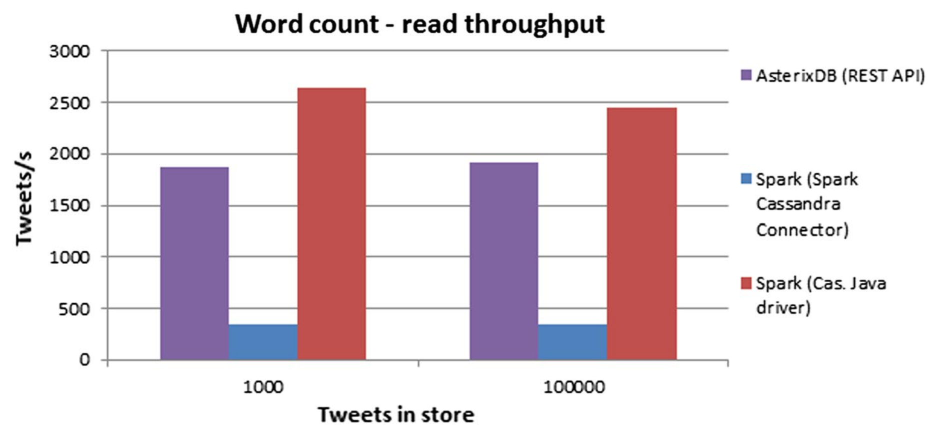

Fig. 6 Reading throughput with word count algorithm

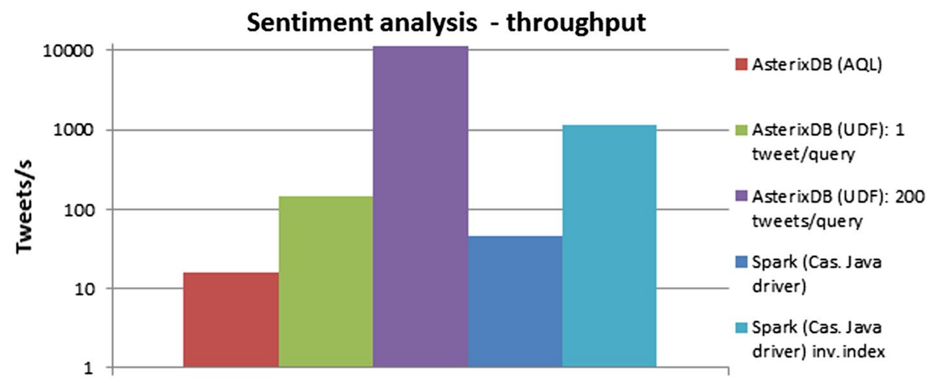

Fig. 7 Reading throughput, when sentiment analysis is performed

analysis algorithm was implemented as a UDF (with an inverted index), AsterixDB had ~11× throughput in comparison to Spark. The increased performance required execution of multiple analysis operations (200) within one request over AsterixDB REST API, in order to avoid REST overhead in communication.

Based on the initial findings related to Cassandra Spark connector, Cassandra's Java driver was utilized in further experiments. It was also decided that AQL wouldn't be used in further tests for loading of tweets nor as an implementation of the sentiment analysis algorithm. Instead, sentiment analysis with an inverted index was used as a UDF in AsterixDB, and a data feed was used for loading of tweets.

Experiment: stream processing (word count)

AsterixDB has 1.5× throughput in comparison to Spark + Cassandra (Fig. 8), when a data feed was used for loading of tweets. AsterixDB can also provide lower latency, when compared to Spark streaming (Fig. 9).

Experiments: stream processing (sentiment analysis)

Figure 10 presents results, when tweets are processed with sentiment analysis algorithms. Spark was tested with two implementations of sentiment analysis algorithms.

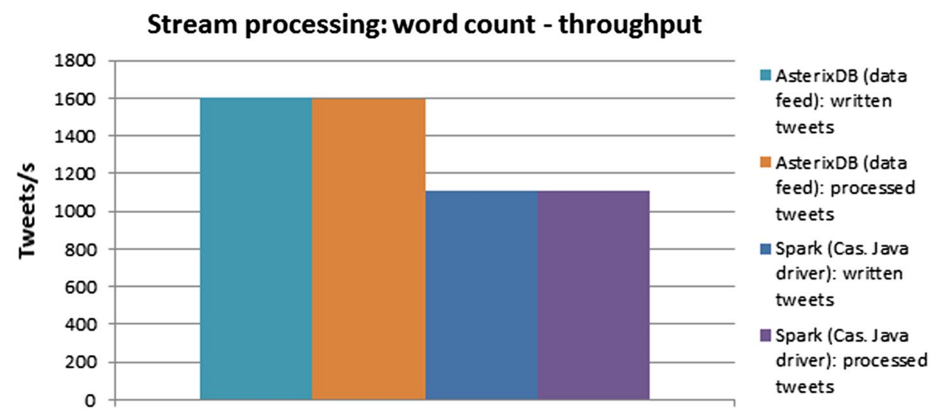

Fig. 8 Stream processing throughput with word count algorithm

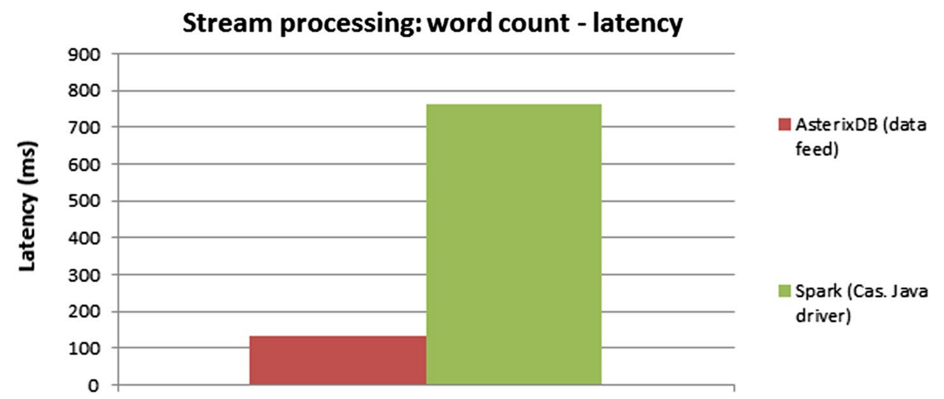

Fig. 9 Stream processing latency with word count algorithm

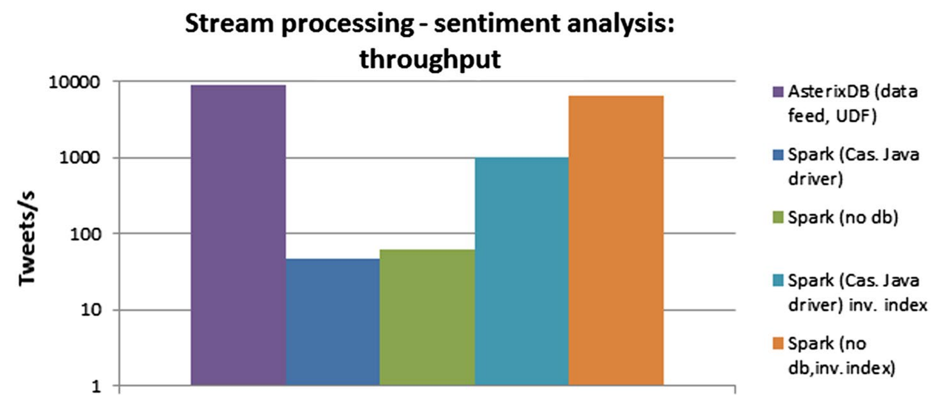

Fig. 10 Stream processing throughput with sentiment analysis algorithm

When an inverted index was created for sentiment analysis, throughput increased 20×. Based on this result, the 2nd version of sentiment analysis algorithm was utilized for comparing AsterixDB and Spark in further stream processing experiments.

The impact of Cassandra can also be seen in the results. Cassandra's impact is largest, when an inverted index is used. AsterixDB was faster than Spark even, when Spark wasn't integrated with a database. AsterixDB provides the best performance, as ~9000 tweets can be processed in a stream, when sentiment analysis with an inverted index has been implemented as a UDF, and tweets are ingested with a data feed.

AsterixDB has significantly lower latency than Spark (Fig. 11). When latency of processing is compared, Spark's analytical queries are queued at the server, and latency is high. However, when no database is utilized, latency is less than 1 s.

Experiment: scalability of word count and sentiment analysis

Performance of the technologies was also compared, when the amount of VMs was increased, in order to study scalability aspects of the technologies (Fig. 12). When nodes were added in word count experiments, performance increased up to 4 nodes with both technologies. AsterixDB has a bit higher throughput with more virtual nodes. In the cluster mode, a pool was utilized for multi-threading of connections to Cassandra (Fig. 2), which most probably led to the increased throughput. When compared to performance achieved with one Spark node (Fig. 10), performance in the cluster experiments is higher.

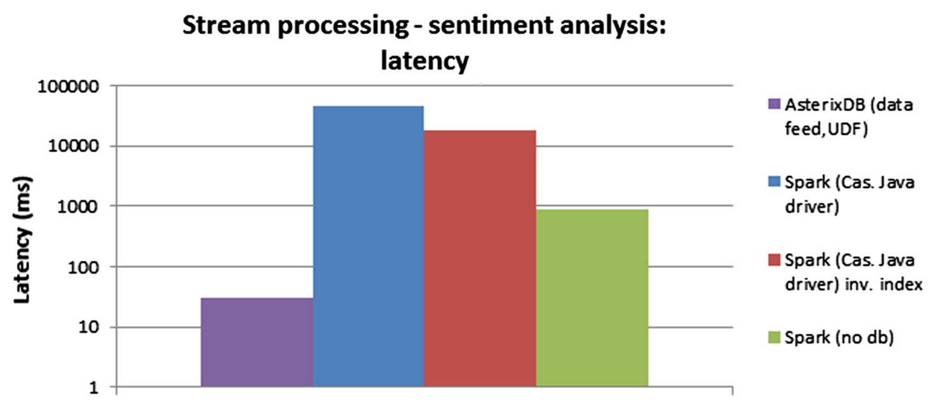

Fig. 11 Stream processing latency with sentiment analysis algorithm

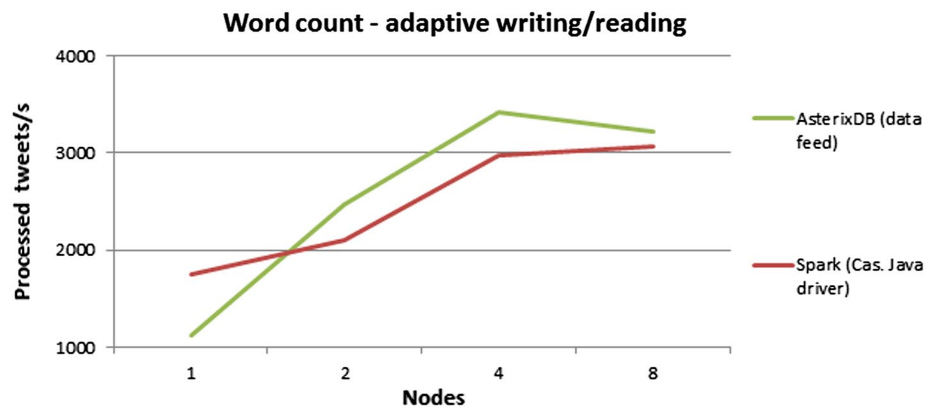

Fig. 12 Scalability of word count algorithm on Eucalyptus cloud plat form

Figure 13 presents scalability of sentiment analysis algorithm on Eucalyptus cloud platform. AsterixDB achieves 3–6× throughput in comparison to Spark + Cassandra. Spark's performance increases up to 4–8 nodes. AsterixDB's performance does not significantly increase with more nodes.

Figure 14 presents scalability results for simultaneous execution of word count and sentiment analysis algorithms. AsterixDB has 1.5–2× higher throughput in comparison to Spark + Cassandra, when the amount of nodes on the cloud platform is increased. It seems tweets are ingested into Cassandra faster than Spark is able to perform sentiment analysis, when the amount of nodes is increased.

Discussion

In the following, lessons learnt from the experiments are discussed. Additionally, the results are compared to earlier literature. Finally, reliability and validity of results, and future work are discussed.

Lessons learnt

In this article stream processing refers to processing after persisting of tweets to the database (near real time processing). In practise, the test client sent analytical queries

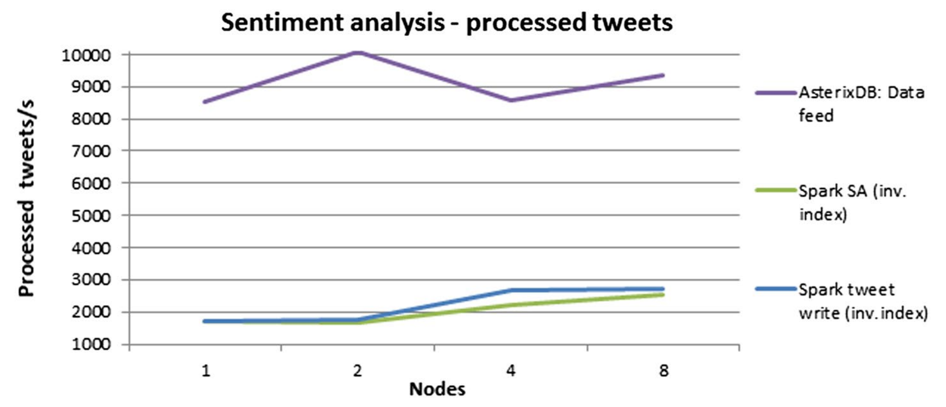

Fig. 13 Scalability of sentiment analysis algorithm on Eucalyptus cloud platform

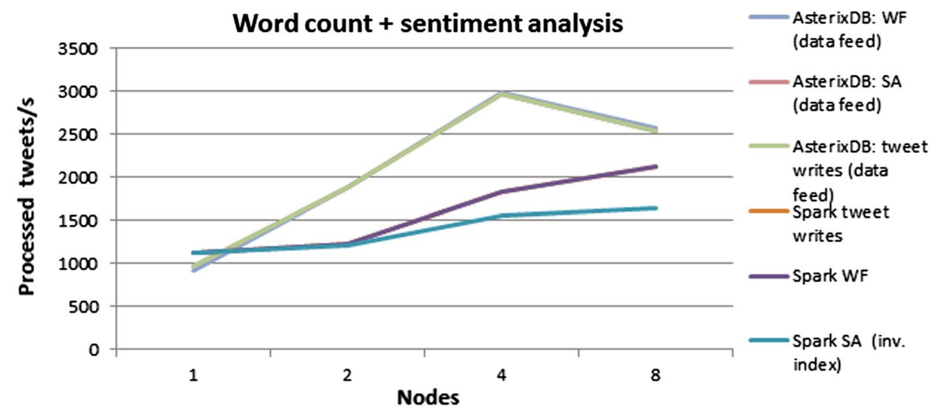

Fig. 14 Scalability of word count and sentiment analysis algorithms on Eucalyptus cloud platform with Spark

adaptively based on the rate tweets were saved to the database. Tweets could also have been processed immediately after reception at the server. The alternative was experimented shortly, when Spark was executed without Cassandra in sentiment analysis (Fig. 10). In this case tweets were processed in Spark after reception, and the result of algorithm(s) was returned to the test client. Real time processing could also have been implemented as a UDF to AsterixDB, by connecting the UDF to the data feed [48]. However, measurement of UDF latency may have been difficult, because the test client wouldn't have received any indication, when the UDF had been completed for streamed tweets.

In the stream processing experiments tweets were read from a database, and algorithm(s) were executed based on tweet content. This approach led to serialization of tweet objects between Spark and Cassandra, when Spark Cassandra connector was utilized, which eventually led to low write/read performance. If algorithms could have been implemented without reading tweets from the database to Spark, Spark Cassandra connector may have had much better performance, and would have been more suitable for the purpose.

Maximum tweet loading throughput with AsterixDB was ~12,000 tweets. When data was loaded, tweets were dropped (not saved into database), when throughput increased above a threshold. In order to guarantee that all loaded tweets were saved correctly, the amount of tweets was checked after each experiment. Different policies may be utilized with data feeds [14]. In the experiments basic policy was used, which leads to buffering of excess records in memory [14]. The effect of other ingestion policies has been studied earlier [14].

As tweets were sent in a socket to AsterixDB feed, transmission was considered completed as soon as data was pushed to the socket. On the contrary, as tweets were saved into Cassandra with Cassandra query language's (CQL) INSERT, the tweet had to be saved to commit log and memtable of at least one Cassandra replica node (consistency level = ONE).

Currently, AsterixDB ingests data within fixed-size frames to the database. The default size of a frame is 128 KB (compiler.framesize in asterix-configuration.xml), which has to become full in order for AsterixDB to save items to the database. Thus, algorithms had to be executed many (1000) tweets behind the last tweet, which was transmitted to AsterixDB. Alternatively, UDF could have been connected with a data feed, which would have enabled execution of UDF immediately after reception of tweets, but would have made UDF measurements difficult.

An inverted index significantly improved performance of sentiment analysis. Sentiment analysis algorithm with an inverted index was created as an UDF with Java, which was executed with calls over the REST API. Multiple analytical queries (200) were made within one REST API call to AsterixDB, in order to avoid REST connection overhead in sentiment analysis (Fig. 7), and for gaining better performance.

The three afore-mentioned issues have to be considered in interpretation of the results. Because tweets cannot be considered saved immediately after transmission, the loading results are more optimistic for AsterixDB. This issue does not affect comparison of stream processing results, because performance was measured based on execution of UDFs (sentiment analysis) or HTTP queries (word count). However, UDF execution had to be delayed in streaming experiments of sentiment analysis, which should be acknowledged in interpretation of the results. Also, in this case multiple analytical UDF queries (200) had to be performed within one HTTP REST API call for achieving the streaming performance results.

When tweets were loaded into Spark/Cassandra, throughput increased significantly, when connections to Cassandra were multi-threaded in the scalability experiments (Figs. 12, 13). Cassandra's connection pool was debugged, which indicated that ~5 threads were used simultaneously. A simple multi-threading client without Spark (with 5 threads) was able to achieve ~6000 writes/s to Cassandra (vs. Fig. 13), which suggests that the current thread pool approach in Spark may be optimized.

The technologies may also be compared in terms of experiences related to programming and deployment of algorithms. The development version of AsterixDB (v0.87) was compiled and installed for development of UDFs and deployment of data feeds. UDF algorithms were compiled in the development environment of AsterixDB, and installed with AsterixDB's management tool. AQL algorithms were installed via the REST API. Library dependencies between Scala, Spark, Cassandra, and Spark Cassandra connector had to be resolved, before algorithms could be compiled and executed with Spark. Spark algorithm deployment differed from AsterixDB is terms of resource allocation for Spark processes (CPU, memory). The most demanding parts (to the author) in programming and deployment of algorithms were learning of the AQL syntax, and functional programming paradigm of Spark/Scala.

Comparison to literature

AsterixDB's data feeds and UDF have been reported to achieve higher ingestion performance than an integrated solution (MongoDB + Storm) [14]. This research came to a similar conclusion, when a different technology set (Spark + Cassandra) was compared to AsterixDB. Reported tweet ingestion throughput per feed was ~10,000 tweet/s [14], which is close to the result achieved in this work (Fig. 3). The earlier study also focused on the impact of feed ingestion policy on performance, which is out of scope for this work. Latency of tweet ingestion over REST API was ~200 ms, which is higher, when compared to earlier insert experiments (~100 ms) [45]. An earlier study focused on experimentation of AsterixDB and three other data management systems with read-only and data modification workloads [49]. However, the experiments differed from this work, which makes comparison of results difficult. The main differences of this work to earlier literature on AsterixDB are stream performance of tweet-related analytical queries, scalability aspects, and comparison to a new integrated solution (Spark and Cassandra).

Earlier Spark experiments have reported sub-second latency for Grep, word count, and TopKCount applications [12]. Other word count experiments with Spark (without database integration) indicated a processing latency of 2–3 s [9], which is higher than the result reported in this paper (Fig. 9). The reported word count latency (in this paper) was on a sub-second range. The difference (to [9]) may be explained with a different data set, and version of Spark (v0.9 vs. v1.3.1). Execution of sentiment analysis with Cassandra led to queueing at Spark server, and much higher latencies (Fig. 11). However, when both algorithms were executed without a database, latency dropped lower than 1 s.

Earlier studies of sentiment analysis have focused mostly on performance comparison between lexical-based and machine learning approaches [42, 43] in terms of prediction accuracy and agreement [22, 24, 31]. Additionally, performance of web services providing sentiment analysis has been studied [23]. Instead, this paper focused on performance of technologies in terms of throughput and latency, when the selected sentiment analysis method (SentiWordNet 3.0) [26] was implemented. Lexicon-based sentiment analysis with MapReduce has been studied in Amazon cloud infrastructure [28]. The achieved throughput (~1000–2000 tweets/s) is in the same range with results of this paper. However, their approach was batch-based, and this paper focused on stream-based analysis. Taghreed [29] can achieve high insertion rates, and low query latency for tweets. Their approach focused on geotagged tweets with spatial and temporal boundaries, which enables implementation of optimizations. Nimbus has integrated Spark with MySQL for providing a filter service for Twitter streams [30], while the approach of this paper focused on real-time processing performance. Twitter sentiment analysis in a Spark cluster indicated that performance improved, when the number of nodes in the cluster was increased [44]. The main difference to this study is that persistence to database was not considered in their study.

Earlier literature was studied for finding a tool, which could be reused for execution of Twitter-related experiments. OLTP-bench [50] has a synthetic workload generator for Twitter data and queries. However, it is aimed for testing DBMSs, and was not easily extendable for experimenting with streaming technologies. StreamBench [9] and a benchmark for social networks [54] could be utilized for testing of streaming applications, but the tools have not been made publicly available. SparkBench [53] is also an interesting alternative, but it is a Spark-specific benchmark. BDGS [10] may be used for creation of semi-structured data (e.g. Amazon movie reviews), but not for simulation of Twitter related data sets. A micro-benchmark has been published [49], which can be used for simulation of social networking applications. The benchmark also aims at benchmarking of multiple big data management systems, but currently lacks streaming functionality by concentrating on read-only or update-only workloads. For these reasons, a new test client was implemented (Fig. 15). Implementation of the testing tool required 1–2 month of working effort. Thus, it would be beneficial to reuse the testing tool in similar experiments. A few SW components of the testing tool (statistics, dictionary, ContentManager) may be reused, if other algorithms or technologies would need to be evaluated. However, communication and actual message protocol (e.g. AsterixDB-Content and AsterixDBImpl in Fig. 15) related to the target technologies would have to be re-implemented.

Reliability and validity of results and future work

Reliability of results may be improved by running longer and more test iterations. However, the experiments included ~180 different setup configurations, which set limits to the amount of time, which was available for experimentation. In this article performance was studied from the point of view of throughput and latency. Additionally, other measures (e.g. CPU/memory consumption at the server) could have been studied [40, 53].

The results can be considered valid for a DSM multiprocessor HW platform, where the HW resources have been virtualized into a cloud environment with Eucalyptus

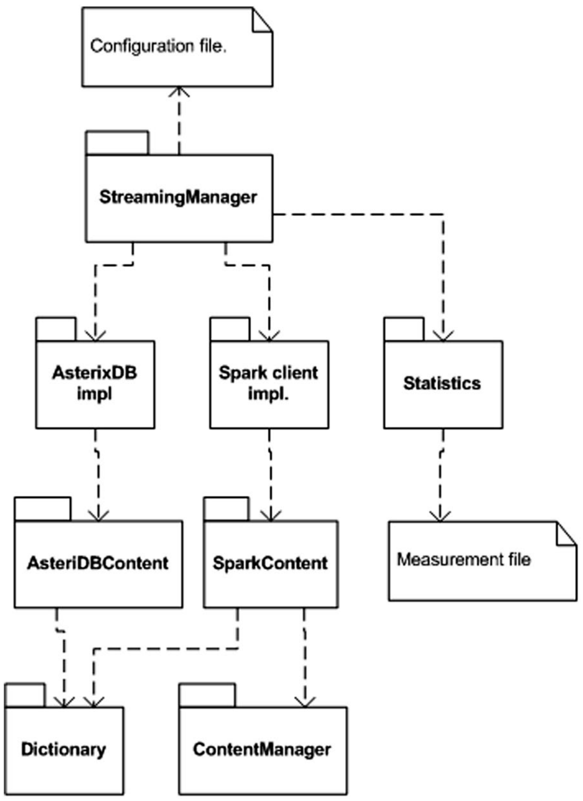

Fig. 15 Architecture of the test client

technology. Alternatively, experiments could have been conducted on a computer cluster, where nodes would have been connected via a high-speed network interface. Disks of the server rack were configured with RAID (level 0), which created a common virtual disk to the cloud environment. Instead, disks could have been configured without RAID, or solid-state drives (SSD) could have been used.

Future work may include feasibility analysis with more complex (CPU and memory-intensive) algorithms (such as [31]). Also, it would be interesting to consider other sentiment analysis methods than SentiWordNet [63], or extend machine learning approach to sentiment analysis (e.g. [28, 44]). Other stream processing technologies (e.g. Storm, S4) with integrated databases could be compared to AsterixDB by reusing the testing tool in experiments. Experiments could also be replicated in an environment with clustered nodes and high-speed disks (SSD).

Conclusions

The main research question was to study how AsterixDB performs relative to Spark streaming, which has been integrated with Cassandra, in the stream processing of semi-structured data. The question was focused on by finding answers to two sub-research questions. The 1st sub-question was to find out how the technologies perform, when content analysis (word count) and sentiment analysis of tweets is performed in a stream. A test bed was created on Eucalyptus cloud platform, for testing the algorithms with both technologies under study. The results indicated that AsterixDB provided higher throughput than Spark + Cassandra in the experimented scenarios. AsterixDB also achieved lower latency. The 2nd research question focused on scalability, when the number of processing nodes on the cloud platform is increased. The results indicated that AsterixDB scaled roughly similarly with Spark streaming in the execution of the word count algorithm. AsterixDB had 3–6× throughput with sentiment analysis, and 1.5–2× throughput, when word count and sentiment analysis were executed simultaneously in a cluster. Based on the experiments it can be concluded that AsterixDB performed relatively better in the processing of semi-structured Twitter data. However, buffering of tweets in a data feed produced more optimistic tweet loading results for AsterixDB, which should be taken into account in interpretation of the results. Also, due to the buffering of tweets with AsterixDB's, the calling of UDFs had to be delayed, which should be taken into account in interpretation of streaming experiments of sentiment analysis. This issue may be improved by writing more frequently to the database (in AsterixDB), which remains to be validated in future work.

Methods

Test client

Architecture of the test client has been described in Fig. 15. The test client simulated generation of tweets and analytical queries, which were executed against the technologies under study. Dictionary produced tweets randomly based on words contained in SentiWordNet 3.0 corpus [26]. Words having no negative or positive meaning were removed from the dictionary (total dictionary size = 15,344 words). Separate tweet data sources (AsterixDBContent/SparkContent) were implemented for producing technology dependent content (tweets, queries) for both technologies under study. Also, separate testers were implemented for Spark (Spark client impl.) streaming and AsterixDB (AsterixDBImpl). Statistics-component was utilized by the technology implementations for reporting events regarding loading of tweets and analytical queries, which were used for calculation of latency and throughput. StreamingManager read test configurations from a file prior to the execution of the tests.

Measurements

Test client was used for performing of measurements as follows. It transmitted tweets and queries to the REST API of AsterixDB in a HTTP POST, and AsterixDB replied with a HTTP 200 OK (details in Figs. 16, 17). AsterixDBImpl recorded a timestamp before sending of the request, and after receiving a reply, and calculated the difference, which was reported to the Statictics-component as latency.

Additionally, data feeds were used for loading of tweets into AsterixDB. Initially, a data feed was created with AQL to AsterixDB. Subsequently, TCP socket at the test client was used for streaming of tweets into AsterixDB. A timestamp was captured before/after transmission of tweets, and the difference was calculated as latency.

Spark streaming measurements were conducted with the Test client, and Spark algorithms (Fig. 1), which were submitted to Spark's standalone cluster manager (in cluster mode) (details in Figs. 16, 17). First, SparkClientImpl captured a timestamp, before a tweet/query had been transmitted to the server. After finalization of a job (write/query),

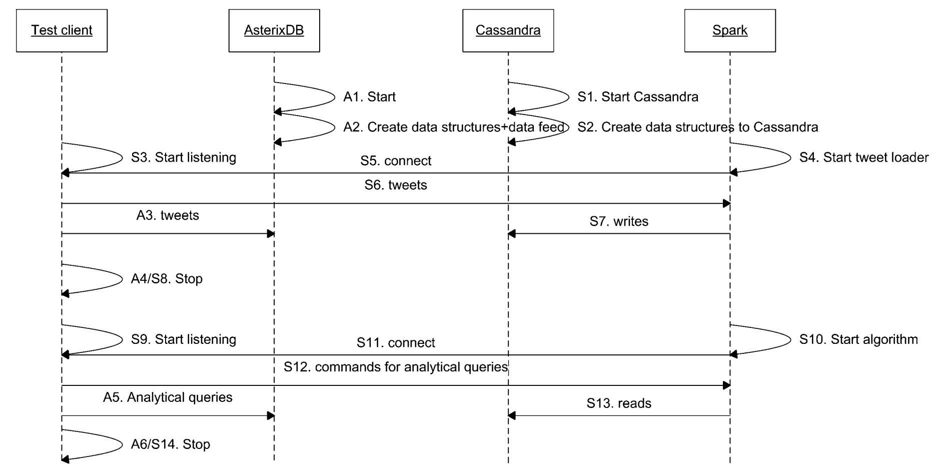

Fig. 16 Communication in tweet analysis experiments. *A* AsterixDB, *S* Spark + Cassandra

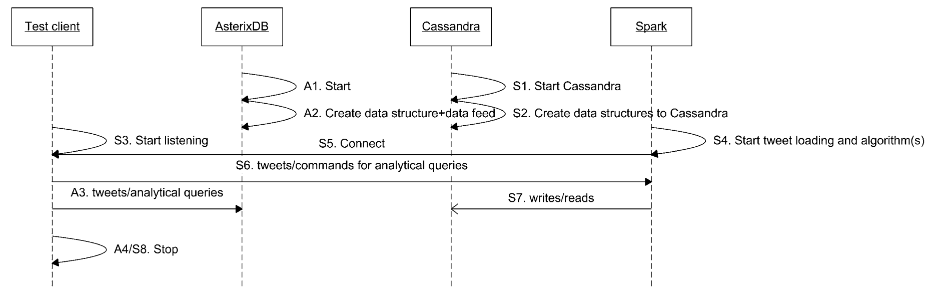
Test client
AsterixDB
Cassandra
Spark
A1. Start
A2. Create data structure+data feed
S1. Start Cassandra
S2. Create data structures to Cassandra
S3. Start listening
S5. Connect
S4. Start tweet loading and algorithm(s)
S6. tweets/commands for analytical queries
A3. tweets/analytical queries
S7. writes/reads
A4/S8. Stop

Fig. 17 Communication in streaming processing experiments

Spark algorithms transmitted a response back to the Test client, which captured a timestamp. Finally, the difference was reported to the statistics-component as latency.

The result of algorithms was transmitted to the test client as a response (AsterixDB: in HTTP 200 OK, Spark: in TCP connection). However, in Spark's cluster mode the result of the algorithms wasn't transmitted to the client, because of the difficulty of fetching the results from the distributed workers to the driver.

In a query test analytical queries were executed for duration of 20 min. In loading and stream processing tests, each test lasted for 200 s + 20 min. The first 200 s was considered as a warm-up phase, which was not included into the measurements.

During each test Statistics-component of the test client measured throughput within a 2 s interval (based on received latency reports), which was saved into a measurement file after completion of a test. The throughput measurements were averaged to get an overall average throughput in a test case. A test case was executed three times in each test configuration. The final throughput is an average of the test case iterations.

Test procedure: Tweet analysis

Figure 16 presents a sequence diagram for describing events in tweet analysis experiments. In AsterixDB experiment, first the database is started (A1). Then, data structures for tweets and a data feed (if used for tweet loading) are created (A2). Tweets are loaded to AsterixDB from the test client (A3), and the test client is stopped (A4). Subsequently, analytical queries are transmitted to AsterixDB via the REST API (A5). Finally, the test client is stopped (A6).

Spark's experiment begins by starting Cassandra at the server node (S1), and by creating data structures to it (S2). Then, the test client starts listening for connections (S3). Spark streaming process is started at the server for loading of tweets (S4), and a connection is established to the test client (S5). Tweets are loaded into Spark + Cassandra from the test client (S6, S7), which is followed by stopping/restarting of the test client (S8, S9). Spark process of an algorithm is started at the server (S10), and a connection is established to the test client (S11). Finally, commands for analytical queries are transmitted from the test client to the Spark server, which executes analytical jobs (S12, S13). Finally, the test client is stopped, which outputs the test results into a file (S14/A6).

Test procedure: stream processing

Communication in the stream processing experiments has been described in Fig. 17. Initially, AsterixDB is started (A1), data structures, and data feed (if used for loading of tweets) are created (A2). Then, tweets and analytical queries are transmitted simultaneously from the test client to the server (A3).

In Spark experiments, first Cassandra is started (S1), and data structures are created (S2). Subsequently, listening is started at the test client (S3). Then, Spark processes (tweet loader, algorithms) are started at the server (S4), and connections are established to the test client (S5). The test client transmits tweets and commands for analytical queries simultaneously to Spark processes (S6, S7). Finally, the test client is stopped, and results are output to a measurement file (A4/S8).

Test configuration

AsterixDB v0.86 was used for tweet loading experiments over REST API, and for testing implementation of algorithms with AQL. The development version (v0.87) of AsterixDB was used for experimentation with data feeds and UDF. Spark v1.3.1 was used with Cassandra 2.1.2. Spark Cassandra connector v1.3.0 was used for connections to Cassandra.

In the following technology specific configurations of the experiments are provided:

- Spark: block interval = 50 ms.
- Spark: memory allocation = 30 GB per worker/VM (1 node experiment), memory was divided evenly to Spark processes [tweet receiver, algorithms (Fig. 1)] in cluster experiments.
- Spark: CPU allocation: all virtual cores for VM (1 node experiment), virtual CPUs were divided evenly to Spark processes in cluster experiments.
- Cassandra. Default consistency level for reads/writes = ONE.
- Test client

- Spark: one thread for analytical command queries, one thread for loading of tweets.
- AsterixDB: 10 (1 node tests)/20 (clustering tests) threads for transmission of analytical commands. Thirty threads for loading of tweets over REST API. One thread for loading of tweets with a data feed.

Additional files

Additional file 1. Word count algorithm for AsterixDB: The file contains AQL-based word count algorithm definition for AsterixDB. An example illustrates how word count algorithm is called from Test client by using AsterixDB's REST API.
Additional file 2. Sentiment analysis algorithm for AsterixDB: The file contains Java-based UDF implementation of sentiment analysis algorithm with an inverted index. An example illustrates how sentiment analysis algorithm is called from Test client by using AsterixDB's REST API.
Additional file 3. Word count algorithm for Spark + Cassandra: The file contains Scala/Java-based implementation of word count algorithm for Spark + Cassandra. An example illustrates how word count algorithm is called from the Test client.
Additional file 4. Sentiment analysis algorithm for: The file contains Scala/Java-based implementation of sentiment analysis with an inverted index for Spark + Cassandra. An example illustrates how sentiment analysis algorithm is called from the Test client.

Abbreviations
ADM: Asterix data model; AQL: Asterix query language; BDGS: Big Data generator suite; CQL: Cassandra query language; DBMS: database management system; DSM: distributed shared memory; IMDb: internet movie database; IoT: internet of things; JSON: JavaScript object notation; LDBC: Linked Data Benchmark Council; QPI: QuickPath interconnect; RAID: redundant array of inexpensive disks; REST: representational state transfer; RDD: resilient distributed datasets; RDF: resource description framework; SPARQL: SPARQL protocol and RDF query language; SSD: solid-state drive; UDF: user defined functions; VM: virtual machine; YCSB: Yahoo! cloud serving benchmark.

Acknowledgements
This research has been carried out in Digile Need for Speed program, and it has been partially funded by Tekes (the Finnish Funding Agency for Technology and Innovation).
Additionally, the author acknowledges feedback received from subscribers of Apache AsterixDB users mailing list, and DataStax's Spark connector for Cassandra mailing list.

Competing interests
The author has no competing interests.

Received: 11 December 2015 Accepted: 30 March 2016
Published online: 08 April 2016

References
1. Thusoo A et al. Data warehousing and analytics infrastructure at Facebook. Paper presented at the ACM SIGMOD international conference on management of data, Indianapolis, Indiana, USA, 6–11 June 2010.
2. Sumbaly R, Kreps J, Shah S. The "Big Data" ecosystem at LinkedIn. Paper presented at the ACM SIGMOD international conference on management of data, New York, New York, USA, 22–27 June 2013.
3. Mishne G, Dalton J, Li Z, Sharma A, Lin J. Fast Data in the era of Big Data: Twitter's real-time related query suggestion architecture. Paper presented at the ACM SIGMOD international conference on management of data, New York, New York, USA, 22–27 June 2013.
4. Busch M et al. EarlyBird: real-time search at Twitter. Paper presented at the IEEE 28th international conference on data engineering, Washington, DC, USA, 1–5 April 2012.
5. Kulkarni S et al. Twitter heron: stream processing at scale. Paper presented at SIGMOD 2015, Melbourne, Victoria, Australia, 31 May–4 June 2015.
6. Goonetilleke O, Sellis T, Zhang X, Sathe S. Twitter analytics: a Big Data management perspective. SIGKDD Explor. 2014;16:11–9. doi:10.1145/2674026.2674029.
7. Zubiaga A, Spina D, Martinez R, Fresno V. Real-time classification of Twitter trends. J Assoc Inf Sci Tech. 2015;66:462–73. doi:10.1002/asi.23186.
8. Proferes MFNJ. A topology of Twitter research: disciplines, methods, and ethics. Aslib J Inf Manag. 2014;66:250–61.
9. Lu R, Wu G, Xie B, Hu J. StreamBench: towards benchmarking modern distributed stream computing frameworks. Paper presented at the IEEE/ACM 7th international conference on utility and cloud computing, London, Great Britain, 8–11 December 2014.
10. Ming Z, Luo C, Gao W, Han R, Yang Q, Wang L, Zhan J. BDGS: a scalable Big Data generator suite in Big Data benchmarking. Lectures notes in computer science, vol. 8585. Switzerland: Springer; 2014. p. 138–54.

11. Liang F, Feng C, Lu X, Xu Z. Performance benefits of DataMPI: a case study with BigDataBench. Lecture notes in computer science, vol. 8807. Switzerland: Springer; 2014. p. 111–23.
12. Zaharia M, Das T, Li H, Hunter T, Shenker S, Stoica I. Discretized streams: fault-tolerant streaming computation at scale. Paper presented at the 24th ACM symposium on operating systems principles, Farmington, Pennsylvania, USA, 3–6 November 2013.
13. Borkar V, Carey MJ, Li C. Inside "Big Data management": Ogres, Onions, or Parfaits? Paper presented at the EDBT/ICDT 2012 joint conference, Berlin, Germany, 26–30 March 2012.
14. Grover R, Carey MJ. Data ingestion in AsterixDB. Paper presented at the 18th international conference on extending database technology. Brussels, Belgium, 23–27 March 2015.
15. He W, Zha S, Li L. Social media competitive analysis and text mining: a case study in the pizza industry. Int J Inf Manage. 2013;33:464–72.
16. He W, Wu H, Yan G, Akula V, Shen J. A novel social media competitive analytics framework with sentiment benchmarks. Inf Manag. 2015;52:801–12.
17. Chae B. Insights from hashtag #supplychain and Twitter analytics: considering Twitter and Twitter data for supply chain practise and research. Int J Prod Econ. 2015;165:247–59.
18. Bello-Orgaz G, Jung JJ, Camacho D. Social big data: recent achievements and new challenges. Inf Fusion. 2016;28:45–59.
19. Feldman R. Techniques and applications for sentiment analysis. Commun ACM. 2013;56:82–9. doi:10.1145/2436256.2436274.
20. Medhat W, Hassan A, Korashy H. Sentiment analysis algorithms and applications: a survey. Ain Shams Eng J. 2014;5:1093–113.
21. Abbasi A, Hassan A, Dhar M. Benchmarking Twitter sentiment analysis tools. Paper presented at the 9th international conference on language resources and evaluation, Reykjavik, Iceland, 26–31 May 2014.
22. Rosenthal S. SemEval-2015 Task 10: sentiment analysis in Twitter. Paper presented at the 9th international workshop on semantic evaluation, Denver, Colorado, USA; 4–5 June 2015.
23. Serrano-Guerrero J, Olivas JA, Romero FP, Herrera-Viedma E. Sentiment analysis: a review and comparative analysis of web services. Inf Sci. 2015;311:18–38.
24. Gonçalves P, Araújo M, Benevenuto F, Cha M. Comparing and combining sentiment analysis methods. Paper presented at the conference on online social networks, Boston, MA, USA, 7–8 October 2013.
25. Esuli A, Sebastiani F. SentiWordNet: a publicly available lexical resource for opinion mining. Paper presented at the 5th conference on language technology conference, Genova, Italy, 24–26 May 2006.
26. Baccianella S, Esuli A, Sebastiani F. SentiWordNet 3.0: an enhanced lexical resource for sentiment analysis and opinion mining. Paper presented at the 7th international conference on language resources and evaluation, Malta, 17–23 May 2010.
27. Mendes PN, Passant A, Kapanipathi P. Twarql: tapping into the wisdom of the crowd. Paper presented at the 6th international conference on semantic systems, Graz, Austria, 1–3 September 2010.
28. Khuc VN, Shivade C, Ramnath R, Ramanathan J. Towards building large-scale distributed systems for Twitter sentiment analysis. Symposium on applied computing, Riva del Garda, Italy, 26–30 March 2012.
29. Magdy A, Alarabi L, Al-Harthi S, Musleh M, Ghanem TM, Ghani S, Mokbel MF. Taghreed: a system for querying, analyzing, and visualizing Geotagged Microblogs. Paper presented at 22nd international conference on advances in geographic information systems, Dallas, Texas, USA, 4–7 November 2014.
30. Lai C, Donahue J, Musaev A, Pu C. Nimbus: tuning filters service on Tweet streams. Paper presented at the IEEE international congress on Big Data, New York, USA, 27 June–2 July 2015.
31. Fang X, Zhan J. Sentiment analysis using product review data. J Big Data. 2015. doi:10.1186/s40537-015-0015-2.
32. Pääkkönen P, Pakkala D. Reference architecture and classification of technologies, products and services for big data systems. Big data Res. 2015;2:166–86. doi:10.1016/j.bdr.2015.01.001.
33. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Paper presented at the 9th USENIX conference on networked systems design and implementation. San Jose, California, USA, 25–27 April 2012.
34. Spark-Cassandra-Connector. 2015. https://github.com/datastax/spark-cassandra-connector. Accessed 17 Sep 2015.
35. Xin RS, Rosen J, Zaharia M, Franklin MJ, Shenker S, Stoica I. Shark: SQL and rich analytics at scale. Paper presented at the SIGMOD 2013, New York, New York, USA, 22–27 June 2013.
36. Xin RS. Shark, Spark SQL, Hive on Spark, and the future of SQL on Spark. In: Databricks blog. 2014. https://databricks.com/blog/2014/07/01/shark-spark-sql-hive-on-spark-and-the-future-of-sql-on-spark.html. Accessed 10 Aug 2015.
37. Armbrust M, Das T, Davidson A, Ghodsi A, Or A, Rosen J, Stoica I, Wendell P, Xin R, Zaharia M. Scaling Spark in the real world: performance and usability. Paper presented at the 41st international conference on very large data bases, Kohala Coast, Hawaii, USA, 31 August–4 September 2015.
38. Xin R, Wendell P. Announcing Spark 1.5. In: Databricks blog. 2015. https://databricks.com/blog/2015/09/09/announcing-spark-1-5.html. Accessed 20 Oct 2015.
39. Spangenberg N, Roth M, Franczyk. Evaluating new approaches of Big Data analytics frameworks Lecture notes in business information processing, vol. 208. Switzerland: Springer; 2015. p. 28–37.
40. Ousterhout K, Rasti R, Ratnasamy S, Shenker S, Chun B. Making sense of performance in data analytics frameworks. Paper presented at the 12th USENIX symposium on networked systems design and implementation, Oakland, California, USA, 4–6 May 2015.
41. Landset S, Khoshgoftaar TM, Richter AN, Hasanin T. A survey of open source tools for machine learning with big data in the Hadoop ecosystem. J Big Data. 2015;2:24.
42. Zheng J, Dagnino A (2014) An initial study of predictive machine learning analytics on large volumes of historical data for power system applications. Paper presented at the 2014 IEEE international conference on Big Data, Washington, DC, USA, 27–30 October 2014.
43. Bhuvan MS et al. (2015) Semantic sentiment analysis using context specific grammar. Paper presented at international conference on computing, communication and automation, Uttar Pradesh, India, 15–16 May 2015.

44. Nodarakis N, Sioutas S, Tsakalidis A, Tzimas G. Large scale sentiment analysis on Twitter with Spark. Paper presented at the 1st international workshop on multi-engine data analytics, Bordeaux, France, 15 March 2016.
45. Alsubaiee S et al. AsterixDB: a scalable, open source DBMS. Paper presented at the 40st international conference on very large data bases, Hangzhou, China, 1–5 September 2014.
46. Borkar V et al. Algebricks: a data model-agnostic compiler backend for Big Data languages. Paper presented at the ACM symposium on cloud computing, Kohala Coast, Hawaii, USA, 27–29 August 2015.
47. Borkar V, Carey M, Grover R, Onose N, Vernica R. Hyracks: a flexible and extensible foundation for data-intensive computing. Paper presented at the 27th international conference on data engineering, Hannover, Germany, 11–16 April 2011.
48. AsterixDB. Apache Incubator. 2015. https://asterix-jenkins.ics.uci.edu/job/asterix-test-full/site/asterix-doc/index.html. Accessed 20 Oct 2015.
49. Pirzadeh P, Carey MJ, Westmann T. BigFun: a performance study of big data management system functionality. Paper presented at the 2015 IEEE international conference on Big Data, Santa Clara, California, USA, 29 October–1 November 2015.
50. Difallah DE, Pavlo A, Curino C, Cudre-Mauroux P. OLTP-Bench: an extensible Testbed for benchmarking relational databases. Paper presented at the 39th international conference on very large data bases, Riva del Carda, Italy, 26–30 August 2013.
51. Erling O et al. The LDBC social network benchmark: interactive workload. Paper presented at SIGMOD, Melbourne, Australia, 31 May–04 June 2015.
52. Arlitt M, Marwah M, Bellala G, Shah A, Healey J, Vandiver B. IoTAbench: an internet of things analytics benchmark. Paper presented at the 6th ACM/SPEC international conference on performance engineering, Austin, Texas, USA, 31 January–4 February 2015.
53. Li M, Tan J, Wang Y, Zhang L, Salapura V. SparkBench: a comprehensive benchmarking suite for in memory data analytic platform Spark. Paper presented at the ACM international conference on computing frontiers, Ischia, Italy, 18–21 May 2015.
54. Zhang R, Manotas I, Li M, Hildebrand D. Towards a Big Data benchmarking and demonstration suite for the online social network era with realistic workloads and live data. Lectures notes in computer science, vol. 9495. Switzerland: Springer; 2016. p. 25–36.
55. Braun L et al. Analytics in motion. Paper presented at SIGMOD 2015, Melbourne, Victoria, Australia, 31 May–4 June 2015.
56. Lourenço JR, Cabral B, Carreiro P, Vieira M, Bernardino J. Choosing the right NoSQL database for the job: a quality attribute evaluation. J Big Data. 2015;2:18. doi:10.1186/s40537-015-0025-0.
57. Klein J et al. Performance evaluation of NoSQL databases: a case study. Paper presented at the 1st workshop on performance analysis of Big Data systems, Austin, Texas, USA, 31 January–4 February 2015.
58. Rabl T et al. Solving Big Data challenges for enterprise application performance management. Paper presented at the 38th international conference on very large data bases, Istanbul, Turkey, 27–31 August 2012.
59. Pääkkönen P, Pakkala D. The implications of disk-based RAID and virtualization for write-intensive services. Paper presented at the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, 13–17 April 2015.
60. Black EB. "inverted index", in dictionary of algorithms and data structures. 2008. https://xlinux.nist.gov/dads//HTML/invertedIndex.html. Accessed 18 Jan 2016.
61. Twitter API. Tweets. 2015 https://dev.twitter.com/overview/api/tweets. Accessed 13 Nov 2015.
62. Kolaczkowski P. Lightning Fast Cluster Computing with Cassandra and Spark. London: Code Mesh; 2014.
63. Koto F, Adriani M. A comparative study on Twitter sentiment analysis: which features are good? Lectures notes in computer science, vol. 9103. Switzerland: Springer; 2015. p. 453–7.