Nimbus: Tuning Filters Service on Tweet Streams

Chien-An Lai [*†], Jim Donahue [*], Aibek Musaev [†], Calton Pu [†]

[*] *Adobe Research, San Jose, CA, USA, 95110*

[†] *Georgia Institute of Technology, Atlanta, GA, USA, 30332*

***Abstract*—With hundreds of millions of tweets being generated by Twitter users every day, tweet analysis has drawn considerable attention for event detection and trending sentiment indication. The problem is finding the few important tweets in this huge volume of traffic. A number of systems provide applications the ability to filter a complete or partial Twitter stream based on keywords and/or text properties to try to separate the relevant tweets from all of the noise. Designing a filter to produce useful results can be extremely difficult. For instance, consider the problem of finding tweets related to the Target Corporation or Guess USA. Just scanning the text of tweets for "target" or "guess" is likely to generate lots of hits, but few really relevant tweets. Nimbus is a service that can be used to tune filters on tweet streams. The Nimbus service builds a database of tweets from a Twitter stream (it does not have to be a full Twitter firehose) and provides an API for testing filters (based on the PowerTrack language and Spark as evaluation engine) against the database. The important feature of Nimbus is that it allows repeatable testing of filter expressions against real Twitter data using the same filter language that can be used against live Twitter streams. This makes it possible for users of the service to tune their filters before putting them into production use.**

I. INTRODUCTION

Hundreds of millions of new contents are created every day on microblog services, e.g., Twitter tweets, Facebook comments and Tumblr posts. Currently, Facebook has over than 1.3 billion monthly active users and over 200 million daily messages sent [1]. Twitter has 645+ million active users who post more than 800 million daily Tweets [2]. In this paper, we will focus on the problem of extracting useful data from the avalanche of Twitter tweets.

Although this huge volume of streaming data is obviously a potentially important information source for a lot of different applications, a very important problems is how to find the few important messages in this huge volume of unstructured or semi-structured data. A number of systems provide applications the ability to filter a complete or partial Twitter stream based on keywords and/or text properties. For instance, GNIP provides its PowerTrack language to allow customers the ability to filter tweets based on keywords, user attributes, geo-location, and many others [3]. Much of the work in tweet analysis has regarded the process of filter generation as a "black box," simply assuming the best set of keywords for a stream filter is known a priori.

In fact, designing a filter to produce useful results can be much more difficult than it looks at the first glance. For instance, consider the problem of finding tweets related to the Target Corporation or Guess USA. Just scanning the text of tweets for "Target" or "Guess" is likely to generate lots of hits, but few really relevant tweets. A user might want to add more keywords into the filter, such like "Target AND Supermarket" or "Guess AND Clothing". The reason that tuning a filter is difficult is there are no obvious rules of thumb to follow. So users frequently find themselves in one of two scenarios: specifying a filter that either matches close to nothing or one that matches almost everything (which turns into a self-inflicted denial of service attack). To make the problem of filter tuning even worse is that current services don't provide users the ability to do controlled experiments. For instance, to test a filter using the Twitter Stream API, the user sees only the current tweets that match the filter. If the user makes a change to a filter, changes in the output reflect both the filter modication and whatever has changed in the stream between experiments – and the user has no way of distinguishing these effects.

In this paper, we present Nimbus: a service for tuning filters on tweet streams. Nimbus builds a database of tweets from a Twitter stream and provides an API for testing filters against the database. Nimbus adopts 1% tweets from Twitter stream for "filter tuning" on a controlled basis. After modifying a filter, the user can test it against the same data to validate the change. Even better, Nimbus allows user to compare multiple filters with the same data set at the same time, updating the matched results and the hit ratios respectively during filter evaluation. The design of Nimbus emphasizes the first-matched response time (the response time between sending a request and receiving the first match result) as well as the completion time (the round-trip response time of request being finished). We don't want users to have to wait for the completed results to understand how good their filters are.

Nimbus uses the GNIP PowerTrack language for filter specification, so a user can directly apply a filter tuned with Nimbus on a full Twitter firehose. The filter can be combined of keywords and logical operators, including "AND", "OR" and "NOT", and it also allows users filtering based on the metadata, such as language and country code. For example, the filter, "Adobe AND Photoshop lang:en", specifies searching the tweets which contain both "Adobe" and "Photoshop" and use English as the language. Nimbus adopts Spark [4] as the parallel evaluation engine and uses ZeroMQ [5] to stream results from database to the client to minimize first-matched response time. In addition, to avoid delays inserting tweets from Twitter into the database, Nimbus uses Spark Streaming to efficiently update the database in small batches of tweets [6].

This work was performed at Adobe Research; J. Donahue is now a CS program advisor at William Jessup University.

978-1-4673-7278-7/15 $31.00 © 2015 IEEE
DOI 10.1109/BigDataCongress.2015.95

Generally, tweets appear in the database within a few seconds of their publication.

The rest of the paper is organized as follows: An overview of architecture of Nimbus is provided in Section II, followed by the description of Nimbus' features in Section III. We also provide some performance figures in Section IV. Finally we introduce related work in Section V, and conclude the paper in Section VI.

II. SYSTEM OVERVIEW

Nimbus is the tuning filter service on tweets streams built on top of a collection of open-source software, including Spark, MYSQL, NodeJS, Spark JobServer and ZeroMQ. Since it is specifically designed to help general users tune their filters on tweets, Nimbus has implemented a few optimizations to improve users' experience. For example, after testing at Adobe, we found in the most cases, users don't need the completed results to tell if their filters are good enough. Hence, rather than focus on the completion response time of users' requests, Nimbus emphasizes the first-matched response time for users from sending the requests and get the first matched tweet instead. For instance, Nimbus uses Ooyala's Spark JobServer to save a few seconds by reusing Spark Context objects instead of instantiating a new Context for every query. In addition, Nimbus uses ZeroMQ to build the communication channel between Spark Workers and the Web Server, so that the users can receive the matched results as early as they are found. These optimization may increase the completion response time relative to simple batch processing, but it improves users' experience by reducing the users' waiting time for the first match.

TABLE I: Hardware Specification for the performance evaluation of Nimbus at Section IV

Item	QTY	Description
CPU	2	Intel Xeon E5-2650 v2 2.60GHz 8-core
Memory	8	MICRON 16GB DDR3 1600 ECC
Network	1	Integrated Gigabit LAN
HD	2	Seagate ST4000NM0033 SATA 6Gb/s 4TB 7200rpm Hard Drive
RAID	1	Integrated SATA RAID 0/1 Controller

In this section, we provide the overview of the system architecture of Nimbus. The hardware we use for the experiments on Section IV is at Table I, while the software stack for Nimbus is shown at Fig 1. Four kind of arrows are specified to represent the transmission direction of synchronous requests, data source, matched results and asynchronous request. Currently, Nimbus is running as a prototype on a single machine, but it is designed to be scaled out in a cluster. In particular, the use of Spark makes Nimbus easy to add more machines as workers to improve the performance of filter evaluation. The role of each software component are described as follows:

A. Twitter Streaming & Spark Streaming

Nimbus gets a randomly sampled 1% tweets from the full Twitter Firehose. To alleviate the pressure of database for writing coming tweets, Nimbus uses Spark Streaming [6] to discretize the tweets stream into small batches; Nimbus then inserts the batches into a MySQL database. The format of the Tweet stream is a stream of JSON objects, as defined in [7]

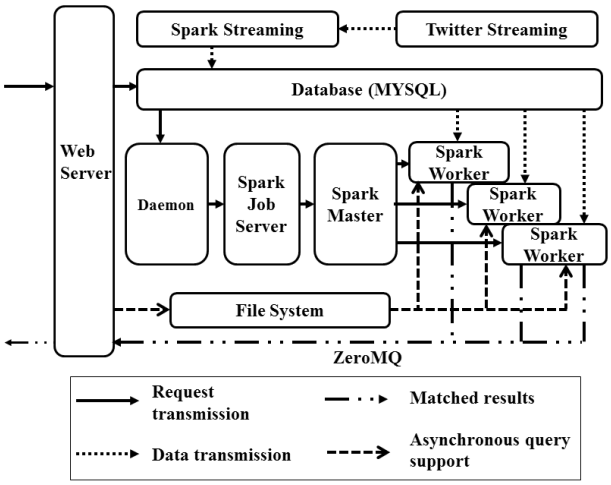


Fig. 1: Software Stacks and Data Flow at Nimbus. Nimbus is built on top of a collection of open-source software, including Spark, MYSQL, NodeJs, Spark JobServer and ZeroMQ. Four kind of arrows are specified to represent the transmission direction of synchronous requests, data source, matched results and asynchronous request.

Nimbus extracts some of the fields of the JSON, such as the creation timestamp, the language of the text, and the geo-information from the original tweet. This information is stored as separate columns in the database (as well as being stored in the JSON BLOB object). This allows some filters to be partially executed by using the database, instead of parsing each JSON tweet object. An experiment shows that using the database incurs the interesting performance tradeoff: the completion response time is reduced by more than 20% when the matched data is scarce, but increases the performance overhead when most of data are matched. We detail the experiment in Section IV-B.

B. Web Server

The web server component of Nimbus adopts Node.js [8], but it does not directly invoke the service backend to run query jobs. Instead, when a job request arrives, it sets up a channel for returning results to the client (using ZeroMQ) and stores the request information in the database. The web server receives query results from the various Spark worker instances (again via a ZeroMQ channel [5]), and forwards them to client as well as storing them in the file system. Saving the results in the file system allows us to support "replay requests," avoiding the costs of rerunning an expensive query. The web server assigns a unique string as the result identifier, selects a port number for receiving results from the Spark query evaluation engine, and inserts them together with the user's request into

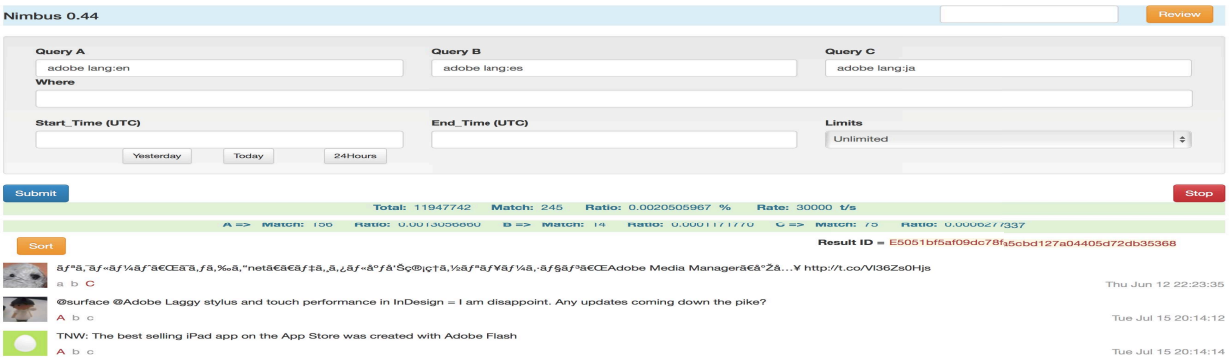


Fig. 2: Screen Shot of Nimbus' Web-UI. Nimbus supports at most three PowerTrack language and a "where filter" on the metadata.Time range and number of matched results can be specified through the Web UI as well.

the database. The result identifier is used as the filename to save the matched results in file system, and it's also forwarded to user so that user can review the result later or share with others. The web server calculates some statistics, like the total number of tweets evaluated already and the point-in-time processing rate, which are also displayed for the user.

C. Storage System

Nimbus' storage system uses both the database and file system. MYSQL is used as the database platform to store the both the tweets and users' requests, while file system is used to save the matched results for users to review or share with others. Streaming tweets are stored as BLOB object with corresponding metadata, like creation timestamp, language and country code. Since Nimbus allows user reviewing or sharing the past search results, the users' request is also stored in the database along with the statistics about its execution, such as total of tweets evaluated. On the other hand, file system is used to store the matched results for reviewing or supporting asynchronous requests later. Files can be written to either the local file system of the machine or to a shared file system like HDFS.

D. Spark JobServer & Spark Cluster

As described above, the Web Server does not directly start new query tasks – instead, it just writes the job information in the database and sets up the channel for returning matching results. The actual request invocation is handled by a daemon process that scans the database for new requests and then starts each request as a Spark job, using the Spark JobServer.

The query execution code is written in Scala. The job task takes as input the query to be performed, the host and port to which results are to be streamed, and a Spark Context. The Spark Context is used to coordinate the execution of the query among the Spark "master" and the collection of Spark "workers" that comprise a Spark Cluster. Spark Contexts are a little expensive to set up, so we use the Job Server to cache Contexts between jobs. This makes job initiation somewhat faster. Each job just applies a shunting-yard algorithm [9] on each filter to produce prefix notation, and then executes evaluation against tweets. This evaluation is done in parallel across all of the workers in the cluster; each worker reports matches to the Web Server as they are found using the ZeroMQ channel set up for result return.

Spark was started as a research project focusing on big data analytics at UC Berkeley [4]. Compared with MapReduce [10], Spark is good at streaming applications that maintain aggregated state over time because it supports low-latency data sharing across multiple parallel operations. In Nimbus, the Spark Master partitions the database and map each partition to a Spark resilient distributed dataset (RDD), in this case a collection of JSON objects. Each Spark Worker is responsible for matching in one or more partitions and matches each tweet in each partition with the user specified filters. When each match is found, the worker sends it back to the WebServer on the ZeroMQ channel. Each worker keeps its own statistics, such as the number of tweets have been evaluated, and transmits to Web Server periodically. These numbers are aggregated by Web Server and updated to users, so that users can understand the job processing status.

III. Nimbus' Features

The goal of Nimbus is to make tuning of filters as efficient as possible. Obviously one important aspect of efficiency is filter matching performance over a large database. To this end, we provide mechanisms to limit the tweets selected from the database for match-testing; this includes selecting based on metadata and time values. Also, since filter evaluation over databases of hundreds of millions of tweets is going to be time-consuming, the filter evaluation results are stored in files and can be "replayed" by just presenting the request ID that was generated when the request was first made; these replay results can also be sorted by various criteria to make it easier to understand the results.

The Web UI of Nimbus is as shown in Fig 2. Nimbus allows users to search tweets using the PowerTrack language for filter expression. The PowerTrack language is composed of a set of keywords, logical operators, including "AND" "OR" "NOT", metadata filters like "lang:en", and the parenthesized expressions. Hence, users can look for what they want exactly

by complex filters, such as "((happy OR party) (holiday OR house) NOT(birthday OR democratic OR republican)) AND lang:en". In the Web UI, users are also able to use PowerTrack language to specify a "where filter" on the metadata. This filter is applied as a SQL WHERE clause to filter the database before doing detailed filter evaluation. The tradeoff of performance for evaluation at different components, database or Spark Cluster, will be discussed more at Section IV-B. In addition, a time range can be specified through the Web UI by choosing from built-in calendar or by short cut buttons; this time range is also included in the WHERE clause used to "pre-filter" the tweet database. Users are also able to limit the number of matched results return in case of being overwhelmed by receiving thousands of matched results (the equivalent of the SQL LIMIT clause). Moreover, every result is assigned a unique result ID and shown on Web-UI as well. With the result ID, users can review or even share the results without rerun the query again. For these asynchronous queries, users can also sort the results by some criteria, like timestamp or number of retweets. We illustrate some of the most important feature of Nimbus in the section in the following.

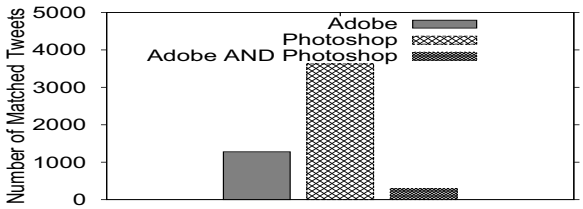
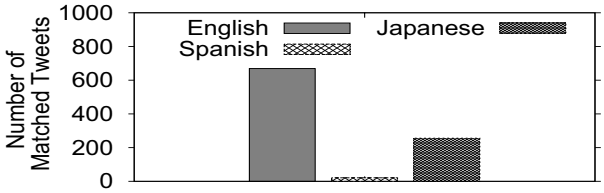


Fig. 3: Over 42M tweets, "Photoshop" is mentioned about three times as much as "Adobe" on tweets streams, while about 25% tweets which contains "Adobe" also mentioned "Photoshop" in the content.

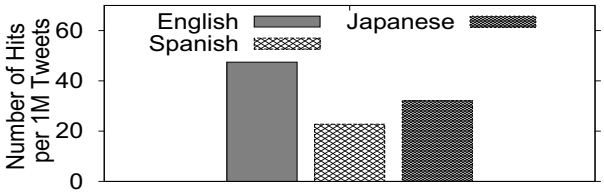
A. Multiple filters one time

So far, Nimbus supports at most three filters for evaluation at the same time, so that users can easily compare the differences, such as hit ratios, among different filters. Besides, to make users easily understand the current status of query processing, Nimbus updates the statistical information every 500 millisecond. These information are listed by different categories: overall information and respective information for each filter. Overall information includes the total number of tweets have been evaluated, the number of matched results have been found, the overall hit ratio, and the point-in-time evaluation rate, while respective information shows the number of matched tweets and the hit ratio for each filter. For each tweet which matches at least one of the filter, Nimbus shows the image of user, the content of text, the match filter, and the creation timestamp on Web-UI. These real-time information help users tell if their filters are good enough as early as possible; we would prefer not to have the users need the completed results to make a decision.

For example, a user might be interested on the difference among "Adobe", "Photoshop", and "Adobe AND Photoshop".



(a) Number of tweets mentioned "Adobe" in different languages.

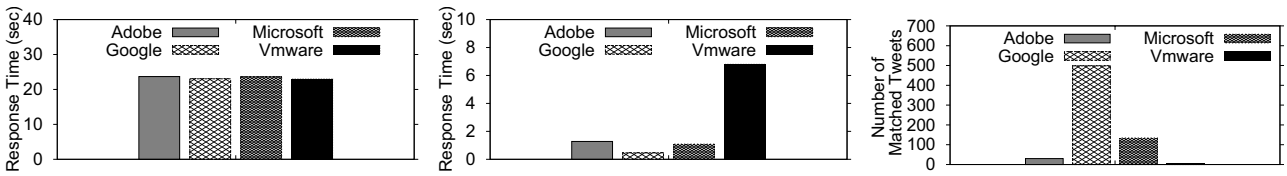


(b) The ratio of the popularity of Twitter for different languages among English.

Fig. 4: Among 42M tweets, "Adobe" is mentioned the most by English users and 10 times more in Japanese than in Spanish in Fig 4a. Fig 4b shows the reason is the popularity of Twitter varies in different regions.

By Nimbus, he/she can clearly tell "Photoshop" are mentioned about three times as much as "Adobe" on tweets streams, while about 25% tweets which contains "Adobe" also mentioned "Photoshop" in the content. The number of matched tweets over 42M for these three filters is shown in Fig 3. The evaluation against three filters over 42M tweets takes less than 30 minutes in our prototype implementation (which has not gone through substantial performance tuning). More detail about the performance of Nimbus will be discussed in Section IV.

Since Nimbus implements the complete PowerTrack language, users are not only able to tell the difference among the sets of keywords, but also among the different metadata filters. For instance, users could investigate among three languages including English, Spanish and Japanese, in which language Adobe is mentioned the most of time by keyword "Adobe" with three language filters "lang:en" and "lang:es" and "lang:ja" at Nimbus. We notice that although Adobe happens to be the same spelling for Spanish, Japanese refer to Adobe also in Kanji. We had tested Kanji in Nimbus but nothing was found, so we decided to omit it. Besides, "Adobe" is also an English word to represent a house made of earth/soil. We did observe a tweet like "there is no life without adobe", which is hard to tell if the use referred adobe as a company or the other meaning; however, we didn't exclude it since it's a rare case. The study of telling the meaning of words is out of the scope in this paper. Fig 4a shows the result for three filters, "Adobe lang:en", "Adobe lang:es" and "Adobe lang:ja". Among over 42M tweets, the keyword "Adobe" is mentioned the most by English users, 669 time specifically. Interesting, "Adobe" is mentioned 10 times more in Japanese than in Spanish. Users might conclude that Adobe is drawing attention in Japan than in Spanish-speaking countries. However, with the



(a) The completion response time for searching four company names among 1M tweets.

(b) The first matched response time for searching four company names among 1M tweets.

(c) The number of matched results for different company names among 1M tweets. Only 5 of 1M tweets contains "VMware".

Fig. 5: Nimbus evaluated 1M tweets against four company names: "Adobe", "Google", "Microsoft" and "Vmware". Nimbus finished the evaluation with 30 seconds, and it returned the first matched in two seconds in most of cases. Although only 5 of 1M tweets contains "VMware", Nimbus demonstrated its efficiency by returning the first-matched tweet within 7 seconds.

filters "lang:en", "lang:es", and "lang:ja", Nimbus shows the reason of Fig 4a is actually because the popularity of Twitter for different languages as shown in Fig 4b. The distribution of Japanese and Spanish in Fig 4a approximately follows the distribution in Fig 4b. But the hit ratio of keyword "Adobe" is much higher in English than in Japanese.

With Nimbus, users are able to compare the difference among at most three filters at the same time. Users can test the different sets of keyword as shown in Fig 3, and they can test the different metadata filters like Fig 4. Moreover, with combined of the results of multiple filters against the same dataset, Nimbus shows the potential as event detection or trend analyzer on tweets streams.

B. Real-Time Update of the Tweet Database

Although the design of Nimbus doesn't particularly focus on the latency between when a tweet is created and when it is searchable, Nimbus does provide "near real-time" updating of the tweet database. The tweets from our 1% Twitter stream are searchable in Nimbus within a minute of their creation. This latency is introduced by our Spark Streaming code, which batches the tweets every few seconds (there are generally a few hundred in a batch), extracts some fields from original tweets as metadata, and writes each batch into the database. Batching the updates definitely helps improve the database performance overall with only a small delay in seeing new tweets.

In general, Nimbus stores around 4M tweets in the database everyday. Our plan is to keep at least three to six months of tweets during the experimental phase of the project.

C. Sorting and Asynchronous Query

The queries mentioned above are synchronous queries, in which users receive the matched results as soon as any can be returned by the Spark Cluster. Nimbus also supports asynchronous queries, where partial results can't be immediately returned. This allows us to adding sorting to the feature set. If the client specifies a sort criterion in the Web UI, we take advantage of the Query Replay mechanism (described below) to collect results, sort them, and then make them available for later retrieval.

The sorting feature takes advantage of Spark to do the sorting across all of the worker nodes in the Spark Cluster without having to write the non-sorted results to external storage. The normal processing pipeline involves doing a Spark "map" operation on the rows in the database (to record which filters match the message), followed by a "filter" to discard all the non-matching rows – the remaining rows are returned to the client. To add sorting, all we need to do is to add a Spark "sortByKey" to the pipeline to get the messages sorted by the chosen key. And, after completing the sort, we can then save the (sorted) results for replay.

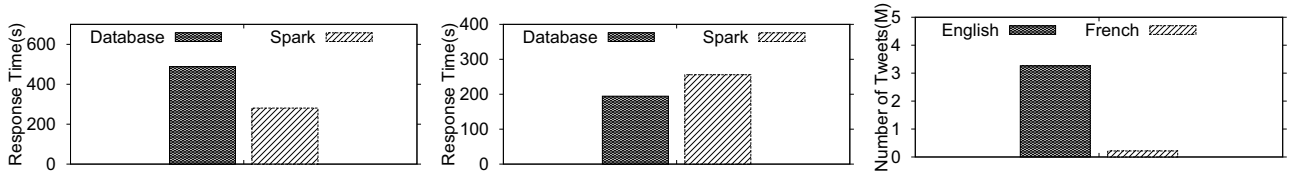
D. Query Replay

Nimbus allocates a unique result ID for each user's query. The result ID will be recorded with user's query together into the database, so that users can review the results as well as the query in the future. The result ID is used as the filename to save the result in file system for asynchronous query later, and it is also shown in Web-UI for users to review or share with others. This feature is important especially when the query is against a large dataset. Although Nimbus is equipped with powerful query evaluation engine, approximately processing 40,000 tweets per second by 4 cores and 16GB memory, it still takes minutes to finish the evaluation for millions tweets. Since Nimbus continues to update the tweets database, the completion response time will keep increasing as well. With the feature to reviewing historical results, users won't need to rerun the query again to get the same results, and they can also save other people's time by sharing the results.

The results stored in file system as a file is not only including the original JSON object, but also three letters to show the evaluation results for three filters. The other information, such as three filters, where constraint, selected time period, total number of evaluated tweets, is stored in database with the unique result ID together. With the stored information, Nimbus is able to reconstruct the results in seconds as if user executes the query again.

IV. PERFORMANCE EVALUATION & OPTIMIZATION

In this section, we evaluate the performance of Nimbus by different aspects. Currently Nimbus is running on a single machine as we described in Table I, but it's sufficient to support tuning service with one percent tweets streaming data.



(a) The completion response time of "English" as the language filter. Spark approach is about 57% faster than Database approach.

(b) The execution time of "French" as the language filter. This time, Spark approach is about 31% slower than Database approach.

(c) Against 10M tweets, Spark Cluster receives 32% tweets for English case, while it receives less than 1% tweets for French case.

Fig. 6: Performance comparison when language filters applied at database and spark respectively. Database can be efficient to search the tweets by index if the density of target tweets is low. However, it introduces extra overhead if the density is high.

With the higher arrival rate of tweets in the future, Nimbus is designed to run each component on individual machine and connect with other components via connectors, such as JDBC and ZeroMQ. Concretely, the following results show Nimbus can provide high throughput and response time of the first match in few seconds.

A. How Fast Can User Receive the First Match?

In our first experiment, we estimate the response time of Nimbus with 1 million tweets under four different filters, including "adobe", "google", "microsoft" and "vmware". We category two kind of response time: first matched response time and completion response time. First matched response time represents the time between the user issues a request and receives the first matched result, while completion response time represents the duration between issuing request and receiving the completed results over 1 million tweets.

Fig 5a shows the completion response time for the four filters with 1M tweets. As we can see, 1M tweets can be evaluated around 23 seconds no matter what the distribution of matched tweets is. Fig 5b shows the first matched response time under the four different company name as the filters. The first match of Google and Microsoft is returned within one second, while the first match of Adobe is returned within 1.5 second. However, the first matched response time of VMware is at least 4 times than the other companies. With more investigation, we found the first matched response time is highly correlated with the number of matched tweets existing in 1M tweets. Fig 5c shows the number of matched tweets in these 1M tweets. Although Nimbus can evaluate thousands of tweets per second, but since the tweets mentioned VMware is few, Nimbus still takes time to find the first match.

B. Does Pre-stored Metadata help Performance?

In the second experiment, we estimate the performance tradeoff between two different language filters applied at database and spark respectively over 10M tweets. Since Nimbus preprocesses the streaming tweets and stores associated metadata, such as language and country code, with JSON objects in the database, it provides two timing to filter to apply language filter on the tweets. One is to use metadata in database, another one is to evaluate in Spark Cluster. In this experiment, we test these two approaches with the same keyword "Adobe" but two language filters, English and French, and the results show interesting performance tradeoff in Fig 6a and Fig 6b. To utilize the index feature, we build an index on language column in database.

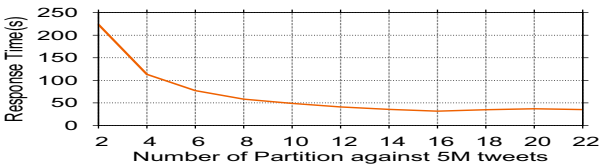
Fig 6a shows doing language evaluation at Spark Cluster is faster than doing it at database, while Fig 6b shows the opposite trending. We believe the reason of the interesting performance difference is the density of matched tweets in database. Fig 6c shows the number of tweets is evaluated at Spark Cluster for different scenarios. Without the database filtering the tweets first, all of the 10M tweets would be sent to Spark Cluster for filter evaluation in both cases. If we choose to use database to filter the language first, Spark Cluster receives 32% tweets when English as language filter, while it receives less than 1% tweets when French as language filter. These results shows database approach can be efficient to search the tweets by index if the density of target tweets is low. However, it introduces extra overhead and hurts the performance if the density is high.

C. How to Partition Database to Improve Performance?

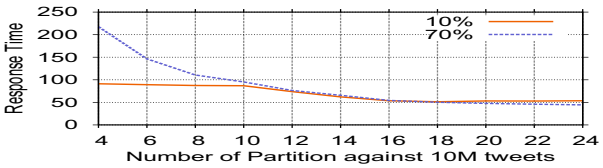
One of the important ways to speed up processing in a Spark cluster is to improve the parallelism among the workers – the more you can keep all the workers busy, the better the performance will be. In the case of Nimbus, one way to improve parallelism is to increase the number of "database partitions."

In Spark, a "resilient distributed dataset" (RDD) can be split into multiple partitions. The Spark driver will schedule workers to handle partitions in parallel to improve performance. In the case of Spark RDDs that are backed by a database, partitions can be created by spliting the key values for the rows into equal-sized buckets – each partition corresponds to all the rows with keys in the corresponding bucket. In the case of Nimbus, we have an auto-incremented ID field as the key for each tweet stored, so it is easy for us to create as few or as many partitions as we want.

In this experiment, we varied the partition from 2 to 22 against 5M tweets to see if the number of partition is a factor for Nimbus' performance. We loaded these 5M tweets into the cache memory of database first to avoid the Disk IO bottleneck. From the results shown in Fig 7a, we found that increasing the number of partition can really improve the



(a) The response time to evaluate 5M tweets at different number of partition scenarios.



(b) The response time for unbalanced data distribution at different number of partition scenarios.

Fig. 7: The response time decreases 84% when the number of partition increases from 2 to 22 in Fig 7a. In unbalanced data distribution scenario, the response time are reduced 75% for 70% data selected by database and 43% for 10% case in Fig 7b.

performance by parallel evaluation processing until the system is saturated, which is around 16 partition in this case.

This corresponds well with the general guidelines for writing Spark applications, which suggest using as many partitions as possible until the overhead (in particular, network communication) of running a task on a partition becomes significant relative to the actual processing done for the partition. Sixteen partitions of 5M tweets means an average partition size of around 300K tweets/partition – having more partitions just drops the processing costs so that the overhead begins to dominate. We are also generally running a four-worker configuration, which means that at 16 partitions we get an even split of partitions/worker.

Another benefit of database partition is that it mitigates the performance degradation when the distribution of data is not uniform. At Nimbus, the database is partitioned based on assigned id equally. However, since Nimbus filters the data with metadata first, each partition probably receives different number of tweets from database. In this case, the completion response time is decided by the partition with the most tweets. In this experiment, we estimate if the performance can be improved by increasing the number of partition at the unbalance data distribution scenario. Specifically, the database evaluates 10M tweets, and only the first n% tweets are sent to Spark Worker for filter evaluation since the metadata meet user's query. In this case, some partitions receives a lot of tweets, while the others receives zero. In this experiment, the threshold is set at 10% and 70% respectively to cover different cases, since we have understood Nimbus' performance is affected by the threshold from Section IV-B. According to the result in Fig 7b, we found out increasing partition helps system improve the performance when the distribution of data is unbalanced. Concretely, the completion response time is reduced at most 75% for 70% threshold and 43% for 10% threshold when the number of partition increases from 4 to 24.

V. BACKGROUND AND RELATED WORK

A. Search on Tweets

Compared with web searching, people are inclined to search micro-blogging social media for temporally relevant information, such as breaking news and popular trend [11]. The challenge of searching on this kind of service, like Twitter, is thousands of concurrent users may tweet simultaneously. Hence, it's important to process new content quick enough to make them be available for searching shortly after creation. A lot of efforts have been made for real-time search service on tweets to enable users to know what's happening *right now* in the world.

For example, Nimbus can be used to pre-filter data from Social Networks by LITMUS [12], a landslide detection service based on physical sensors and social networks. Starting with labeling data via its own filtering component, LITMUS identifies the relevance of data items and applies machine learning classification to recognize landslide as a natural disaster. Since Nimbus can efficiently handle large amounts of incoming data and supports complex filtering, then it is an obvious choice that will improve LITMUS performance.

Twitter's Earlybird provided real-time search service in which tweets were searchable within ten seconds after creation without consideration of ranking [13]. Earlybird improved the efficiency of inserting new tweets by limiting the scope of index updated and using a single-writer multiple-reader lock-free model. Based on Earlybird, Twitter introduced real-time related query suggestion and spelling correction service to provide the relevant results of breaking news events within minutes [14]. The paper also described the problems of using Hadoop for low-latency processing on big data, which inspired Nimbus using Spark as in-memory processing engine [4].

Rather than returning the top K tweets, Log-Structured Inverted Indices (LSII) maintained a sequence of inverted indices with exponentially increasing size to guarantee the completeness of query results [15]. New tweets are first inserted into the smallest index and moved to the larger indices later in a batch manner. Nimbus is designed to return the full set of query results to the users without considering any ranking, but it also allows users sorting the results by specified criteria.

B. Query Language on Tweets

Keyword matching is the mainstream method for real-time microblog searching [16] [13] [15], despite the difference on accuracy, ranking function and memory management. Because of the very high arrival rate in microblog services, a lot of works focus on spatio-temporal queries over [17] [18]. Including associated geo-location, created timestamp, as well as text description, spatial-temporal keywords queries return top‘k relevant results. Recently, Chen et al. proposed a benchmark

and provided an empirical study to evaluate twelve spatial keyword query performance [19].

On the other hand, other researches investigated the possibility to utilize RDBMS on semi-structured JSON objects. TweeQL is a SQL-like query language interface to generate structured data from unstructured tweets [20]. Users are able to issue SQL-like query with keyword, location, or userid filters on top of Twitter Stream API and receive match results. Moreover, Chasseur presented an automated mapping layer for storing JSON data in a relational system [21], while Liu et al. explored the principles to manage JSON data in RDBMS [22]. These works offer significant benefits to application developers as they can use SQL to query both relational and JSON data.

Nimbus chooses to implement PowerTrack language [3], including keywords, logical operators and tagged metadata, rather than SQL. Twitter acquired Gnip in April, 2014 [23], so it's likely that PowerTrack will become the filter language for Twitter API in the future. This made it a natural choice for Nimbus.

VI. CONCLUSION & FUTURE WORK

In this paper, we have described Nimbus, a service for tuning the filters used to extract useful information from the mountain of Twitter tweets generated every day. The design of Nimbus was based on some simple observations:

1) To make "filter tuning" practical, the filter language used by Nimbus had to be precisely the same "production" language used to filter a full Twitter Firehose. Thus, Nimbus implements the full GNIP PowerTrack language and exposes PowerTrack explicitly through our Web UI. So we can reasonably promise that "what you see should be what you get."
2) Because any kind of testing on tens or hundreds of millions of tweets is going to be expensive, the service needs to be carefully designed to be responsive. When testing a filter, results should be returned as quickly as possible (so a user can terminate a test as soon as the results are obvious). And keeping past test results around for doing efficient replay is a necessity.
3) Because we're interested in building a service managing large amounts of data and handling many users, it needs to be built from loosely coupled components that can be independently scaled. Spark is an important part of the design because it allows us to easily spread processing load across as a potentially large cluster while maintaining the responsiveness the service requires.

Our current prototype, as described above, shows the promise of the design. It would be relatively easy to extend the PowerTrack language to allow keyword specifications to also include a part of speech and to hook up our Spark filter evaluation engine to a syntactic analyzer to get part of speech information as part of filter evaluation. One obvious concern is the implications for performance and responsiveness, which will be a critical part of any extension we make along these lines.

ACKNOWLEDGMENT

This research has been partially funded by National Science Foundation by CNS/SAVI (1250260, 1402266), IUCRC/FRP (1127904), CISE/CNS (1138666, 1421561) programs, and gifts, grants, or contracts from Fujitsu, HP, Intel, Singapore Government, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

REFERENCES

[1] S. Brain, "Facebook Statistics," http://www.statisticbrain.com/facebook-statistics/.
[2] ——, "Twitter Statistics," http://www.statisticbrain.com/twitter-statistics/.
[3] I. GNIP, "The Overview of PowerTrack Language," http://support.gnip.com/apis/powertrack/overview.html.
[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012, pp. 15–28.
[5] F. Akgul, *ZeroMQ*. Packt Publishing Ltd, 2013.
[6] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: fault-tolerant streaming computation at scale," in *SOSP*, 2013, pp. 423–438.
[7] Twitter, "Field Guide of Tweets," https://dev.twitter.com/docs/platform-objects/tweets.
[8] S. Tilkov and S. Vinoski, "Node.js: Using javascript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010.
[9] E. W. Dijkstra, *A short introduction to the art of programming*. Technische Hogeschool Eindhoven, 1971, vol. 4.
[10] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.
[11] J. Teevan, D. Ramage, and M. R. Morris, "#twittersearch: a comparison of microblog search and web search," in *WSDM*, 2011, pp. 35–44.
[12] A. Musaev, D. Wang, and C. Pu, "LITMUS: a Multi-Service Composition System for Landslide Detection," *IEEE Transactions on Services Computing*, vol. PP, no. 99, 2014.
[13] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, "Earlybird: Real-time search at twitter," in *ICDE*, 2012, pp. 1360–1369.
[14] G. Mishne, J. Dalton, Z. Li, A. Sharma, and J. Lin, "Fast data in the era of big data: Twitter's real-time related query suggestion architecture," in *SIGMOD Conference*, 2013, pp. 1147–1158.
[15] L. Wu, W. Lin, X. Xiao, and Y. Xu, "Lsii: An indexing structure for exact real-time search on microblogs," in *ICDE*, 2013, pp. 482–493.
[16] C. Chen, F. Li, B. C. Ooi, and S. Wu, "Ti: an efficient indexing mechanism for real-time search on tweets," in *SIGMOD Conference*, 2011, pp. 649–660.
[17] W. Liu, Y. Zheng, S. Chawla, J. Yuan, and X. Xing, "Discovering spatio-temporal causal interactions in traffic data streams," in *KDD*, 2011, pp. 1010–1018.
[18] A. Skovsgaard, D. Sidlauskas, and C. S. Jensen, "Scalable top-k spatio-temporal term querying," in *ICDE*, 2014, pp. 148–159.
[19] L. Chen, G. Cong, C. S. Jensen, and D. Wu, "Spatial keyword query processing: An experimental evaluation," *PVLDB*, vol. 6, no. 3, pp. 217–228, 2013.
[20] A. Marcus, M. S. Bernstein, O. Badar, D. R. Karger, S. Madden, and R. C. Miller, "Tweets as data: demonstration of tweeql and twitinfo," in *SIGMOD Conference*, 2011, pp. 1259–1262.
[21] C. Chasseur, Y. Li, and J. M. Patel, "Enabling json document stores in relational systems," in *WebDB*, 2013, pp. 1–6.
[22] Z. H. Liu, B. C. Hammerschmidt, and D. McMahon, "JSON data management: supporting schema-less development in RDBMS." in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, 2014, pp. 1247–1258.
[23] Twitter, "Twitter welcomes Gnip to the flock," https://blog.twitter.com/2014/twitter-welcomes-gnip-to-the-flock.