HHS Public Access

Author manuscript

Proc IEEE Int Congr Big Data. Author manuscript; available in PMC 2018 March 28.

Published in final edited form as:

Proc IEEE Int Congr Big Data. 2017 June ; 2017: 376–383. doi:10.1109/BigDataCongress.2017.55.

Experiences with the Twitter Health Surveillance (THS) System

Manuel Rodríguez-Martínez

Department of Computer Science and Engineering, University of Puerto Rico, Mayagüez

Abstract

Social media has become an important platform to gauge public opinion on topics related to our daily lives. In practice, processing these posts requires big data analytics tools since the volume of data and the speed of production overwhelm single-server solutions. Building an application to capture and analyze posts from social media can be a challenge simply because it requires combining a set of complex software tools that often times are tricky to configure, tune, and maintain. In many instances, the application ends up being an assorted collection of Java/Scala programs or Python scripts that developers cobble together to generate the data products they need. In this paper, we present the Twitter Health Surveillance (THS) application framework. THS is designed as a platform to allow end-users to monitor a stream of tweets, and process the stream with a combination of built-in functionality and their own user-defined functions. We discuss the architecture of THS, and describe its implementation atop the Apache Hadoop Ecosystem. We also present several lessons learned while developing our current prototype.

Keywords

big data analytics; streaming; use-defined functions; Twitter; social media

I. Introduction

Social media has become an important platform to gauge public opinion on topics related to our daily lives, for example, the results of the 2016 U.S. Presidential Election. Posts to social platforms (e.g., Twitter, Facebook) can be collected and mined in search for information about issues that concern a given population. These concerns can include diseases, political issues, or economic considerations, to name a few. For example, sentiment analysis [1], [2], [3], [4], [5] has been used to track Twitter posts related with the Flu and other diseases. These same techniques can be used to study potential psychological effects (e.g., anger, depression) that political or economical events can exert on a population. Such information is vital for public officials and health care providers that seek mitigation of the problems that erupt from such events.

In practice, processing these posts requires big data analytics tools since the volume of data and the speed of production overwhelm single-server solutions. Machine learning and statistical methods are used to analyze the data, typically using classification techniques to assign posts to specific topics or determine trends [5]. Such analysis can be done in *batch mode*, where analytics jobs run over a collection of data previously collected and currently sitting on stable storage. For example, we might run a classification process (training,

validation, and actual usage) on data sitting on a parallel file system (e.g., Apache HDFS). In contrast, processing in *online mode* refers to analysis done on a live data stream, in which data become available for analysis as soon as acquired through a data streaming system. In this case, we might run a classifier for tweets as soon they become available from the Twitter Stream API. This later approach can be used to show live trending, or to raise notifications when some event of interest is detected as it starts to develop.

Building an application to capture and analyze posts from social media can be a challenge simply because it requires combining a set of complex software tools that often times are tricky to configure, tune, and maintain. For instance, setting up a processing pipeline that captures tweets into Apache Kafka, then feeds those tweets from Kafka into Apache Spark, and finally stores the results in HDFS requires understanding the intricacies of three very different platforms. Coding an application against these software stacks is also challenging due to the number of dependencies that must be linked into the application. In many instances, the application ends up being an assorted collection of Java/Scala programs or Python scripts that developers cobble together to generate the data products they need. This makes it difficult for small business, or enterprises not focused on IT (e.g., manufacturing, health care) to acquire and operate big data technology necessary to revamp their operations.

Our team at the University of Puerto Rico, Mayagüez is currently developing the Twitter Health Surveillance (THS) application framework. THS is designed as a platform to allow end-users to monitor a stream of tweets, and process the stream with a combination of built-in functionality and their own user-defined functions. THS enables online processing capabilities of the raw tweet stream, which is a departure from methods in [1], [2], [3], [4], [5], which mostly use curated collections of tweets for their analysis. We currently focus on Twitter because it is a popular social platform that provides access to posts without the need for establishing “friendship” relationships between its users. Nonetheless, the ideas associated with THS can be applied to any other platform comprising of text messages that are streamed to a large audience, and which can be analyzed in search for keywords, hashtags, topics, and so on. In the future, we shall expand our efforts into other social platforms. In this paper, we present the architecture of THS, and describe its implementation atop the Apache Hadoop Ecosystem. We discuss our implementation rationale and several lessons learned while developing our current THS prototype.

A. Contributions

In this paper, we describe THS and make the following contributions:

- Describe how the problem of sentiment analysis can be formulated as a big data online streaming problem.
- Present THS as a reference architecture for applications that need to process stream data with non-trivial methods.
- Describe a reference implementation of THS, using Java, Python, and big data software tools from Apache Hadoop/
- Discuss lessons learned during the implementation process that could help other developers facing similar situations.

B. Paper Organization

The rest of this paper is organized as follows. Section II contains the motivation for THS and an overview of the problem we want to solve. In section III, we go into the details of the THS architecture. The current implementation status is presented in section IV. Section V presents a list of lessons learned so far. Related works are presented in section VI. Finally, a summary of the paper is presented in section VII.

II. System Overview

In this section we provide an overview of the THS platform, and describe its major components.

A. Motivation

Sentiment analysis [6], [7], [8], [9], [10], [11] is a popular technique to analyze social media, blogs, and news articles in search for keywords that denote positive or negative views against a particular product, person, or situation. In the context of THS, one of our goals is to collect a stream of messages ("tweets") $m_1, m_2, \ldots, m_n$ that make reference to a particular subject T. Next, we have a dictionary consisting of k pairs $w_1, w_2, \ldots, w_k$, where each w_i is a pair of the form (t_i, s_i). The component t_i represents some word or phrase, and the component s_i is a number that provides a sentiment score for t_i. If the word t_i is not found on a message, then we score a value of 0 for it. Each message m_j is then matched one by one against each w_i in the dictionary, and the sentiment values $s_1, s_2, \ldots, s_i, \ldots, s_k$ are used to compute the overall message sentiment $\sigma(m_j)$. One approach to compute this sentiment is for $\sigma(m_j)$ to add the individual sentiment values for each word w_i found in the message m_j. Other methods [12] first segregate the sentiment values into two lists: those from positive words and those for negative words, Then, a ratio r is computed by dividing the sum of the negative sentiments over the sum of the positive sentiments. Ratio r becomes the sentiment $\sigma(m_j)$ of the message m_j.

B. Processing Pipeline

Conceptually, the tweets are processed using a pipeline as shown in Figure 1. As tweets are generated, they are captured and then filtered based on keyword so they can be assigned to a given topic. Next, tweets are routed for processing at additional stages. One stage performs sentiment analysis as described in the preceding section. More stages can be added to perform custom-processing such as additional keyword filtering, emoji analysis, classification, or computing target keyword frequency. The output from all these stages can be configured to go into dash boards, databases, HDFS, etc. In our case, raw tweets are always stored into HDFS to enable further analysis in the future. Notice that it is possible to re-ingest some of the output back into the sentiment analysis or custom processing stages, perhaps to fine tune the results as models get re-calibrated.

III. THS Architecture

We are developing THS as a collection of daemons and web services that work together to collect, index, analyze, support queries on the tweets, and help make predictions. Figure 2

depicts the components of THS: a) Data Acquisition Component (DAC), b) Sentiment Analyzer (SA), c) Predictive Analytics Component (PAC), and d) Data Access Provider (DAP).

A. Data Acquisition Component

We use the Twitter's Stream API to continuously monitor a sample of the stream of messages coming from Twitter. One or more Data Acquisition Components (DAC) capture these messages from the social network, as shown in Figure 3. Each DAC is a lightweight server-side application, written in Java, which continuously monitors the social network, scanning and matching messages based on user-defined criteria.

In our current experimental set up, messages are matched against a set of health-related keywords (e.g., fever, joint pain) to ensure that tweets that might be health related get captured even if no additional stage has been explicitly configured to capture them. These filtering parameters are fed into the DAC by means of configuration files encoded in JSON. For example, suppose a user wants to monitor all messages with keyword `influenza` posted within the State of Florida by any user. Then, the JSON configuration file will look like this:

```
{
    "Hashtag" : "influenza",
    "Location" : "FL",
    "Sender" : *
}
```

In this example, the character * is used as a wildcard indicating that posted messages from any users must be captured. Alternatively, one or more Twitter screen names (separated by commas) can be specified in this field.

In the near future, these files will be generated from a web app, designed with a friendly User-Interface (UI). This would make the system easy to use by one of our target audiences: public health officials, epidemiologists are other professionals whose job is to monitor diseases and health conditions.

In a sense, our DAC solves a similar issue that SQL solved for early database developers, namely, removing the need to write a program for each query that was posed to the database. Instead, SQL enables users to specify the data they require as a string that specifies target data collections (tables), attributes, and filtering criteria. Similarly, the DAC enables users to specify what keywords must be used to capture tweets, with no need to setup scripts and programs to do so. Thus, data acquisition becomes a ready-to-use service provided by THS.

The DACs store the captured tweets into Hive, which is a cluster-based data warehouse, built atop Hadoop MapReduce and the HDFS file system. Each tweet is stored as read, but also additional attributes (disease, date, geo-location, hashtag) are added into the Hive tables to help index the tweets and simplify/accelerate searches. Summarized data, for example

number of tweets per topic, week, users, are kept in a PostgreSQL database that can be queried to see trending over time. In section IV, we present more implementation details of the current DAC prototype.

B. Sentiment Analyzer (SA)

In THS, one or more Sentiment Analyzer (SA) awaken periodically to sift over a collection of tweets to be analyzed. Some of these SAs work directly on the tweet stream, while others act upon tweets stored in Hive. The former types are intended for near real-time trending information (online work), while the later types are geared toward more complex and time-consuming work. In our current prototype, the SA works as a daemon application written in Python, which can be customized by developers to create specific functionality into the SA. Custom analysis functions are implemented and bundled into the application, as well as the specific tweet topic (e.g., keyword(s)) to be analyzed. By using Python, we can leverage on the wealth of existing Python code for sentiment analysis. The organization of the SA is depicted in Figure 4. Although not shown in the picture, the SA relies on Apache Spark and Spark Streaming to process the tweets in a cluster.

Each SA is configured with a dictionary containing the words and phrases it must search for, and the sentiment scores for each one of them. Currently, our dictionary is mostly health related. The dictionary is loaded from a JSON file that the user supplies, and can be modified as needed to adapt the dictionary. In our project, we assume these words and phrases are in English or Spanish. However, the SA can be expanded to include dictionaries in any of the other languages supported by Twitter. The SA performs sentiment analysis on each tweet, using either built-in or user-supplied sentiment functions that implement the desired algorithms. By default, we a variation of the formulation in [12], defining the sentiment of a message m, as

$$\sigma(m)=\frac{\text{count(positive words)}}{\text{count(negative words)}}$$

For trending purposes, a topic x (e.g., influenza) can be linked to a sequence of tweets m_1, m_2, …, m_k. Thus, at time t the sentiment on topic x, denoted as x_t can be computed by aggregating and averaging the sentiment of the associated tweets. A simple method is to use the average:

$$x_t=1/k\sum_{i=1}^{k}\sigma(m_i)$$

Other alternatives include using weighted averages to give more importance to more recent messages. Similarly, we can find averages for the sentiment on a topic over a time period [t_1, t_2].

The SA stores the results of the analysis for each tweet back into Hive and also stores a record to remember each tweet it has already seen. These results can be used for several purposes, besides trending. In one instance, the results can be used to train a classifier that

detects if a new, unseen tweet is positive or negative. Such tweets can then be send to analysts for further reviews. For example, health care users might wish to determine if the messages being posted denote complaints about symptoms, relief that a diseases is gone, or concerns about some possible outbreak. In another instance, the result go into a reporting system that generate daily statistics on tweet activity, with special focus on a given region and topic combination.

C. Predictive Analytics Component

The Predictive Analytics Component (PAC) provides the “hooks” to access basic analytics functionality to help developers build customized applications for making predictions. This will enable value-added apps for end-users to access the system, run predictions, and visualize the data at hand. The PAC is being built as a REST API using Python Flask. Underneath, the REST API connects to the Machine Learning Library (MLLib) of the Spark system, to capitalize on the performance obtained by running queries and machine learning operations on memory-resident data.

Figure 5 presents the organization of the PAC. At the top of the figure, we have the REST API that enables users to communicate with the PAC. The Data Loader component provides a series of classes to load tweets from the Hive Warehouse (actually from the HDFS) into Spark. The Operations Coordinator provides the mechanism to pose commands to Spark and run analytics operators on the target tweets. These two components are implemented as Java daemons. The main idea with all these components at the PAC is to provide a higher layer of abstraction at the analytics layer that helps integrate the different pieces of the system while reducing the amount of boilerplate code that developers need to write in their applications. At the moment of this writing the PAC is rather rudimentary, and we expect to enhance its functionality within the next six months.

D. Data Access Provider (DAP)

The Data Access Provider (DAP) serves as the entry point for visualization applications built atop THS. We are building the DAP as a REST-based service powered the Python-based Django web framework. Figure 6 shows the organization of the DAP. The DAP is hosted inside the Django web container, which in turn is hosted inside the Apache2 web server. Requests for data are received as REST operations. These request are handled by one of our custom-built Django view controllers, which are Python objects. Each request is examined to determine if an HTML response or a JSON response is expected. HTML responses are returned to browser applications, while JSON responses are given to Javascript web apps and native applications running on the mobile devices. Based on this determination, the view controller object instantiates a request handler object to manage the data extraction process. The request handler follows the functionality of the front controller and the façade patterns.

The request handler instantiates objects in the data model in order to: a) query the underlying database (e.g., Hive, PostgreSQL) to fetch records necessary to answer the request from the user, b) assemble model objects from database records, c) perform any necessary business logic, and d) build collections with the results of the operation. The request handlers use the services of the data access objects (DAOs) to access the underlying

database. In turn, the DAOs use a simple representation of the database records that has no business logic and is provided by the classes at the Data Transfer Objects (DTO) layer. Notice that both DAOs and DTOs are design patterns. Finally, the DB API provides the DAO with the necessary code to read data from the database. This layer includes not only code to access either Hive or Spark, but also code to make appropriate invocation of predictive analytics routines.

IV. Implementation status

At the moment of this writing, we an have alpha-state implementation of the system. All Java modules have been developed using Java 1.8, and all Python modules have been developed using Python 2.7.10. Our development team has used maven, pip, and virtualenv for dependency management. The system is run on a cluster of about twenty machines with various hardware specifications. All machines have 8 GB of RAM, and 1 TB of disk space. However, around ten machines have AMD Phenom 2 Quad Core CPUs, while the rest have Intel Xeon Quad Core CPUs. The machines have 1Gbps Nic, but are connected to the new UPRM SciNet Science DMZ Network, which features 10Gbps bandwidth. All machines in this cluster run Ubuntu Linux 14.04 LTS as their OS.

Most of our current efforts have focused on the DAC since we need that component to collect the tweets that are processed by the rest of the system. Figure 7 depicts the current implementation of the DAC. As mentioned before, the tweets are consumed by a Java daemon, and are then sent to a fleet of Apache Kafka servers. Kafka implements a distributed streaming platform, using a publish-subscribe model. Our main rationale for using Kafka was to ensure that we could keep up with the Twitter stream. The documentation for the Twitter Stream API explicitly indicates that applications that cannot keep up receiving tweets are automatically disconnected. In our early implementation of the DAC, we frequently had that problem because we were trying to capture the tweets and route them directly for the next processing state. Using Kafka eliminated that problem by simplifying the process. Right now, out capture daemon uses the Twitter4J API to collect the tweets and channels them straight to the Kafka system. This communication is done asynchronously by Kafka's client API, so the daemon can get back to retrieve the next tweet.

Kafka implements a log-base storage abstraction that enables the next stage in our pipeline to consume tweets without concern about getting disconnected from the stream. Moreover, Kafka keeps track of the last record consumed by a client, and this has become handy for us in order to recover from failures while testing the system. In our case, the next stage after ingestion into Kafka is the filtering module, which inspect the text of the tweet and other metadata in the JSON representation of the tweet. This stage is implemented by a Java application that invokes the Spark Streaming engine on batches of tweets. The filters include keyword, sender, location (if available), date, and language, to name a few.

The result of the filtering phase are then stored as follows: a) the raw tweet is stored in HDFS, and b) metadata derived from the tweet are stored in PostgreSQL. For example, for each tweet we store in PostgreSQL a record containing the id of the tweet, the screen name

of the sender, and the date it was created. In addition, we add a record in PostgreSQL for each topic to which the tweet can be related.

The remaining components of the system have been implemented as described in section III, and we shall report on their status in the near future.

V. Lessons Learned

During the course of our implementation effort we have learned several lessons that we would like to share now with the reader in hope of helping those with similar development efforts.

A. The Learning Curve is Steep

Building a system like THS requires learning about a set of very different software tools, each one having its complexity and idiosyncrasy. Do not expect that by just reading the documentation, you will be able to get things running quickly. Rather, you will need working with the tools extensively to get a grasp of the tricks and pitfalls to get them to operate they way you want. Working with the tools in standalone and pseudo distributed mode is easy. Once you get into cluster mode, you end up trying various settings (e.g., directory for logs, network ports) until you get the system to run consistently without problems. Despite the abundance of tutorials and books about Hadoop, HDFS, and Spark, we often had to spend hours and days to debug our configurations and applications to make them work.

Another factor that complicates matters is that different tools feel at home with different programming languages. While HDFS and Hadoop MapReduce are Java based, Spark feels more at home with Scala. Our team of students had ample experience with Java and Python, but not so much with Scala. So we did not attempted much coding in Scala as we are trying to get an operational system as quickly as possible.

B. Networking is Tricky

Often times, our problems were not bugs or misconfigurations but rather issues with the network. In one instance, we had trouble running HDFS on a fleet of VMs. It turned out that the VMs did not had the reverse DNS mapping properly configured in the DHCP server, thus preventing the HDFS datanotes from properly registering with the HDFS namenode. In addition, our network firewalls were blocking the VMs as they did not like their MAC address configuration. After several attempts, we scratched VMs and ran HDFS straight on the host OS. For similar reasons, we gave up on using containers (e.g., Docker). In the end, we felt that we would need heavy interaction with the IT staff to get these things sorted out, but our staff did not have the man power to accommodate our needs.

C. Start Small

Given the previous lessons, it should be clear that you have to start small. It is tempting to try to run things on a cluster of 100 nodes, but you will be asking for trouble. We got most of our problems worked out on a cluster configuration with four nodes: a) master node for

running the namenode, b) two slaves nodes to run the datanodes, and c) a node to run the client. Once we got that running, we ran Spark on another set of three nodes. Then, we ran Kafka in one node before moving to a three node configuration. After these configurations worked as expected, we moved to deploy the system on our full cluster.

D. Keep it Simple

There are many systems within the Apache Hadoop Ecosystem and its other related Apache projects. At first we tried to use tools like Apache Storm, MapReduce, HDFS, Spark, Zookeeper and Ozzie. We soon realized it was simply too much. We settled on using Spark for data processing because it has batch processing, streaming, SQL-like capabilities, Graph processing capabilities, and MLLib. We also decided to use Kafka for helping with data ingestion. This decision helped us to move from a “fighting the tools” mode into a more productive “finding my bugs” mode.

E. Maven is a Must

Running your Java apps with Spark or Hadoop requires assembling a jar file with your own code, plus all the dependencies (or a way to specify where to find those dependencies). Here dependencies refer to third-party libraries (e.g., Twitter 4J) that your application needs. In our initial implementation efforts we attempted to build the jar files either from the IDE or with tools other than Maven. We faced all sorts of problems putting the dependencies into the jar, and getting right the configuration for the Log4j logger (used extensively by the Apache tools). Moving to Maven solved all of our building and packaging problems right away. In our view, it makes little sense to look elsewhere because these tools play well with Maven.

F. Log4j is a Must

The distributed nature of the applications makes debugging difficult. Most (if not all) of the Apache tools use Log4j for logging purposes. Since most of these Apache tools run as daemons, you need to send your debugging output to log files. And they should not be the same log files as those used by tools because they really throw a lot of messages into their logs. You will have a hard time finding your logging messages there. You need to configure your own logger and trace as much as possible to make sure the system is behaving as you expect. We had several instances where our application was reading the wrong configuration file, and the only way to find that out was to use Log4j to trace the actual configuration parameters that the application was reading.

VI. Related Works

The term “big data” refers to data collections whose gargantuan volume and speed of generation, typically at Petabyte scale, makes them impractical for storage and processing using traditional, single-computer relational database systems (e.g., MySQL). Instead, computer clusters, parallel processing techniques, Map-Reduce [13], and statistical inference are needed to analyze the data at this level of scale. Big Data collections naturally arise in experiments associated with social media, gene sequencing, high-energy physics, and health care monitoring, just to name a few.

Apache Hadoop has become the principal software platform for big data processing. An ecosystem of tools has evolved around Hadoop and its HDFS file system [14] including Hive [15] (SQL-style data warehouse for Hadoop), HadoopDB [16](integration of PostgreSQL and Hadoop), and SpatialHadoop [17](Spatial Indexing for Hadoop). Recently, Spark [18], [19] has emerged as a popular alternative for big data processing. Spark uses HDFS, but it is not based on MapReduce. Instead, is uses a Direct Acyclic Graph (DAG) model to represent the flow of computational tasks, and keeps all data items in main memory. This enables Spark to achieve better response time for data processing operations, particularly analytics and machine learning jobs, versus MapReduce variants that rely on disk-resident data.

Sentiment analysis [6], [7], [8], [9], [10], [11] is a popular technique to analyze social media, blogs, and news articles in search for keywords that denote positive or negative views against a particular product, person, or situation. Clues about health conditions affecting citizens of a region can be obtained from the public messages that these citizens post in social media apps such as Twitter. In fact, recent research work has focused on using Twitter as a tool to help uncover health trends [1], [2], [3], [4], [5] by looking at keywords in the messages that mention specific health conditions. THS leverages on many of these techniques, while proving a turnkey solution to build applications without much boiler plate code, and accessing the live Twitter stream instead of using a curated collection. To the best of our knowledge no other system offers the capabilities that THS has.

VII. Summary

In this paper, we have presented the Twitter Health Surveillance (THS) application framework. THS is designed as a platform to allow end-users to monitor a stream of tweets, and process the stream with a combination of built-in functionality and their own user-defined functions. THS enables online processing capabilities of the raw tweet stream, which is a departure from methods that mostly use curated collections of tweets for their analysis. We have presented the architecture of THS and its principal components: a) Data Acquisition Component (DAC), b) Sentiment Analyzer (SA), c) Predictive Analytics Component (PAC), and d) Data Access Provider (DAP). We have also discussed the current implementation status of the system atop the Apache Hadoop Ecosystem, and the lessons learned while performing that implementation.

Supplementary Material

Refer to Web version on PubMed Central for supplementary material.

Acknowledgments

THS is being implemented by a set of graduate and undergraduate students at the University of Puerto Rico, Mayagüez: Arun Sharma, Cristian Garzón, Luis G. Rivera, Andrés Hernández, Jady Urbina, Diego Figueroa, and César Cruz.

Research reported in this publication was supported by the National Library of Medicine, of the National Institutes of Health under award number R15LM012275. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

References

1. Parker, J., Wei, Y., Yates, A., Frieder, O., Goharian, N. A framework for detecting public health trends with twitter. Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining; ASONAM ’13; New York, NY, USA. ACM; 2013. p. 556-563.
2. Aramaki, E., Maskawa, S., Morita, M. Twitter catches the flu: Detecting influenza epidemics using twitter. Proceedings of the Conference on Empirical Methods in Natural Language Processing; EMNLP ’11; Stroudsburg, PA, USA. Association for Computational Linguistics; 2011. p. 1568-1576.
3. Culotta, A. Towards detecting influenza epidemics by analyzing twitter messages. Proceedings of the First Workshop on Social Media Analytics; SOMA ’10; New York, NY, USA. ACM; 2010. p. 115-122.
4. Diaz-Aviles, E., Stewart, A., Velasco, E., Denecke, K., Nejdl, W. Towards personalized learning to rank for epidemic intelligence based on social media streams. Proceedings of the 21st International Conference Companion on World Wide Web; WWW ’12 Companion; New York, NY, USA. ACM; 2012. p. 495-496.
5. Paul MJ, Dredze M. Discovering health topics in social media using topic models. PLoS ONE. 2014; 9:e10340808.
6. Choi, Y., Kim, Y., Myaeng, S-H. Domain-specific sentiment analysis using contextual feature generation. Proceedings of the 1st International CIKM Workshop on Topic-sentiment Analysis for Mass Opinion; TSA ’09; New York, NY, USA. ACM; 2009. p. 37-44.
7. Kim, S-M., Hovy, E. Determining the sentiment of opinions. Proceedings of the 20th International Conference on Computational Linguistics; COLING ’04; Stroudsburg, PA, USA. Association for Computational Linguistics; 2004.
8. Wang, H., Can, D., Kazemzadeh, A., Bar, F., Narayanan, S. A system for real-time twitter sentiment analysis of 2012 u.s. presidential election cycle. Proceedings of the ACL 2012 System Demonstrations; ACL ’12; Stroudsburg, PA, USA. Association for Computational Linguistics; 2012. p. 115-120.
9. Fan W, Gordon MD. The power of social media analytics. Commun. ACM. Jun.2014 57:74–81.
10. Arias M, Arratia A, Xuriguera R. Forecasting with twitter data. ACM Trans. Intell. Syst. Technol. Jan.2014 5:8:1–8:24.
11. Tsytsarau, M., Amer-Yahia, S., Palpanas, T. Efficient sentiment correlation for large-scale demographics. Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data; SIGMOD ’13; New York, NY, USA. ACM; 2013. p. 253-264.
12. O’Connor, B., Balasubramanyan, R., Routledge, BR., Smith, NA. From Tweets to Polls: Linking Text Sentiment to Public Opinion Time Series; Proceedings of the International AAAI Conference on Weblogs and Social Media; 2010.
13. Dean, J., Ghemawat, S. Proc. of 2004 OSDI. San Francisco, CA, USA: 2004. Mapreduce: Simplified data processing on large clusters; p. 137-150.
14. Chang, F., Dean, J., Ghemawat, S., Hsieh, WC., Wallach, DA., Burrows, M., Chandra, T., Fikes, A., Gruber, RE. Bigtable: A distributed storage system for structured data. Proceedings of the 7th Symposium on Operating Systems Design and Implementation; OSDI ’06; Berkeley, CA, USA. USENIX Association; 2006. p. 205-218.
15. Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R. Hive: A warehousing solution over a map-reduce framework. Proc. VLDB Endow. Aug.2009 2:1626–1629.
16. Abouzeid A, Bajda-Pawlikowski K, Abadi DJ, Silberschatz A, Rasin A. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. VLDB 2009. 2009
17. Alarabi, L., Eldawy, A., Alghamdi, R., Mokbel, MF. Tareeg: A mapreduce-based web service for extracting spatial data from openstreetmap. Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data; SIGMOD ’14; New York, NY, USA. ACM; 2014. p. 897-900.

18. 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. Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation; NSDI'12; Berkeley, CA, USA. USENIX Association; 2012. p. 2-2.

19. Xin, RS., Rosen, J., Zaharia, M., Franklin, MJ., Shenker, S., Stoica, I. Shark: Sql and rich analytics at scale. Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data; SIGMOD '13; New York, NY, USA. ACM; 2013. p. 13-24.

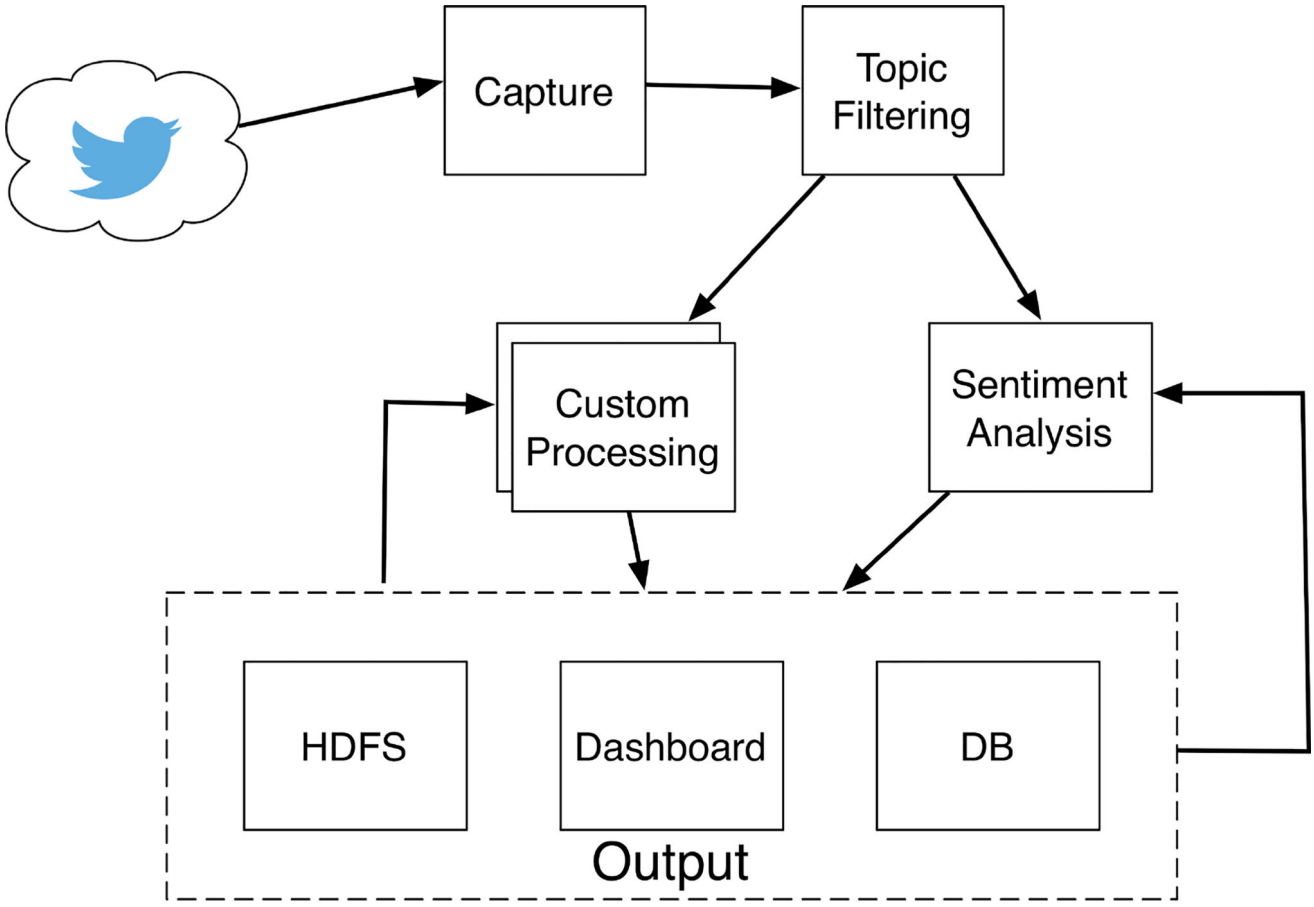

Figure 1.
Tweet Processing Pipeline

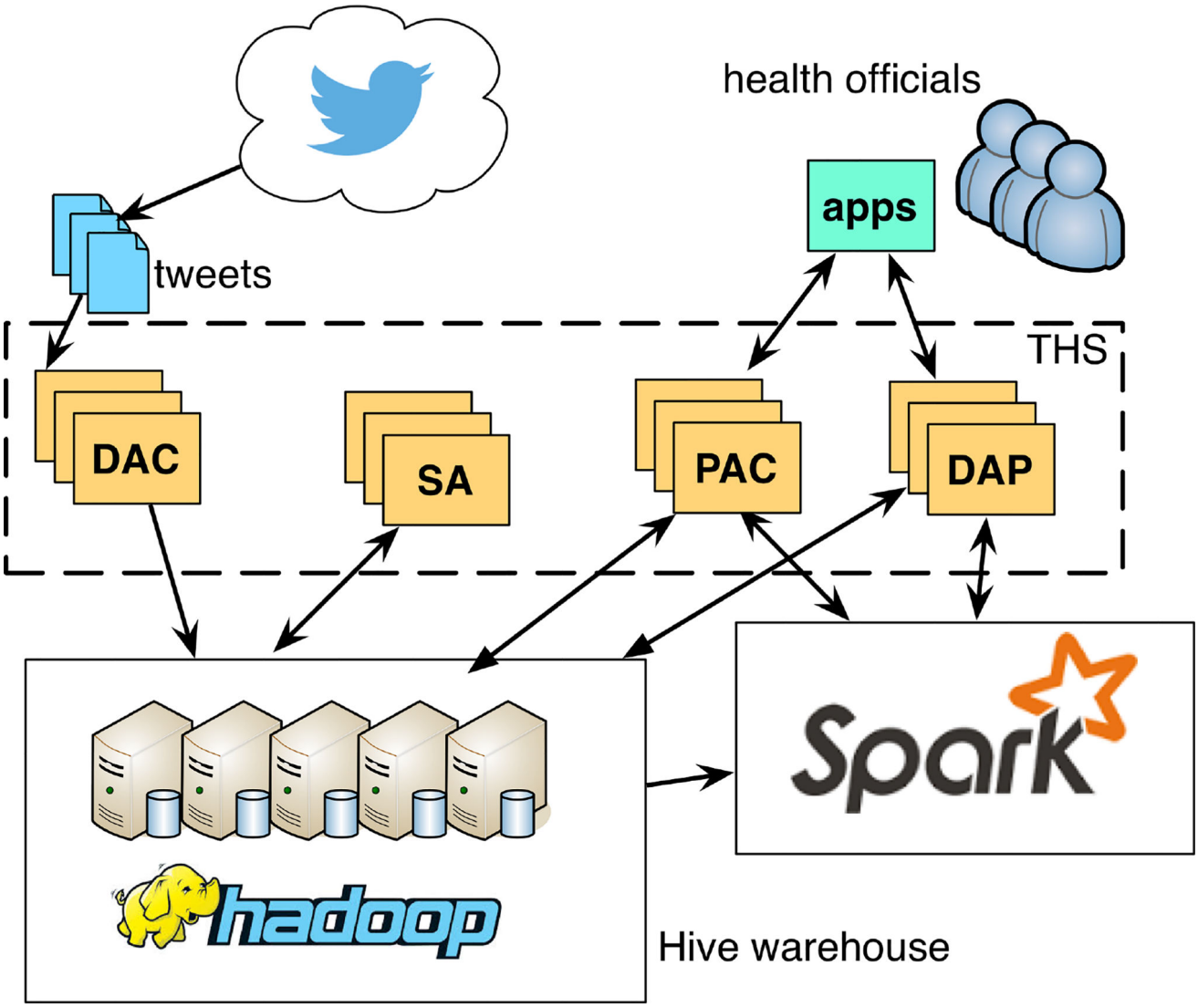

Figure 2.
General System Architecture

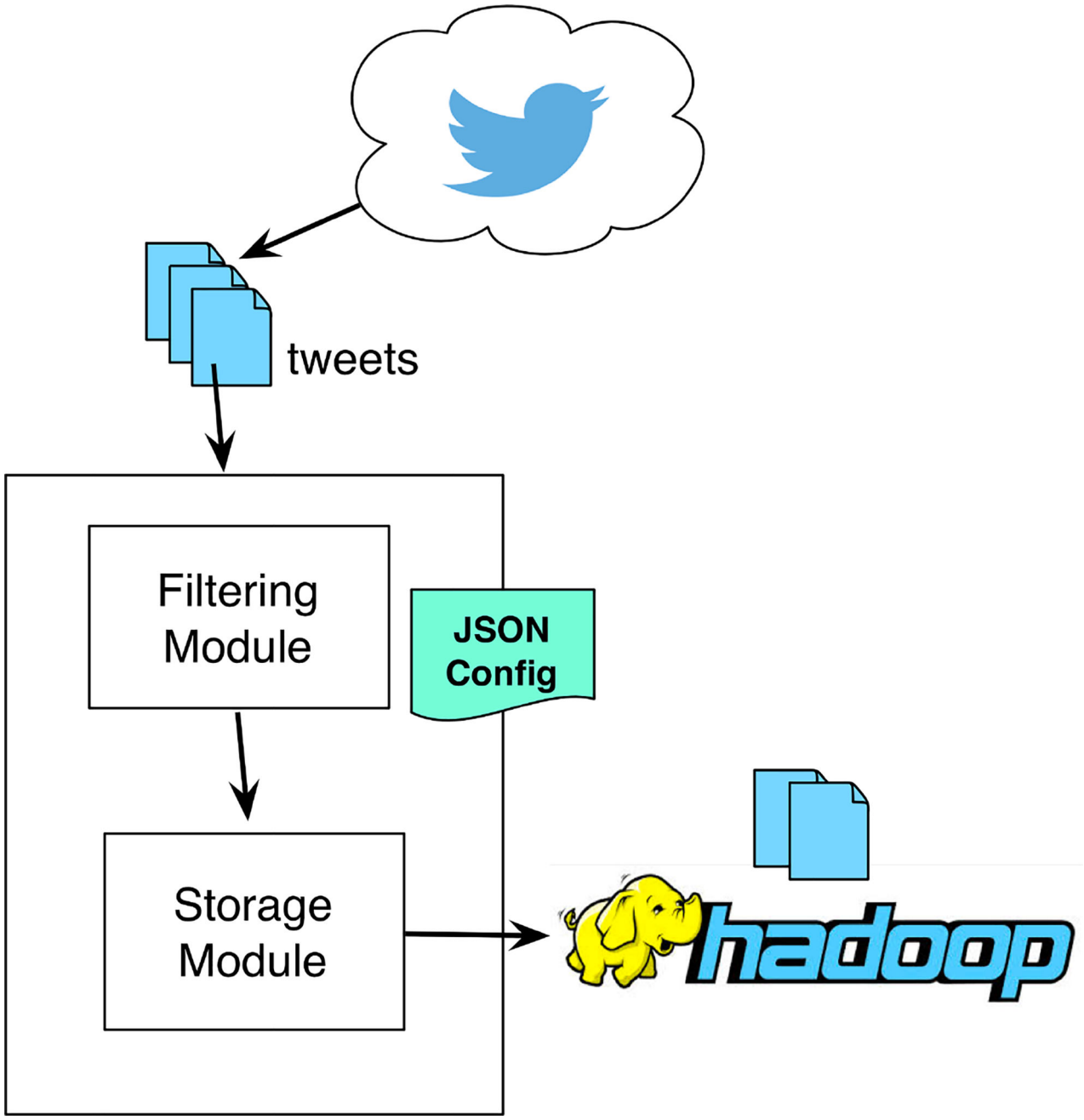

Figure 3.
DAC Organization

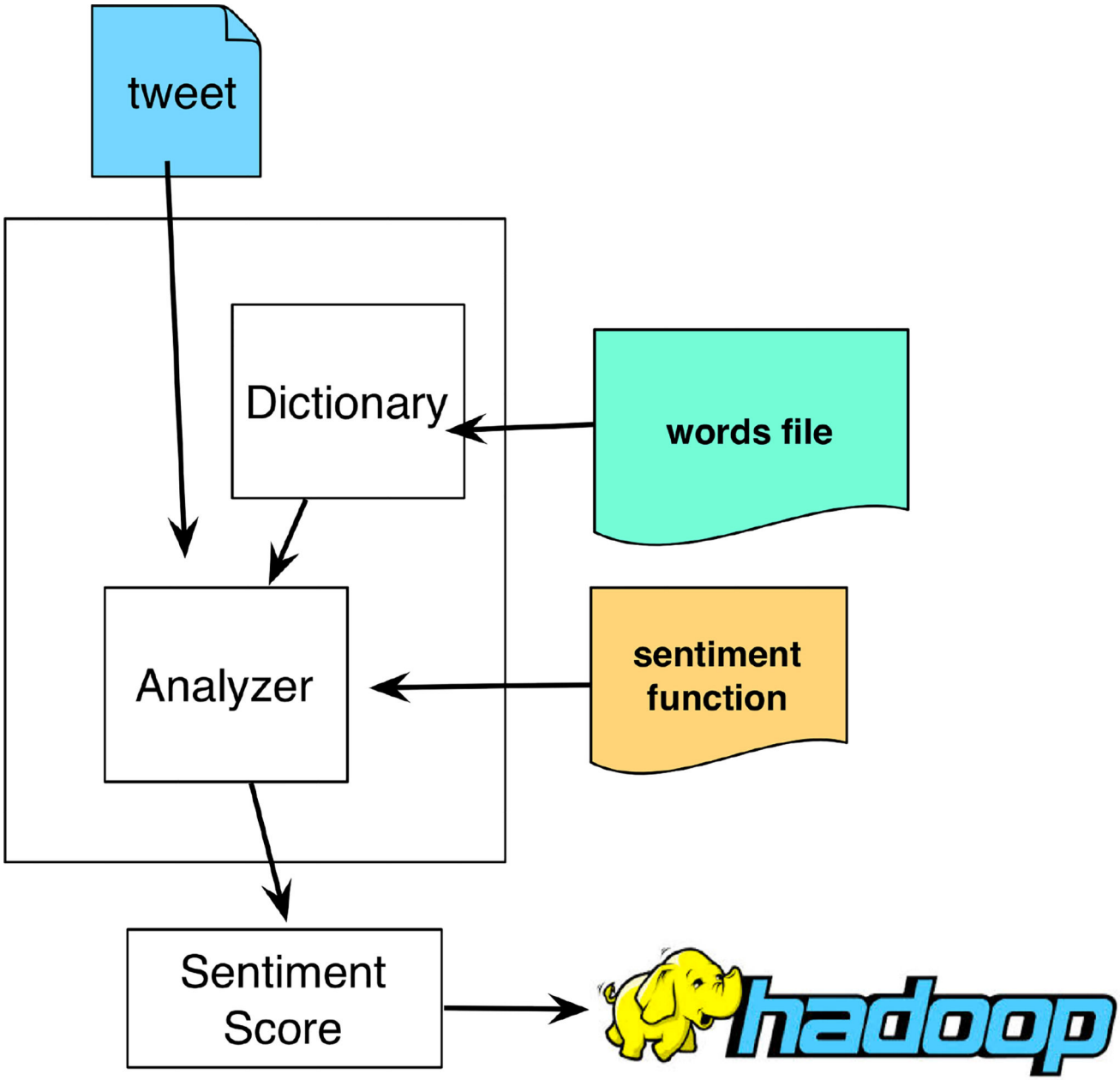

Figure 4.
Organization of the Sentiment Analyzer

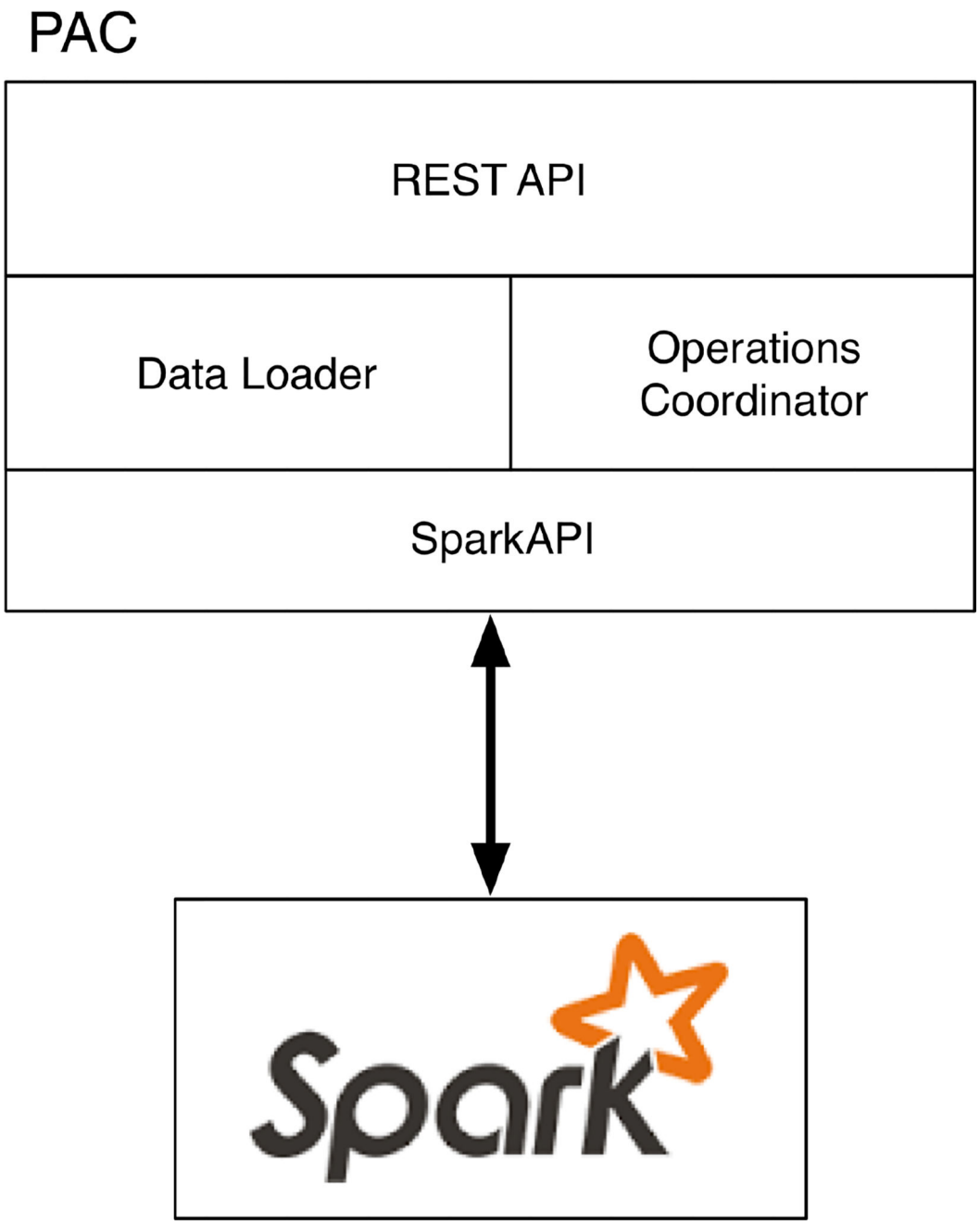

Figure 5.
Organization of the PAC

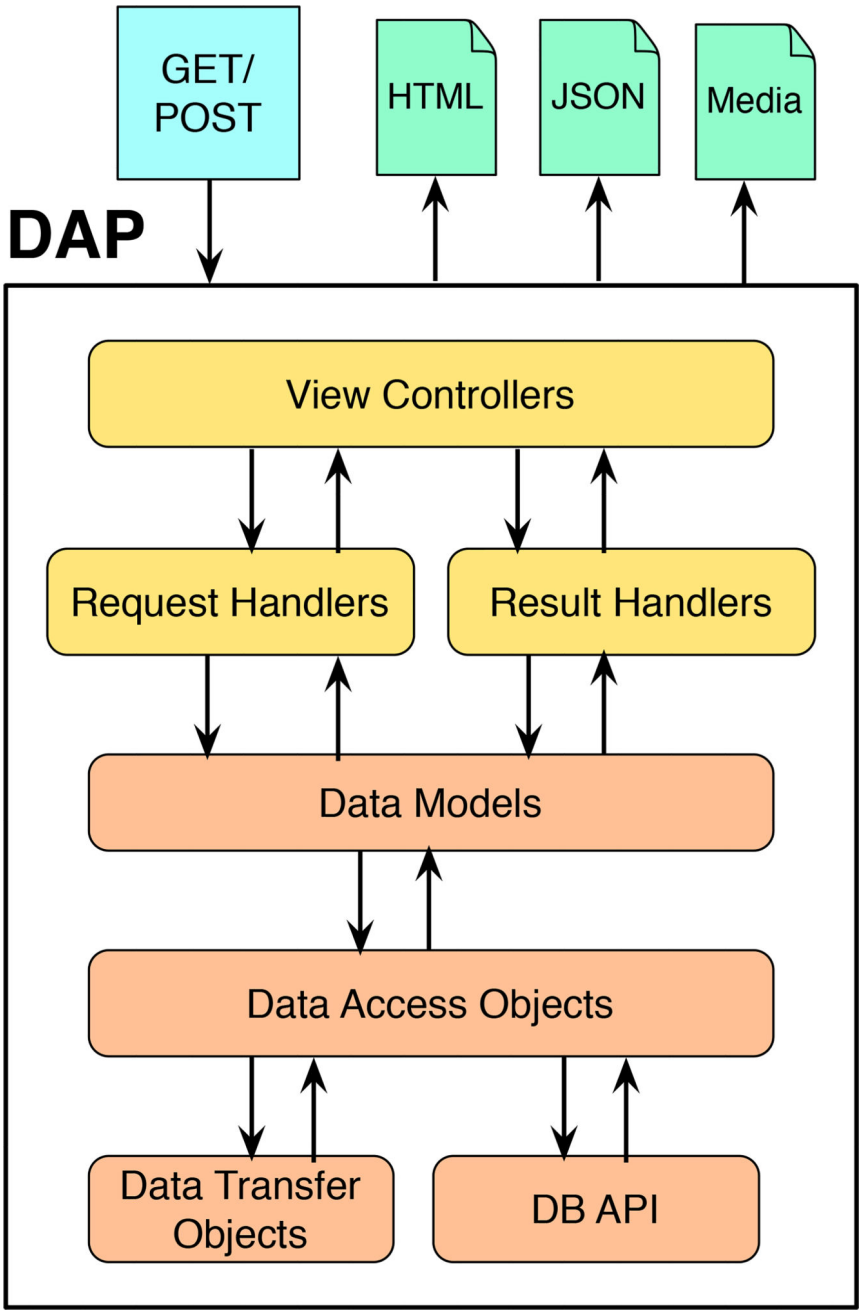

Figure 6.
Organization of the DAP.

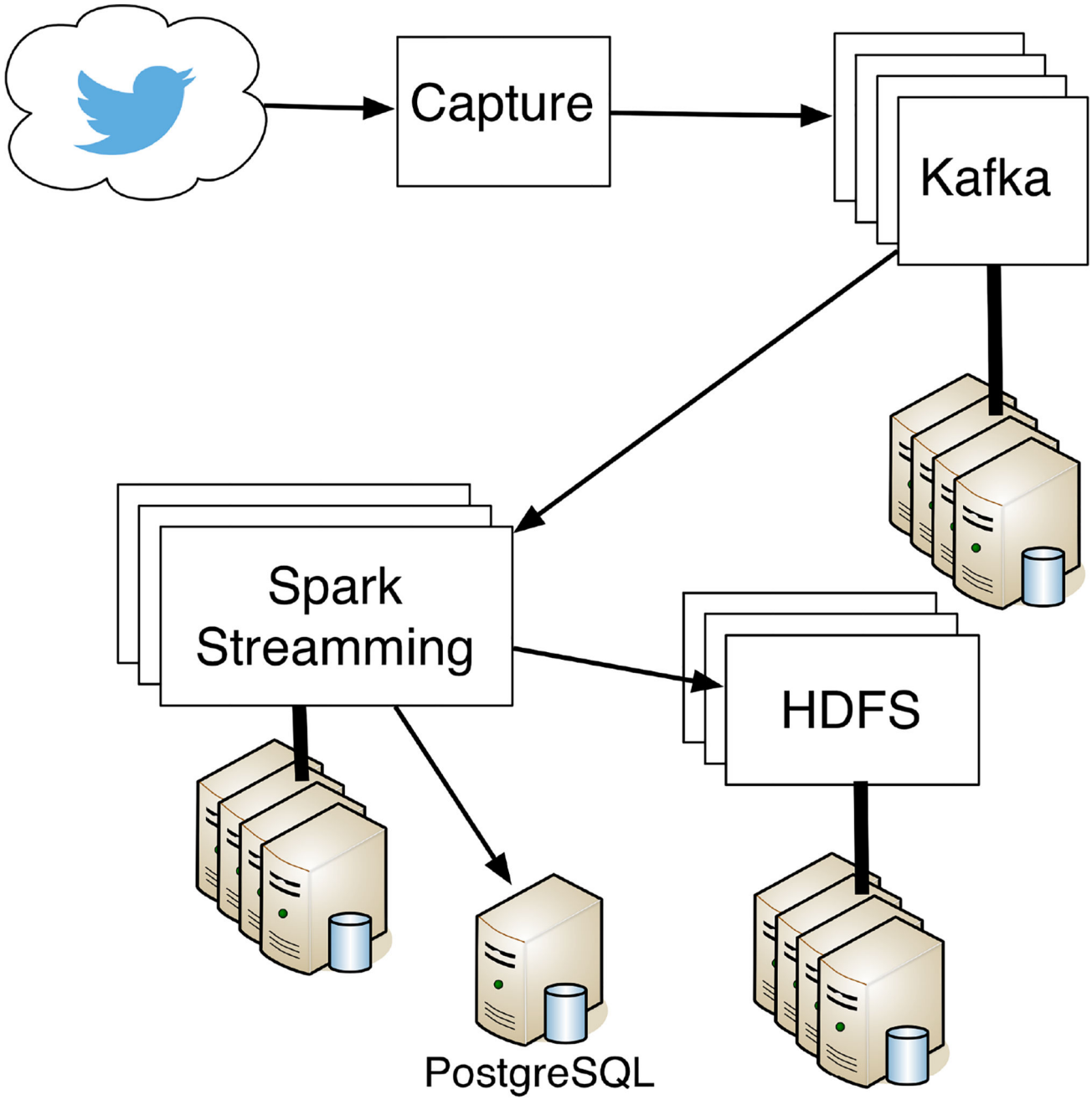

Figure 7.
Current DAC Implementation.