Interactive querying and data visualization for abuse detection in social network sites

Leandro Ordonez-Ante, Thomas Vanhove, Gregory Van Seghbroek, Tim Wauters, Filip De Turck
Department of Information Technology
Internet Based Communication Networks and Services (IBCN)
Ghent University - iMinds
Technologiepark-Zwijnaarde 15, Gent, Belgium
Email: leandro.ordonez@intec.ugent.be

Abstract—**Big Data technologies have traditionally operated in an offline setting, collecting large batches of information on clusters of commodity machines and performing complex and time-consuming computations over it. While frameworks following this approach served well for most applications involving big data analysis during the last decade, other use cases have recently emerged posing challenging requirements on latency and demanding real-time data processing, querying and visualization. That is the case for applications aiming at detecting threatening behaviors in social network platforms, where timely action is required to avoid adverse consequences. In this sense, more and more attention has been drawn towards online data processing systems claiming to address the limitations of batch-oriented frameworks. This paper reports a work in progress on distributed data processing for enabling low-latency querying over big data sets. Two software architectures are discussed for addressing the problem and an experimental evaluation is performed on a proof of concept implementation showing how an approach based on query pre-processing and stateful distributed stream computation can meet the requirements for supporting interactive querying on large and continuously generated data.**

I. Introduction

One of the main challenges of Big Data is to provide organizations with the ability to deal with large volumes of data being produced at high rates and in a variety of formats. Many research and development efforts have been aimed at conceiving mechanisms enabling high throughput data processing and efficient querying while dealing with constraints of consistency, availability and partition tolerance inherent to distributed computing systems [1].

Moreover, access latency requirements posed by data-intensive applications are more and more demanding by the day. Being able to process and access data as it is available, is becoming of paramount importance for the competitiveness of data-driven business in a plethora of application domains including social media, IoT and sensor data, fraud detection, and networking and security. Consider for instance, the AMiCA project [2], which is intended to identify possibly threatening situations on social networks by means of text and image analysis. Applications meeting the purpose of this project requires efficient methods to process, query and visualize the massive amount of continuously generated data in these social platforms, to effectively spot harming behaviors and mitigate their impact in a timely way.

In spite of the clear need for mechanisms and tools supporting near real-time query and visualization of continuously-generated data, this is still a largely open Big Data problem where three main requirements are to be met, namely: (*i*) high throughput resolution of arbitrary queries, enabling data analysis and information retrieval on an as-needed basis, (*ii*) deal with heterogeneous data sources, since data is regularly available in a myriad of structured and unstructured formats, and (*iii*) strict latency demands. Multiple approaches claim to address this problem, ranging from open source projects working on top of widespread Big Data frameworks like Hadoop [3] (e.g. Apache Drill [8] and Apache Impala [9]), to complete technology-agnostic software architecture patterns such as the Lambda [4] and Kappa [5] Architectures.

This paper explores two approaches for addressing this problem. The first one involves using a distributed SQL engine for supporting ad-hoc querying on top of a big data set, while the second one takes on the flaws and limitations of the previous approach by establishing a clear separation between data storage and query treatment, and also by pre-processing the raw data set in a streaming setting to generate materialized views for answering specific queries on the available data.

The remainder of this paper is organized as follows: Section 2 describes the related works. Section 3 illustrates a use case for applying and evaluating the previously mentioned approaches. Sections 4 and 5 elaborate on each of the proposed approaches specifying their features and limitations. Section 6 deals with the experimental setup and results. Finally conclusions and pointers towards future work are provided in Section 8.

II. Related Work

Multiple approaches currently claim to address the latency limitations of traditional batch processing frameworks. Platforms like Apache Drill [8] and Apache Impala [9] circumvent MapReduce by using a custom query execution engine. These projects are loosely based on Google's Dremel [10] (or BigQuery as it is commercially known), both claiming to have interactive performance and supporting ad-hoc queries. Besides the SQL primitives, Drill and Impala support the creation of user-defined functions allowing the user to conduct some more complex processing on the data beyond projection, aggregation and grouping. According to [12], while the response time of these platforms outperforms those from batch processing frameworks like Apache Hive [11], they still cannot guarantee low latency query resolution on their own [13].

There are also tools tailored for ad-hoc querying and data visualization such as Tableau [14] and Pentaho [15]. However the processing power of these tools is often limited to basic ETL (*Extract, Transform and Load*) and aggregation operations, resorting to third party platforms for performing more elaborated procedures on the data, such as streaming analysis.

Other frameworks tackle the stated problem by relying on distributed streaming computation systems. Two of the most representative implementations of this kind of frameworks are the open source projects Apache Samza [16] and Apache Flink [17]. Apache Samza runs on top of Apache Hadoop YARN [18] for resource management and fault tolerance, and relies on a distributed message broker (Apache Kafka [19] commonly) as canonical storage. A job running on this framework consumes the stream of entries available in the message broker, then performs arbitrary transformations on each incoming record and finally pushes the result back into an output stream for further processing. Apache Flink meanwhile, is more flexible in terms of the data sources it can use, and is also able to perform batch processing as a special case of stream processing (i.e. bounded stream).

Both Samza and Flink acknowledge that a major part of the streaming applications are stateful, hence they provide a mechanism for performing incremental computation and maintaining the state of a streaming job in a distributed key/value store. Managing the state this way entails an evident limitation: as these stores require fine-grained access to key/value pairs to do random reads/writes they are not suitable for complex queries involving filtering, scans and aggregations.

The proposed approach also relies on a distributed streaming processing engine and allows for stateful computation over a continuous data stream. The state in this case holds pre-computed results for arbitrarily complex queries as materialized views. Such views are then indexed and placed into custom data stores, serving as back-end for visualization applications and further processing. This way the approach herein supports interactive querying (i.e. $latency \leq 0.15s$, according to [20]) and provides partial query results over streamed data sets.

III. Use Case

Social networking and social media platforms have enabled people to be more connected than ever before. Users of these sites are allowed to create their own content and share it with their fellow peers, build communities around common interests and even establish direct communication with celebrities, companies and organizations they like or follow.

Along with the increasing popularity of these platforms there is an exploding amount of data being continuously collected about their users. Analyzing this data is key for offering a rich and personalized user experience, and lately also has proven valuable for conducting public health monitoring [21] and spotting abusive or threatening behaviors like cyber-bullying, depression and suicidality [2]. In application domains like this, being able to query, process and visualize data as soon as it becomes available is of capital importance, so that action can be taken on time to avoid or mitigate adverse consequences.

Two alternative methods have been explored to address this problem in this particular domain:

(i) Using a distributed SQL Query Engine on top of the main data set for supporting ad-hoc querying.

(ii) Using a distributed streaming computation system able to perform incremental operations on the available data and store the outcome as materialized views, tailored to answer specific and arbitrarily complex queries with low latency.

In order to evaluate these approaches, a bulk of submissions made by the users of Reddit[1]—a highly popular social news and bookmarking service—was used. The corpus (available at [22]) comprises the contributions made by users from January 2006 to August 2015, adding up to 269 GB of data. Based on the information available in the corpus and for proof of concept purposes, two basic text analysis jobs were run on it: (*i*) a simple per-submission sentiment analysis operation based on the implementation available at [23] using the *Vader Sentiment Analysis Lexicon* [24], and (*ii*) a data set-wide term-frequency (TF) analysis.

The next sections describe the two mentioned approaches, starting with an introduction to their architecture and components and then detailing their actual implementation.

IV. Distributed SQL query engine approach

This approach represents the most straightforward and widely used method to enable querying against heterogeneous big data sets. It harnesses the processing capabilities of a distributed SQL query engine running on top of the main data set, allowing front-end clients to issue ad-hoc queries of arbitrary complexity, by using standard SQL statements. These engines usually support the definition of user-defined functions (UDFs) enabling them to conduct complex processing on the data beyond canonical SQL operations.

A. Architecture

Two main components fit together to compose the architecture of this approach: a distributed SQL query engine, and the main distributed data storage.

A distributed SQL engine is able to break complex queries into small operations and spread them over a cluster of commodity machines where execution units are deployed. Some of these engines run on top of well-known distributed computing frameworks like Hadoop's MapReduce, while others implement their own execution environment. In both cases the engine performs a series of generic steps: (*i*) query parsing and optimization, (*ii*) generating an execution plan,

[1] Available at https://www.reddit.com

(*iii*) submitting the plan to the execution environment, and (*iv*) returning the results back to the client. In general these engines operate in a batch setting. Consequently, they are prone to high latency whenever non-trivial queries (e.g. those involving execution of UDFs, aggregations and joins) are issued by client applications.

The second component of this approach is the main data storage, which commonly lies on a non-relational data store, i.e. distributed file systems, NoSQL and cloud storage, or might also be the output of an event/message source supplying an unbounded data stream. Figure 1 outlines an overview of this architecture.

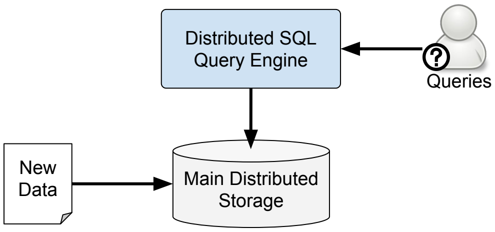

Fig. 1. Overview of the distributed SQL query engine approach

B. Technology mapping for evaluation

The actual implementation of the above architecture involved choosing one of the distributed SQL query engines publicly available. The open source Apache Drill project was selected in this case since it offers several appealing features like:

1) Support for multiple datastores (Hadoop, MongoDB, HBase, cloud storage providers) and file formats (Parquet, JSON, CSV, TSV, PSV)
2) On-the-fly schema discovery
3) Extensive client support through ODBC and JDBC drivers, HTTP API, and Java and C libraries
4) Support the creation of tables, views and UDFs

By harnessing the UDF capabilities of Apache Drill a custom operator was defined for running sentiment analysis on each of the Reddit submissions available in the data set.

As for the main data storage, an HDFS [25] cluster was deployed for storing the Reddit submissions bulk file, with nine (9) Drill execution units configured to run on top of it. Figure 2 illustrates how the mentioned technologies were mapped to the components of this approach architecture.

V. Stateful stream processing approach

The second approach comprises the use of a distributed stream computation system supporting the incremental processing of the input data set (stateful transformations). The rationale behind this, is to set up a system able to run continuous queries, namely those producing new results as new data arrives at the stream processing engine. This way, the system is able to provide partial results on the processed data, avoiding the typical latency of batch processing frameworks.

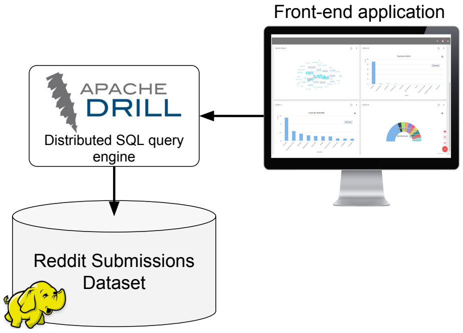

Fig. 2. Implementation of the distributed SQL query engine approach

This approach is based on the software architectural pattern conceived by Kreps known as Kappa Architecture [5], which encourages the clean separation between query processing and data storage, also enables the use of a single code base for specifying the data transformations, and allows for the entire reprocessing of the input dataset to be conducted when required (e.g., when changes are made in the stream processing code base).

A. Architecture

When conceived, the Kappa Architecture was to provide an alternative to the Lambda Architecture [4], by enabling reprocessing of the input data set on eventual code changes but escaping the need for having a batch processing layer. In that sense, this architecture relies entirely on a distributed stream processing system, using a single programming paradigm for implementing data transformation. It also circumvents the problem of creating and maintaining two code bases (for online and offline processing) typical for the Lambda Architecture.

To achieve this, the Kappa architecture harnesses an append-only transaction/event log as canonical data store, able to maintain a massive and ordered set of messages (historical data) and also capable of supporting multiple concurrent clients/subscribers. This log streams the available data to a distributed stream computation engine (acting as subscriber) in charge of transforming, processing and generating incremental views as new data is ingested. Those views are then queried and visualized by users, featuring reasonable latency for interactive applications. Figure 3 presents an overview of the architecture of this approach.

B. Technology mapping for evaluation

The above architecture was implemented by combining various open-source technologies, as illustrated in Figure 4. The details regarding the construction of this system are discussed below.

As the Reddit submissions data set is stored in HDFS, a data ingestion stage is required in order to read the records, weed out fields holding irrelevant information or containing redundant data, and then feed the output to the append-only log storage. A data pipeline was set up in *StreamSets* [26] for

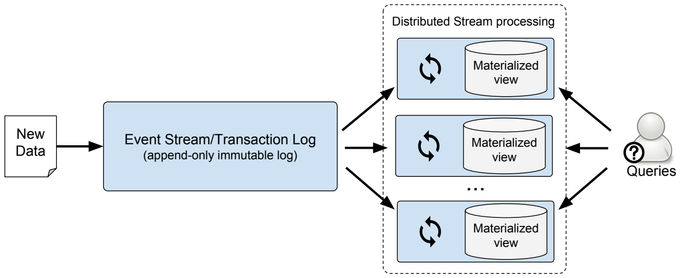
Distributed Stream processing
New Data
Event Stream/Transaction Log
(append-only immutable log)
Materialized view
Materialized view
...
Materialized view
Queries

Fig. 3. Overview of the stateful stream processing approach

this, harnessing the support this engine provides for multiple operations of data cleansing, filtering and transformation. The final stage in this pipeline consist in writing the processed submissions into an Apache Kafka topic, serving as the append-only immutable log from the architecture above.

The submissions published in Kafka are then fed into a distributed stream processing engine, in charge of running both the sentiment and term-frequency analysis jobs and incrementally generating two materialized views, keeping the resulting state of those jobs: one view storing the per-submission sentiment analysis results, and the other keeping track of the per-author term-frequency and overall term-frequency analysis. In this implementation the streaming library of Apache Spark [27] was used as stream processing engine and MongoDB [28] as store for the materialized views.

The implemented system is then able to provide partial results on the text analysis performed as new data gets ingested, while acknowledging the entire set of historical data stored in the append-only log.

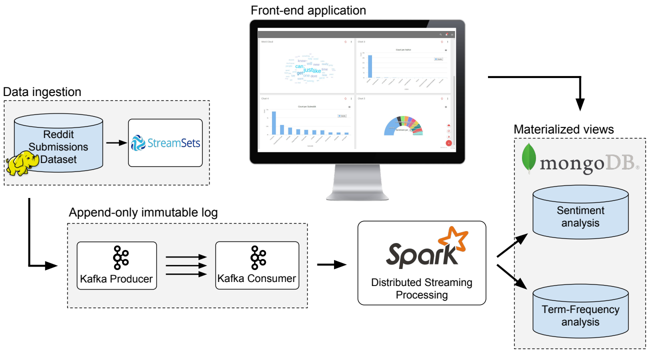

Fig. 4. Implementation of the stateful stream preprocessing approach

VI. Evaluation

One of the main motivations of this work was to build a system enabling low-latency querying and visualization of a sizable data set. As previously stated, a bulk of Reddit submissions was used, which in storage space add up to 269 GB of raw data contained in one single JSON file.

In this corpus, each Reddit submission is serialized as a JSON object with 39 fields, being `title`, `url`, `selftext`, `author`, `subreddit` and `score` among them. Fields like `title` and `selftext` hold text provided by the users authoring the submissions, while others like `num_comments` and `score` are indicators regarding the reception of the community of users on each particular submission.

After filtering out and removing the irrelevant JSON attributes the resulting submission objects comprise only nine fields, including the ones listed above.

Based on this data, two basic text analysis jobs were executed on the submission corpus, namely a simple per-submission sentiment analysis operation, and a corpus-wide term-frequency (TF) analysis. It is worth noting that, while the former analysis entails a stateless operation, the latter—since it is computed over the entire data set—requires for the system to keep track of the state of execution of the job and update the corresponding views accordingly.

As a final stage, a web client was built for querying and visualizing the Reddit corpus and the resulting materialized views. This web application provides a customizable dynamic dashboard implemented in Node.js and AngularJS that allows the user to create various charts displaying the result of parameterizable queries running against the Apache Drill API and the views stored in MongoDB. The charts also get dynamically updated as new results are available in the corresponding collections.

A. Experiment Set Up

The outlined implementation was deployed on the iLab.t Virtual Wall experimental testbeds [29] which comprise around 360 nodes. The Tengu experimentation platform [30] is running on the Virtual Wall testbeds, making it possible to automatically deploy several data processing, storage and cloud technologies, turning this process agile and straightforward. For the evaluation of the proof of concept implementation 10 third generation nodes were used: 2x Hexacore Intel E5645 (2.4GHz) CPU, 24GB RAM, 1x 250GB hard disk, 1-5 gigabit nics. The whole set up consist of 28 nodes as listed below:

- 11 Hadoop nodes (1 NameNode, 1 Resource manager, and 9 DataNodes)
- 9 Drillbits (Apache Drill execution units)
- 3 Kafka nodes
- 3 Zookeeper nodes
- 1 Spark node
- 1 MongoDB node
- 1 StreamSets instance (collocated with Hadoop's Resource manager)

To evaluate the two introduced approaches, the criteria mentioned earlier in the first section of this paper were considered: (*i*) support for ad-hoc querying, (*ii*) ability to deal with heterogeneous sources, and (*iii*) latency. The following subsections elaborate on how each approach addresses all those criteria.

B. Ad-Hoc Querying

When it comes to ad-hoc querying the approach encouraging the use of a distributed SQL query engine is the one providing the best support. The complexity of the queries a front-end application can issue against these engines is only limited by the expressiveness of SQL (and the subset of SQL actually supported by them). Moreover, such expressiveness might be further extended when working with a UDF-enabled engine, as in the case of Apache Drill and Apache Impala. However, this extensive expressive power for query building comes at the price of latency performance, as evidenced later on in this section.

On the other hand, a system implementing the stateful stream processing approach is able to provide immediate answer to arbitrarily complex queries depending on the information available in the incrementally generated materialized views. For the use case at hand, a streaming application was built on Apache Spark able to perform sentiment and term-frequency analysis on the Reddit submissions entering the system. The outcome of such application became available as views, enabling front-end applications to perform low-latency querying against the results of those analysis.

C. Heterogeneous sources

After implementing the two explored approaches, it is safe to say that this criterion has no significant impact on the architectural level and largely depends on the technologies used for data ingestion. Thereby, for instance, by using Apache Drill for implementing the SQL query engine approach, the system is enabled to support multiple data stores and several serialization formats.

The same applies for the stream processing approach where StreamSets was used as data ingestion tool, supporting a wide spectrum of formats (Avro, delimited, JSON, Google's protocol buffers, Text, XML) coming from multiple sources (event sources, message brokers, relational and non-relational storage systems) .

In this sense, while of major importance as requirement for real world big data applications, this criterion may not be deemed as a deal-breaker among the presented approaches.

D. Latency

The evaluation of this criterion involved measuring the response time of the implemented systems when dealing with non-trivial queries. Figure 5 reports on the latency measurements for the system implementing the distributed SQL query engine approach. By leveraging the Apache Drill SQL API, multiple queries of variable complexity were submitted, involving operations of aggregation, sorting, and the execution of the sentiment analysis UDF. Figure 5 shows the average response time for four different queries, which were issued against two versions of the data set: first over the full Reddit submission corpus, and then against a 100 megabytes slice comprising 100 000 user entries. When querying the entire data set, the system takes about 1.5 hours time to resolve queries consisting of standard SQL operators and functions, and more than 2 hours to process queries involving sentiment analysis. Not unexpectedly, latency values obtained for the 100MB slice were fairly low, but still not small enough to ensure interactive querying.

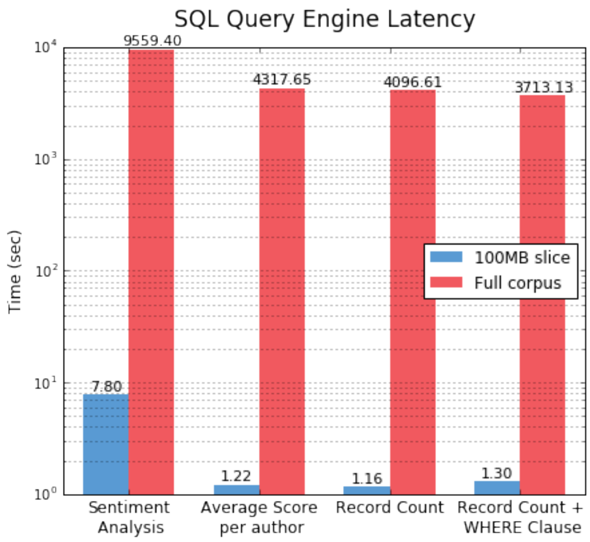

Fig. 5. Latency of the SQL query engine implementation

While optimized for enabling ad-hoc querying on top of large data sets, these SQL engines are still batch-oriented computation systems, unable to provide partial results on the incoming queries. In consequence, they are not suitable for near real-time or interactive visualization applications.

In contrast, the system implementing the alternative stream processing approach shows a response time lower by several orders of magnitude in comparison to the results obtained for the previous implementation. By using the *Apache HTTP server benchmarking tool* [31] it was possible to measure the time it takes for the system to process queries issued against the information views generated by the stream computation engine. It is worth noting that latency measurements are subject to the technology used for storing such views. In this case the system used MongoDB collections for storing pre-computed query results about sentiment and term-frequency analysis.

That being said, 2000 request were issued with a concurrency level set to 200, getting an average time per request of 33.561 milliseconds, while for visualizing the query results on the implemented web dashboard it takes 44.456 milliseconds on average. In consequence, systems implementing stateful stream processing are able to provide an interactive experience ($latency \leq 0.15s$), as long as the views resulting from the stream computation contain the information required for handling the queries issued by visualization and front-end applications. Table I details the time it takes for the implemented system to process individual records of the input stream.

VII. Conclusion

For data-intensive applications like cyber-bullying detection in social media streams, being able to query, process and visualize data as it is generated is a sensitive requirement to spot risky behavior, so that timely actions can be taken to avoid

TABLE I
AVERAGE PROCESSING AND RESPONSE TIMES

Per-record processing time (ms)			**Querying (ms)**	**Visualization (ms)**
StreamSets ingestion	Sentiment analysis	TF analisys		
0.224	3.84	18.07	33.561	44.456

or mitigate harmful consequences. Traditional frameworks for data processing fall short in meeting this requirement because they are designed to perform complex and time-consuming computations over large distributed data sets.

This paper introduces an ongoing work on low latency querying for visualization of big data sets leveraging existing technologies and software architectural patterns for enabling scalable storing and processing of continuous streams of data. Two approaches have been explored for tackling the stated problem: the first one, involves using a distributed SQL engine for supporting ad-hoc querying on top of a big data set; the second entails the usage of a stream processing engine for performing computations on the data as it becomes available, tackling the limitations of the previous approach by enabling the pre-processing of arbitrarily complex queries on the available data.

The evaluation conducted on a proof of concept implementation of the two mechanisms spots their benefits and flaws, and evidences how the approach proposing the use of a stateful stream processing engine is the most suitable alternative for enabling low latency querying on large and continuously growing data sets. One attractive feature of a system implementing this approach is the separation of concerns, in the sense that it features a clear split between data processing operations and query management, and also between data ingestion and information retrieval. This of course comes at a price: the data entering the system is not immediately accessible as is the case in traditional databases. A strategy for bridging this gap is to bump up parallelism when executing the stream processing job, so that more data gets processed in less time. Further research will involve the definition of a comprehensive analytical model for the interactive querying and near real-time data visualization problem.

ACKNOWLEDGMENT

This work was partly carried out with the support of the AMiCA (Automatic Monitoring for Cyberspace Applications) project, funded by IWT (Institute for the Promotion of Innovation through Science and Technology in Flanders) (120007).

REFERENCES

[1] Gilbert, S., & Lynch, N. A. (2012). Perspectives on the CAP Theorem. Institute of Electrical and Electronics Engineers.

[2] Van Hee, C., Lefever, E., Verhoeven, B., Mennes, J., Desmet, B., De Pauw, G., Daelemans, W., & Hoste, V. (2015). Automatic detection and prevention of cyberbullying. In International Conference on Human and Social Analytics (HUSO 2015) (pp. 13-18). IARIA.

[3] White, T. (2012, May 19). Hadoop: The definitive guide. ”O'Reilly Media, Inc.”

[4] Marz, N., & Warren, J. (2015, March 31). Big Data: Principles and best practices of scalable realtime data systems. Manning Publications Co.

[5] Kreps, J. (2014). Questioning the lambda architecture. https://www.oreilly.com/ideas/questioning-the-lambda-architecture. Retrieved: Jun, 2016.

[6] ”Elasticsearch - Search & Analyze Data in Real Time,” https://www.elastic.co/products/elasticsearch Retrieved: Aug 2016

[7] ”Kibana - Explore & Visualize Your Data” https://www.elastic.co/products/kibana Retrieved: Aug 2016

[8] Hausenblas, Michael, and Jacques Nadeau. ”Apache drill: interactive ad-hoc analysis at scale.” Big Data 1.2 (2013): 100-104.

[9] Kornacker, M., & Erickson, J. (2012). Cloudera Impala: Real Time Queries in Apache Hadoop, For Real. http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real.

[10] Melnik, S., Gubarev, A., Long, J. J., Romer, G., Shivakumar, S., Tolton, M., et al. (2010). Dremel: interactive analysis of web-scale datasets. Proceedings of the VLDB Endowment, 3(1-2), 330-339.

[11] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., et al. (2009). Hive: a warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment, 2(2), 1626-1629.

[12] AMPLab University of California, Berkeley (2014). AMPLab Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/

[13] ”Allegro Tech - Fast Data Hackathon,” http://allegro.tech/2015/06/fast-data-hackathon.html Retrieved: Aug, 2016

[14] ”Tableau, Business Intelligence and Analytics,” http://www.tableau.com/ Retrieved: Aug 2016

[15] ”Pentaho, Data Integration and Business Analytics Platform,” /url-http://www.pentaho.com/ Retrieved: Aug 2016

[16] ”Apache Samza,” http://samza.apache.org/ Retrieved: Aug, 2016

[17] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S. and Tzoumas, K., 2015. Apache Flink: Stream and Batch Processing in a Single Engine. IEEE Data Engineering Bulletin.

[18] Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S. and Saha, B., 2013, October. Apache hadoop yarn: Yet another resource negotiator. In Proceedings of the 4th annual Symposium on Cloud Computing (p. 5). ACM.

[19] ”Apache Kafka, A high-throughput, distributed messaging system,” http://kafka.apache.org Retrieved: Jul, 2016

[20] Tolia, N., Andersen, D.G. and Satyanarayanan, M., 2006. Quantifying interactive user experience on thin clients. Computer Science Department, p.77.

[21] Paul, M. J., Sarker, A., Brownstein, J. S., Nikfarjam, A., Scotch, M., Smith, K. L., & Gonzalez, G. (2016). Social Media Mining for Public Health Monitoring and Surveillance. In Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing (Vol. 21, p. 468). Chicago

[22] ”Full Reddit Submission Corpus,” https://www.reddit.com/r/datasets/comments/3mg812/full_reddit_submission_corpus_now_available_2006/. Retrieved: Jul, 2016

[23] ”Writing a custom SQL function for sentiment analysis,” http://www.dremio.com/blog/writing-a-custom-sql-function-for-sentiment-analysis/. Retrieved: Jul, 2016

[24] Hutto, C.J. & Gilbert, E.E. (2014). VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text. Eighth International Conference on Weblogs and Social Media (ICWSM-14). Ann Arbor, MI, June 2014.

[25] ”HDFS Architecture Guide,” https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. Retrieved: Jul, 2016

[26] ”StreamSets,” https://streamsets.com/. Retrieved: Jul, 2016

[27] ”Apache Spark Streaming,” http://spark.apache.org/streaming/ Retrieved: Jul, 2016

[28] ”MongoDB,” https://www.mongodb.com/ Retrieved: Jul, 2016

[29] ”iMinds iLab.t Virtual Wall,” http://doc.ilabt.iminds.be/ilabt-documentation/virtualwallfacility.html Retrieved: Jul, 2016

[30] T. Vanhove, G. Van Seghbroeck, T. Wauters, F. De Turck, B. Vermeulen and P. Demeester, ”Tengu: An Experimentation Platform for Big Data Applications,” 2015 IEEE 35th International Conference on Distributed Computing Systems Workshops, Columbus, OH, 2015, pp. 42-47. doi: 10.1109/ICDCSW.2015.19

[31] ”Apache HTTP Server Benchmarking tool, ” http://httpd.apache.org/docs/current/programs/ab.html Retrieved: Ago, 2016