A Platform for Analysing Stream and Historic Data with Efficient and Scalable Design Patterns

Rebecca Simmonds
School of Computing Science
Newcastle University
r.m.simmonds@ncl.ac.uk

Paul Watson
School of Computing Science
Newcastle University
paul.watson@ncl.ac.uk

Jonathan Halliday
Red Hat
jonathan.halliday@redhat.com

Paolo Missier
School of Computing Science
Newcastle University
paolo.missier@ncl.ac.uk

***Abstract*—Social media is an increasingly popular method for people to share information and interact with each other. Analysis of social media data has the potential to provide useful insights in a wide range of domains including social science, advertising and policing. Social media information is produced in real-time, and so analysis that can give insights into events as they occur can be particularly valuable. Similarly, analytics platforms providing low latency query responses can improve the user experience for ad-hoc data exploration on historic data sets. However, the rate at which new data is generated makes it a real challenge to design a system that can meet both of these challenges. This paper describes the deisgn and evaluation of such a system. Firstly, it describes how a meta-analysis of the types of questions that were being asked of Twitter data led to the identification of a small set of queries that could be used to answer the majority of them. Secondly, it describes the design of a scalable platform for answering these and other queries. The architecture is described: it is cloud-based, and combines both continuous query, and noSQL database technology. Evaluation results are presented which show that the system can scale to process queries on streaming data arriving at the rate of the full Twitter firehose. Experiments show that queries on large repositories of stored historic data can also be answered with low latency. Finally, we present the results of queries that combine both streaming and historic data.**

***Index Terms*—Complex event processing, Distributed database, NoSQL, Scalability and Social Media.**

I. Introduction

Social media systems such as Twitter provide ways for millions of users to publish information and interact with each other. Therefore analysing social media data to identify interesting events and patterns as they happen, or in historic data, can provide insights into many domains including social science, advertising and policing. However, with social media networks such as Twitter averaging 5,000 tweets per second and reaching peaks of over 100,000 [1], storing and processing the data presents a major challenge. In this paper, we describe our work to address this challenge: the design and evaluation of a scalable system for social media analysis. Focusing initially on Twitter, we were interested in developing a platform providing comprehensive analysis, which would support any type of query pattern. However, given the importance of performance scalability, we needed to understand which types of queries would mainly be used to optimise the system. These were identified by conducting a meta-analysis of the types of questions that were being asked of Twitter data in academic literature and the media. This small set of query patterns can be used for 90% of use cases. A scalable system was then designed and developed that was optimised to answer these queries, by using efficient indices and denormalisation patterns. This requires data to be processed using three methods: stream, historic and combined. The first "stream" queries the data as it arrives from the firehose: the challenge we met here was to be able to query data arriving at the full rate of the Twitter firehose. An example is to identify trends as they occur. The second "historic" requires the ability to store data arriving from the Twitter firehose, while simultaneously running queries against what may be years-worth of stored data (at the current rate, 150 billion tweets will be generated this year). An example is to find how many times a hashtag occurred over a time-period that is in the past. The third "combined" [2] uses both stream and historic analysis techniques together, and can lead to interesting cases in which queries on historic data are triggered by a pattern of events in the incoming stream. Designing a system that can scale to handle the full arrival rate of events from the Twitter firehose while simultaneously answering user queries against the stream and historic data is challenging. As will be shown this was overcome by improving database performance with the use of indices. Denormalising the data supports quick reads and writes so they can be executed simultaneously. Combined queries answer questions such as identifying all tweets that have been retweeted more than a certain number of times in a time-period that stretches from the past to the future. The main contributions of this paper are: (i) A platform which optimizes read and write performance to combine historic and stream analysis with low latency reads and high throughput of writes. (ii) Efficiently designed indexes to provide aggregate and exact match querying patterns. (iii) A system was deployed in the cloud, and achieves scalability by horizontally scaling the database across a set of cloud nodes. (iv) The resultant system has now become a tool to enable easier access to Twitter research and analysis for non-expert users.

The paper is organised as follows. After reviewing related work (II), we outline the case studies (III) used to derive the generic queries (IV). The architecture of the system designed to scalably process the queries is then described (V), followed by an evaluation that demonstrates the ability of the system to

978-1-4799-5069-0/14 $31.00 © 2014 IEEE
DOI 10.1109/SERVICES.2014.40

meet the performance challenges (VI).

II. Related Work

The emergence of cloud computing has provided cost-effective, pay-as-you-go capacity to economically process big data. It has encouraged the exploration and analysis of data sets collected from the Internet, which has led to research investigating social media. Twitter is a huge producer of data, which is available by connecting to the firehose. Research investigating Twitter analysis has previously concentrated on various components of the tweet itself. [3] rates news sites by counting their URL mentions, [4] counts the number of followers that a user has, exploring celebrities and their followers, [5] has focused on defining trends by the co-occurrence of words within tweets and [6] describes a mechanism for determining popular messages on Twitter so they can be forwarded to people who are following fewer users. Other components of tweet metadata that are being investigated are gender and location [7]. Twitter can be used as an opinion-sharing network, to establish online tensions, with [8] using sentiment analysis to track racism in football. As an opinion-sharing network Twitter does not guarantee reliability of the content of its messages; [9] questions this by asking how tweets mirrored the real-life results of the NBA playoffs.

The techniques of querying and identifying events in Twitter are also split into two techniques; namely stream and historic analysis. Real-time counting, (e.g. URLs and Twitter clicks) has been introduced by a commercial system called Rainbird [10]. Twitter have recently released a system providing batch, stream and hybrid processing of data (hybrid being a combination of batch and stream processing) [11]. Batch processing of historic data can be a long process. However, our system enables low latency queries over the historic store to support the combination of stream and historic data. Another system uses realtime counting and then stores the information for batch processing [12]. Twitter data is used in [13] for stream event detection of road accidents.

Much of the literature focuses on application-specific querying of Twitter data, unlike the system shown in this paper which provides a wider scope of querying. The queries enable a core set of questions, which can be used in different application areas. [10] and [12] limit their querying to only agrregate functions, our system uses aggregate functions and time series data to answer questions. [11] provides the three techniques of querying (historic, stream and composite), however our system aims to improve this by querying historic data as if it were stream. [13] shows stream processing, we go further and allow a combination of historic and stream processing.

III. Case Studies

We reviewed the questions being asked of Twitter data in order to provide insights into a range of domains. From these, we derived a set of generic queries that could be used to answer key questions in these domains (Section IV). In this section we give examples of ways in which information was being derived from tweets in three domains: marketing and advertising, conferences and emergency response.

A. Marketing and Advertising

The need to understand the popularity, reach and spread of tweets and topics contained within them is important commercially [14]. For example, [15] highlights some of the important questions that companies want answering. This includes monitoring tweets with specific keywords, and using retweets to identify the popularity of certain marketing items. Examples of questions that have been identified within this domain are:

1) What is currently trending? [5]
2) Did this hashtag trend yesterday? [14]
3) What retweets will be popular in the next hour? [5]

B. Conferences

In conferences and presentations, the use of a common hashtag, along with sometimes a dedicated Twitter profile, is commonplace. This is used as a method of publicising the event, as well as monitoring attendee and external engagement [16]. Some common questions from this application domain are:

1) What has been posted about this conference? [17]
2) What are people's opinions on this paper? [18]
3) Which aspects of the conference have been the most popular? [6]

C. Emergency Response

Emergency response encompasses a range of types of incidents, e.g. from natural disasters such as floods and earthquakes, to crimes in your neighbourhood. Social media is emerging as an important tool to support policing. [19] explains in depth the important role that social media has when responding to emergencies. Key questions that have been asked in this domain are:

1) What information is available about this flood? [20]
2) What tweets were sent from the city that the earthquake originated from? [21]
3) Which retweets were posted most during the riots? [22]

IV. Generic Queries

A. Data Model

To enable the querying of Twitter data to answer common questions, a data model was designed to capture the relations between the data types in Twitter, and is illustrated in Figure 1.

We faced two main challenges; firstly, to provide a scalable means of answering such queries even if there were a large number of users or applications simultaneously generating them; secondly, to provide a simple way for non-expert users to ask queries about tweets. We addressed both these issues by identifying a small core set of generic queries that could answer the questions from the case studies, highlighted in section III. A scalable, cloud-based system was then designed and deployed to answer those queries.

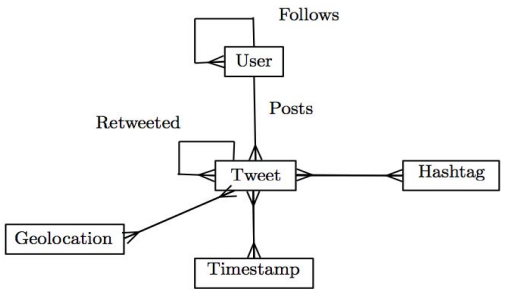

Figure 1: Twitter data model

Parameters	description
hashtag	topics in a tweet prefixed by #
count	a tool used to store the number of times an event has occurred
threshold	an integer that represents a total the count is constraind by
location	a string representing the location of a tweet
retweet	a tweet that has been reposted by another user
time	this is a timestamp of when a tweet was posted

Table I: Parameters and select values

B. Queries

The conceptual design of the queries is defined in the continuous query language, CQL [23]. This provides a formal method of describing the queries, which has standardised semantics including temporal definitions. One novelty of our approach is that we provide a uniform interface that allows the user to specify the time-bound of the query in a standard way, irrespective of whether the time period begins and ends in the past, present or future. It is the system not the user that decides whether to query from the stream and, or the historic data. The system therefore consists of two main components. One can query the stream of incoming tweets. The other stores all those tweets in a database and supports queries on them. We now describe the queries. In these descriptions, terms that are used are listed in table I. The queries cover a range of aggregate functions, exact matches and composite queries. In the CQL we use "Tweets" as a placeholder, for either stream and, or historic data, irrespective of the type of query (table II). A noSQL database called Cassandra is being used, schema and query design differs from relational databases. Queries are derived and used to define the schema, unlike traditional RDBMS. Therefore this section will describe the queries, then the schema will be described in section V.

Query One identifies trends and returns the resulting hashtags and their count. This query takes three parameters, which specify a set of upper and lower time bounds and a threshold limit. The value of threshold specifies how many times a hashtag must occur between the time bounds before it is identified as a trend. The counter is incremented each time a duplicate hashtag in the same time-bound is mentioned in a tweet. A counter is a Cassandra device which increments automatically. In CQL, the query is:

```
Q1(startTime, endTime, threshold)
SELECT hashtag, counter
FROM Tweets[$startTime-$endTime]
HAVING COUNT(hashtag) > $threshold
```

Query Two identifies popular retweeted tweets. Here the variable threshold provides the minimum number of times a tweet must have been retweeted. If the retweet count exceeds the threshold then the tweet is considered a popular retweet and a result returned to the user.

```
Q2(startTime, endTime, threshold)
SELECT  counter, tweetid
FROM Tweets[$startTime-$endTime]
HAVING COUNT(retweet) > $threshold
```

Query Three finds all tweets with a hashtag and location specified by the user in a time-window.

```
Q3(startTime, endTime, location, hashtag)
SELECT tweet
FROM Tweets[$startTime-$endTime]
WHERE tweet.location=$location
AND tweet.hashtag=$hashtag
```

Query Four returns a count of how many times the specified hashtag has appeared in tweets within the time bounds.

```
Q4(startTime, endTime, hashtag)
SELECT COUNT(tweetID)
FROM Tweets[$startTime-$endTime]
WHERE tweet.hashtag = $hashtag
```

Query Five allows a user to specify a hashtag of interest to them; the query returns the tweets containing the hashtag within the time-bound.

```
Q5(startTime, endTime,hashtag)
SELECT tweet
FROM Tweets[$startTime-$endTime]
WHERE tweet.hashtag=$hashtag
```

Query Six identifies the spread of a tweet by identifying users that have posted the tweet.

```
Q6(tweetID)
SELECT userID as userid
FROM Tweets
WHERE $tweetID=tweet.id

SELECT noOfFollowers
FROM Tweets
WHERE userID = userid
```

Query Seven returns all the times that a tweet has been posted or re-tweeted, in a time-bound

```
Q7(tweetId, starttime, endtime)
SELECT time
FROM Tweets[$startTime-$endTime]
```

```
WHERE $tweetID=tweet.id
```

	Questions	Queries
Marketing and Adevertising	Question 1	Query One
	Question 2	Query Four
	Question 3	Query Two
Conference	Question 1	Query Five
	Question 2	Query Five
	Question 3	Query One
Emergency Response	Question 1	Query Five
	Question 2	Query Three
	Question 3	Query Seven

Table II: Questions and matching queries

V. Architecture

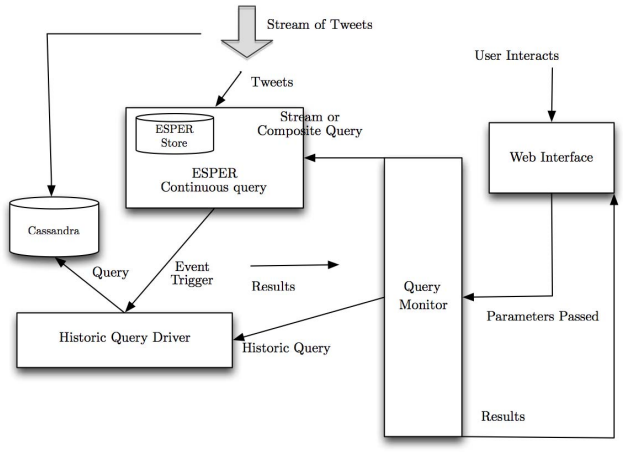

Figure 2: System architecture diagram

A system was designed and implemented to support the above queries, against streaming and historic data. Its goal was high throughput and low latency. This must be achieved while the system ingests the full Twitter firehose. The system architecture is shown in Figure 2 and is now described. Tweets from the firehose are collected using the Twitter4j streaming API [24]. Each is stored in a historic database. Queries can be sent to the system through a web portal designed to make it easy for users to create one of the generic queries described above: they simply select the query and fill in the parameters on a form. When data from the web interface arrives at the query monitor, it is analysed and turned into queries against the streaming data and/or the historic data depending on the time-bounds. The Complex Event Processing system ESPER is used to process queries against the streaming data. ESPER was chosen as it can handle large numbers of events, and specify time windows proficiently. It was decided to explore the use of a noSQL database for historic tweet storage and querying. Cassandra was selected due to its scalability. In particular, it supports high write rates, which is important given the rate at which tweets from the firehose need to be stored. The API used for connecting to Cassandra is datastax, a Java library which provides methods to connect to and query the database. Cassandra is a column-oriented database, based on keyspaces which are the equivalent to a database in a RDMS. Each keyspace is divided into column families that are similar to RDBMS tables. Each column family is made up of rows, and within these rows are sets of columns. Each row in Cassandra can have a different number of columns; there are static and dynamic column families. Static column families use a static set of names, although the number of these used in each column can vary depending on the data ingested. Dynamic column families store arbitrary column names taken from the data. This can support storing the data in a more easily retrievable way; the column families below will use this technique. A schema was designed to provide efficient access to data for answering the queries. One important design decision was to denormalise data to reduce execution times and increase throughput by eliminating the cost of joins for the generic queries. The two key design challenges for creating a scalable system were:

1) how to efficiently access tweets by time.
2) how to efficiently perform aggregations

We now describe the database design and show how these challenges were met. The column family tweet is indexed by tweetid with the status being used as the column name. As has been shown some queries use tweetid for identifying tweets. This indexing technique allows the status to be retrieved with increased efficiency and performance. This also improves write performance as Cassandra can write without reading, this is because the data is written sequentially. The system takes the tweetid and writes to the next space on disk, therefore no read is required beforehand. This also mean no other keys need to be changed and consistency is not an issue.

```
CREATE TABLE tweet (
    tweetid text,
    status text,
    PRIMARY KEY (tweetid))
```

The column family user is indexed using the same pattern as tweet, but tweet information is replaced by information about the user.

```
CREATE TABLE user (
    userid bigint,
    handle text,
    PRIMARY KEY (userid))
```

The hashtag_time column family answers queries one and four. All tweets arrive from the firehose with metadata that includes their creation time. For efficient access to data from tweets created within particular time-bounds, the schema uses time buckets, each of which stores data created within a particular time range. A timestamp is extracted from the metadata of the tweet and split into two values: day and minute. Day is the primary key and minute the column name. Each column in the row represents a minute within a day (e.g. 0-1399). The value of the day is determined by dividing the timestamp by 86400000 (as this is the amount of milliseconds in a day

and timestamps are stored in milliseconds). The granularity of the time buckets was chosen by analysis of typical queries within the case studies. The schema also takes advantage of Cassandra's composite column keys: this column family uses minute and hashtag as a composite column key (the PRIMARY KEY keywords in the schema introduce the keys: the first value is the row key and the rest form the column name; when the column name consists of multiple columns, it is called a composite column key). To deal with the challenge of efficiently handling aggregations, a count is maintained of the number of times that a hashtag has occured within the period identified by the day and minute. When a new tweet arrives from the firehose the composite key is used to increment the appropriate counter. The historic versions of queries one and four exploit this schema. One performs a range scan over the row, using the minutes to identify the correct range, which selects the hashtags and their counts. Once these have been retrieved, the count for each different hashtag is totalled and returned to the user if it exceeds the threshold. Query Four instead uses the day and the whole composite key to identify the count of a specific hashtag in that time period. For stream data, when the query is executed in ESPER the hashtags and their count are stored in an in-memory hashmap. The hashes are the keys and the value is a counter which is incremented each time the hashtag is included in a tweet. As the queries are time-bounded, the space occupied in the hashmap by a record can be recovered once the query reaches the end of time, and so the memory requirements are limited.

```
CREATE TABLE hashtag_time (
    day bigint,
    minute bigint,
    hashtag text,
    value counter,
    PRIMARY KEY (day, minute, hashtag))
```

The column family tweet_time follows an identical pattern to hashtag_time, with hashtags replaced by tweetids. It is used by Query Two and Seven to access data based on tweetid. If the time range is in the future, as a tweet is received from the stream the query identifies the retweet count from the metadata in the tweet. This is different to the continuous query used for Query One, as no state is required to be kept in-memory. The retweet count is used by the query to idenitfy if the count exceeds the threshold. If not the tweet is discarded and another tweet examined. This continues for the whole time-bound and the user is notified each time a retweet count exceeds the given threshold.

```
CREATE TABLE tweet_time (
    day bigint,
    minute bigint,
    tweetid text,
    value counter,
    PRIMARY KEY (day, minute,tweetid))
```

Tweet_geo is based on the same time bucket structure as the previous column families, it is used to answer Query Three. The column name is a composite column key of timebucket, hashtag and location, allowing for an exact key match to efficiently retrieve the status of the tweet based on location and hashtag. If the time-bound is based in the future as tweets are received this query questions the metadata of the tweet. It has the same pattern as Query Two. The place and hashtag from the metadata are checked for a match.

```
CREATE TABLE tweet_geo (
    day bigint,
    minute bigint,
    location text,
    hashtag text,
    status text,
    PRIMARY KEY
    (day, minute, location, hashtag))
```

Hashtag_status is indexed by time, it allows Query Five to retrieve a status using the hashtag. It adds redundant data (status) instead of creating a relation between hashtag and status for more efficient querying.

```
CREATE TABLE hashtag_status (
    day bigint,
    minute bigint,
    hashtag text,
    tweetid text,
    status text,
    PRIMARY KEY
    (day, minute, hashtag))
```

The last two column families are simply indexed like tweet. This provides effcient indexing for Query Six.

```
CREATE TABLE tweet_user (
    tweetid text,
    userid bigint,
    PRIMARY KEY (tweetid,userid))


CREATE TABLE user_follower (
    userid bigint,
    follower_count bigint,
    PRIMARY KEY (userid,follower_count))
```

For scalability the database is partitioned over several cloud nodes using consistent hashing [25] of each column family's rowkeys; this ensures that columns are never divided across different machines. The set of nodes can be viewed as a ring divided into different ranges. Each hashed key is part of a range within the set of nodes. The ranges are defined by assigning a number of virtual nodes to each physical node (this is just a number and defines the range that the node covers in the cluster). For this cluster, the ranges are equal, as each of the machines have the same specification. The replication strategy of this cluster is "simple strategy" this means that each row is saved on the designated node, and also one node along

the ring, moving clockwise. This provides fault tolerance for the cluster. The cluster uses the load balancing policy, round robin, which evenly distributes queries across the cluster they are then either satisfied by the receiving node or sent to the correct one for execution.

VI. Evaluation

To evaluate the system a collection of tweets were compiled using the Twitter4j streaming API [24] and replayed to replicate the firehose. The experiments were executed in the Amazon cloud, using m1.large machines.

A. Scalably Writing Tweets to the Database

For performance and scalability, Cassandra must be distributed across multiple nodes in the cloud: as will be seen, we cannot assume that one node is sufficient to handle storing the full Twitter firehose while executing all user queries. To scale the write capability of the system to the database two methods were used: increasing the size of the cluster and modifying the strategy for writing to the database. For a write in Cassandra to be satisfied the data must have been written to the commit log (for durability) and the in-memory structure called memtables. Using an asynchronous library tweets were batched before being inserted into the database. The system also uses a connection pool to allow more connections to the database. Both of these techniques made sure the system was configured for high write throughput, by ensuring the database was not idle waiting for data. Cassandra has a very high write rate so by ensuring the database is saturated with enough writes performance can be improved.

Figure 3 shows that by tuning and adding more nodes to the cluster the write throughput increases linearly. The optimal number of nodes to use is four as it can store up to 20,000 tweets per second in the database, which is four times greater than the average rate of tweets in the Twitter firehose. Therefore the number of nodes we will use for the proceeding experiments will be four as it has the capability of capturing the entire firehose and potential bursts.

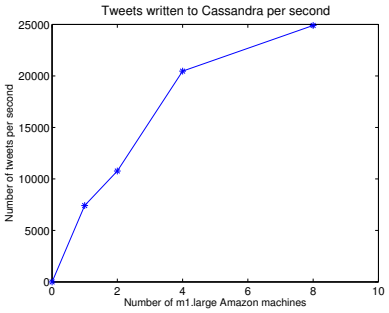

Figure 3: Tweets per second written to the database from the Twitter firehose

B. Historic Querying Performance

We first investigated queries on the database. The aim is for the system to scale to handle many simultaneous queries from users or applications. A range of different queries have been used to explore the performance. To test the scalability, the experiment uses a jmeter plug-in to create threads that generate queries. Each thread submits ten queries. The two types of queries that will be explored are One and Four, this is because they are examples of the two different types of indexing. One is an exact match query and Four a range query. Query Four uses the composite column key (timebucket and hashtag). The time-bound parameters were randomly generated values, ensuring fifty percent were in the earlier half of the day, fifty in the later half. These were used to demonstrate the capability of the system, by testing its scalability and performance under stress. By using values that are randomly ordered, and placed at the start (earlier in the day) and end (later in the day) of the row, reads to the database are not sequential. This was to simulate real world values, as the case studies suggest, queries would be written based on events, which do not have to be chronological. A randomly generated set of days was used to represent queries that span more than one day, which requires multiple queries. Multiple queries are required as each day is a new row. The database only allows range queries on the row not across different rows, this was taken into consideration as the schema was designed. Cassandra stores the most recent data to the memtables, until the commitlog reaches capacity and flushes to SSTables (on disk). Therefore if the data is newer it will still be in-memory but if it is older it will have been written to disk. Therefore having a spread of dates measures the systems response time while reading from memory and disk. The last parameter that is given in Query Four is the hashtag, eighty percent were included in the data, twenty percent not. The database behaves differently when no return data is selected, and sometimes users may be returned empty results. This is because the database need only look once on disk and as there is no result it returns straight away. The query executes across the hashtag_time column family and returns the count of the hashtag specified. Figure 4 shows a graph of the average response time for Query Four when executed across the database. The results illustrate the systems ability to scale the number of queries linearly, providing efficient response times. As can be seen for up to 10,000 users the query response times are under 400 milliseconds. This performance allows historic data to be queried as if it were stream data.

The next experiment used Query One. This is a range query and parameters select all hashtags within a time-bound. The parameters for this query were generated with the same distribution as the previous one. This query differs from four as it uses the minutes to bind the query to a time range. The queries for this experiment are generated by the same mechanism as the previous. The average response time for Query One is displayed in figure 5, this shows 1000 queries executing with a response time of under 2000 milliseconds. The graph also displays the linear scaling of the reads as the number of queries increases. This is because the indexing reduces processing increasing the efficiency and performance. This query is slower than Four, as a range query increases disk

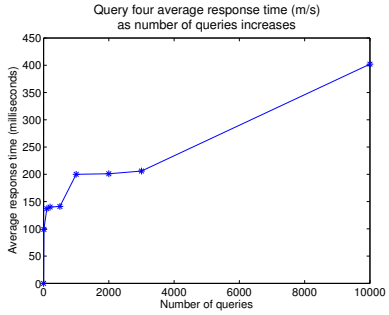

Figure 4: Average response time of Query Four across the database

seeks and returns more results.

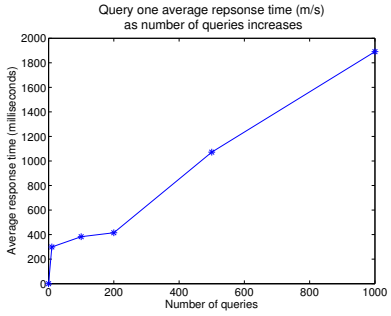

Figure 5: Average response time of Query One across the database

C. Stream Queries

Figure 6 shows both ESPER executing without a trigger to the historic store and one with. The first data set shows the throughput of tweets, for Query One, with no read is over 160,000 in 18 seconds (9000 per second). This demonstrates EPSER's ability to process the Twitter firehose quickly and efficiently. For each tweet the hashtag is identified and counted. These are saved and a count for each kept and checked until the threshold is exceeded.

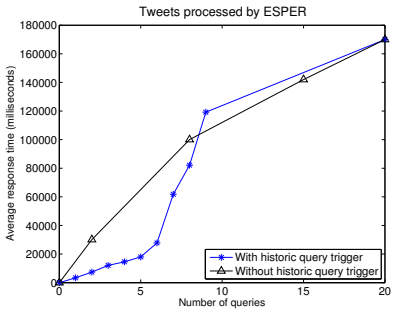

Figure 6: Event throughput in tweets per second

D. Querying Performance Results with Write Load

We have shown how the database performs with independent writes and reads. Next, this experiment identifies how query response time is affected by a simultaneous write load. The collected tweets are again used to simulate the Twitter firehose with a peak load. Each tweet is then written to the database. While the data is being written, different numbers of Query Four are executed across the database. Query Four was used as it had the most efficient execution and the parameters were kept the same as the previous experiment to simulate an identical read load. Figure 7 illustrates the same linear shape as the previous results as the amount of queries increases. The average response time is under 600 milliseconds for 2000 queries, eventually reaching 1.5 seconds for 5000. This illustrates the systems ability to process a heavy simultaneous read and write load.

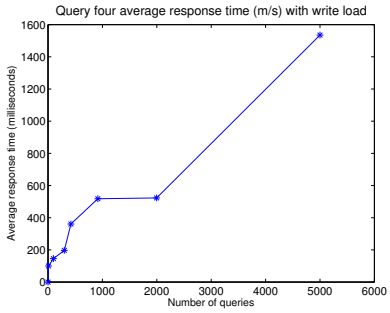

Figure 7: Read performance of Query Four with write load

E. Combined Query Performance

The system can also be used to combine stream and historic processing. Combining these techniques can be challenging, as historic querying must return with low latency to return up to date results that are consistent with the stream query being executed. It would be a huge drawback to receive the results after the stream query had returned and another was executing. The parameters generated for this query are a threshold, a time-bound specifying a time in the past and a time-bound in the future. The query monitor uses this to identify it is a composite query and sends the data to ESPER (to trigger a historic query if the threshold is exceeded). For this combined query, Query One and Five have been used. As the tweets are received from the stream ESPER keeps state about the hashtags and counts each time they have been mentioned, until they exceed the given threshold. If this occurs then Query Five is executed over the database, by passing the hashtag (the selected result from the last query) and the time-bound for the historic data. This returns whether the hashtag has been mentioned before in the time-bound and selects the contents of the tweet it was contained by. The time-bound is twenty seconds with a threshold of twenty hashtags. As previously shown, Query One executes efficiently in ESPER, but does combining it with a query to the historic store affect its

performance? Figure 7 demonstrates the linear ingestion rate of the system, which still has the capability of consuming the entire firehose. The second data set in Figure 7 shows the different average response times for the triggered query. The results are varied as the response times a little over 0 milliseconds are returning no result and the others return data. The triggered query causes a slight dip in performance, but this may also be accounted for by ESPER starting up. This is illustrated in the results, as by ten seconds there is an increase in the tweet throughput from the previous experiment (120,000 as apposed to 100,000 tweets). Figure 8 shows execution times of 800 milliseconds, while the database is also executing a write load. This demonstrates the systems ability to perform efficiently while combining querying techniques (returning data with immediate effect to be combined with the stream data) and writing data to the database. These low latency response times ensure the drawback mentioned previously never occurs and stream and historic results are combined efficiently.

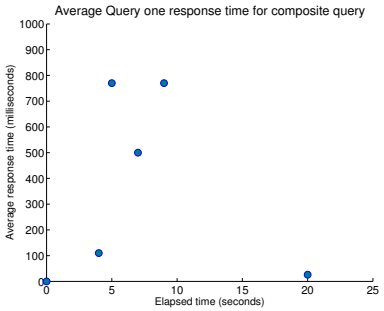

Figure 8: Query execution time over the database with the combined query

VII. Conclusion

In this paper we have presented a novel system. For scalable processing of Twitter data. It enables high performance querying with near real-time responses even while ingesting and storing the Twitter firehose at peak rate. This is enabled by the design of efficient indexing patterns which improve performance for aggregate functions and exact matches. This system was designed to provide users with the ability to analyse Twitter data, without being limited to a specific application domain. To achieve this we analysed a range of literature on questions asked of Twitter data to identify a core set of generic questions that drove the system design. The system was designed, implemented and deployed on the cloud, to provide scalability over mulitple nodes. Perfromance results were presented showing that the system performs well for a range of queries, and that four nodes is sufficient to handle the current peak arrival rate of tweets from the firehose. Future work will include integrating queries over the graph of users and their followers.

Acknowledgment

This work was supported by the Research Councils UK Digital Economy Programme [grant number EP/G066019/1 - SIDE: Social Inclusion through the Digital Economy]

References

[1] D. D'Orazio, "Twitter breaks 400 million tweet-per-day barrier, sees increasing mobile revenue The Verge," 2012. [Online]. Available: http://www.theverge.com/2012/6/6/3069424/twitter-400-million-total-daily-tweets

[2] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42–47, Dec. 2005. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1107499.1107504

[3] M. Grinev, "Analytics for the Real-Time Web," *System*, pp. 1391–1394.

[4] B. Meeder, B. Karrer, C. Borgs, R. Ravi, and J. Chayes, "We Know Who You Followed Last Summer : Inferring Social Link Creation Times In Twitter," *Focus*, 2012.

[5] M. Mathioudakis and N. Koudas, "TwitterMonitor : Trend Detection over the Twitter Stream," pp. 1155–1157, 2010.

[6] L. Hong and B. D. Davison, "Predicting Popular Messages in Twitter," *ReCALL*, pp. 57–58, 2011.

[7] A. P. Burnap, W. Housley, J. Morgan, L. Sloan, N. Avis, A. Edwards, O. Rana, M. Williams, and P. Burnap, "Working Paper 153 : Social Media Analysis , Twitter and the London Olympics (A Research Note) Social Media Analysis , Twitter and the London Olympics 2012," pp. 1–10.

[8] P. Burnap, O. F. Rana, N. Avis, M. Williams, W. Housley, A. Edwards, J. Morgan, and L. Sloan, "Detecting tension in online communities with computational Twitter analysis," *Technological Forecasting and Social Change*, May 2013. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0040162513000899

[9] E. Baucom and M. Chen, "Mirroring the Real World in Social Media : Twitter , Geolocation , and Sentiment Analysis Categories and Subject Descriptors," pp. 61–67.

[10] K. Weil, "Rainbird : Real-time Analytics @ Twitter," 2011.

[11] InfoQ, "twitter-summingbird." [Online]. Available: http://www.infoq.com/news/2014/01/twitter-summingbird

[12] Horvits, "analytics for big data-venturing with the twitter use case." [Online]. Available: http://horovits.wordpress.com/2012/01/27/analytics-for-big-data-venturing-with-the-twitter-use-case/

[13] R. McCreadie, C. Macdonald, I. Ounis, M. Osborne, and S. Petrovic, "Scalable distributed event detection for Twitter," *2013 IEEE International Conference on Big Data*, pp. 543–549, Oct. 2013. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6691620

[14] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter , a Social Network or a News Media ? Categories and Subject Descriptors," pp. 591–600, 2010.

[15] D. Fisher, "Interactions with Big Data Analytics population by running controlled," pp. 50–59, 1983.

[16] Twitter, "cs conferences." [Online]. Available: https://twitter.com/CS_Conferences

[17] D. S. Shiffman, "Twitter as a tool for conservation education and outreach: what scientific conferences can do to promote live-tweeting," *Journal of Environmental Studies and Sciences*, vol. 2, no. 3, pp. 257–262, Jul. 2012. [Online]. Available: http://www.springerlink.com/index/10.1007/s13412-012-0080-1

[18] T. L. Samuels, a. Papadopoulou, J. W. Willers, and D. R. Uncles, "Logging the delivery of intravenous lipid emulsion for cocaine and other lipophilic drug overdoses." *Anaesthesia*, vol. 67, no. 4, pp. 437–8, Apr. 2012. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/22409807

[19] A. B. P. Guide, "Social Media in an Emergency."

[20] N. T. murakami Akiko, "Tweeting about the Tsunami? -Mining Twitter for information on the Tohoko Earthquake and Tsunami," pp. 709–710, 2012.

[21] S. Nagar, A. Seth, and A. Joshi, "Characterization of social media response to natural disasters," *Proceedings of the 21st international conference companion on World Wide Web - WWW '12 Companion*, p. 671, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2187980.2188177

[22] P. J. R. Alan Rusbridger, "Reading the Riots," *The guardian*, 2012.

[23] A. Arasu, S. Babu, and J. Widom, "The CQL Continuous Query Language : Semantic Foundations and Query Execution ," *Science*, pp. 1–32.
[24] Twitter, "Twitter Steaming API." [Online]. Available: https://dev.twitter.com/docs/streaming-apis
[25] D. Kargerl, T. Leightonl, and D. Lewinl, "Consistent Hashing and Random Trees : Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web *," pp. 654–663.