

Processing and Visualizing the Data in Tweets

Adam Marcus, Michael S. Bernstein, Osama Badar,
David R. Karger, Samuel Madden, Robert C. Miller
MIT CSAIL

{marcua, msbernst, badar, karger, madden, rcm}@csail.mit.edu

ABSTRACT

Microblogs such as Twitter provide a valuable stream of diverse user-generated data. While the data extracted from Twitter is generally timely and accurate, the process by which developers extract structured data from the tweet stream is ad-hoc and requires reimplementa-tion of common data manipulation primitives. In this paper, we present two systems for querying and extract-ing structure from Twitter-embedded data. The first, TweepQL, provides a streaming SQL-like interface to the Twitter API, making common tweet processing tasks simpler. The second, TwitInfo, shows how end-users can interact with and understand aggregated data from the tweet stream, in addition to showcasing the power of the TweepQL language. Together these systems show the richness of content that can be extracted from Twitter.

1. INTRODUCTION

The Twitter messaging service is wildly popular, with millions of users posting more than 200 million *tweets* per day¹. This stream of messages from a variety of users contains information on an array of topics, in-cluding conventional news stories, events of local inter-est (e.g., local sports scores), opinions, real-time events (e.g., earthquakes), and many others.

Unfortunately, the Twitter interface does not make it easy to access this information. The majority of useful information is embedded in unstructured tweet text that is obfuscated by abbreviations (to overcome the 140-character text limit), social practices (e.g., prepending tweets from other users with *RT*), and references (e.g., URLs of full stories, or the *@usernames* of other users). Twitter’s APIs provide access to tweets from a particu-lar time range, from a particular user, with a particu-lar keyword, or from a particular geographic region, but provides no facility to extract structure from tweets, and does not provide aggregate views of tweets on different topics (e.g., the frequency of tweets about a particular topic over time.)

¹<http://blog.twitter.com/2011/06/200-million-tweets-per-day.html>

In this paper, we describe two approaches we have devised to help programmers and end-users make sense of the tweet stream. For programmers, we have built TweepQL, a SQL-like stream processor that provides streaming semantics and a collection of user-defined functions to extract and aggregate tweet-embedded data. For end-users, we built TwitInfo [7], a timeline-based visualization of events in the tweetstream, linked to raw tweet text, sentiment analysis, and maps.

2. TWEEQL

TweepQL provides a SQL-like query interface on top of the Twitter streaming API. The streaming API allows users to issue long-running HTTP requests with key-word, location, or userid filters, and receive tweets that appear on the stream and match these filters. TweepQL provides windowed select-project-join-aggregate queries over this stream, and utilizes user-defined functions for deeper processing of tweets and tweet text.

We begin by describing the TweepQL data model, and then illustrate its operation through a series of exam-ples. We close with a discussion of challenges with building TweepQL and future directions.

2.1 Data Model and Query Language

TweepQL is based on SQL’s select-project-join-aggregate syntax. Its data model is relational, with both traditional table semantics as well as streaming semantics.

2.1.1 Streams

The primary stream that TweepQL provides is *twitter_stream*. TweepQL users define new streams based on this base stream using the *CREATE STREAM* state-ment, which creates a named substream of the main twitter stream that satisfies a particular set of filters. For example, the following statement creates a querable stream of tweets containing the term *obama* generated from the *twitter_stream* streaming source:

```
CREATE STREAM obama_tweets
FROM twitter_stream
WHERE text contains 'obama';
```

While *twitter_stream* offers several fields (e.g., *text*, *username*, *userid*, *location*, *latitude*, *longitude*), the Twitter API only allows certain filters to be used as access methods for defining a stream. When defining a new stream on top of *twitter_stream*, the developer must provide a key lookup on *userid*, a text match on *text*, or a range lookup on *latitude* and *longitude*. If a user tries to create a stream from a streaming source but omits API-required filters, TweepQL will raise an error.

Users are not allowed to directly query the raw *twitter_stream* because Twitter only provides access to tweets that contain a filter. If users wish to access an unrestricted stream, Twitter provides a sampled, unfiltered stream that TweepQL wraps as *twitter_sample*. A complete, unfiltered stream is not provided by Twitter for performance and financial reasons.

While our examples show users creating streams from the *twitter_stream* base stream, in principle one could also wrap other streaming sources, such as RSS feeds, a Facebook news feed, or a Google+ feed. Once wrapped, derived streams can be generated using techniques similar to the examples we provide.

2.1.2 UDFs

TweepQL also supports user-defined functions (UDFs). UDFs in TweepQL are designed to provide operations over unstructured data such as text blobs. To support such diversity in inputs and outputs, TweepQL UDFs accept and return array- or table-valued attributes. TweepQL UDFs also help wrap web APIs for various services, such as geocoding services.

Complex Data Types. TweepQL UDFs can accept array- or table-valued attributes as arguments. This is required because APIs often allow a variable number of parameters. For example, a geocoding API might allow multiple text locations to be mapped to latitude/longitude pairs in a single web service request.

UDFs can also return several values at once. This behavior is needed both for batched APIs that submit multiple requests at once, and for many text-processing tasks that are important in unstructured text processing. For example, to build an index of words that appear in tweets, one can issue the following query:

```
SELECT tweetid, tokenize(text)
FROM obama_tweets;
```

The *tokenize* UDF returns an array of words that appear in the tweet text. For example, *tokenize*("Tweet number one") = ["Tweet", "number", "one"]. While arrays can be stored or passed to array-valued functions, users often wish to "relationalize" them. To maintain the relational model, we provide a *FLATTEN* operator (based on the operator of the same name from Olston et al.'s Pig Latin [8]). Users can wrap an array-valued

function found in a *SELECT* clause with a *FLATTEN* to produce a result without arrays. For example, instead of the above query, the programmer could write:

```
SELECT tweetid, FLATTEN(tokenize(text))
FROM obama_tweets;
```

The resulting tuples for a tweet with *tweetid* = 5 and *text* = "Tweet number one" would then be:

```
(5, 'Tweet')
(5, 'number')
(5, 'one')
```

Web Services as UDFs. Much of TweepQL's structure-extraction functionality is provided by third parties as web APIs. TweepQL allows UDF implementers to make calls to such web services to access their functionality. One such UDF is *geocode*, which returns the latitude and longitude for user-reported textual locations as described in Section 2.1.4. The benefit of wrapping such functionality in third party services is that often the functionality requires large datasets—good geocoding datasets can be upward of several gigabytes—that an implementer can not or does not wish to package with their UDF. Wrapping services comes at a cost, however, as service calls generally incur high latency, and service providers often limit how frequently a client can make requests to their service.

Because calls to other web services may be slow or rate-limited, a TweepQL UDF developer can specify several parameters in addition to the UDF implementation. For example, the developer can add a cache invalidation policy for cacheable UDF invocations, as well as any rate-limiting policies that the API they are wrapping allows. To ensure quality of service, the developer can also specify a timeout on wrapped APIs. When the timeout expires, the return token *TIMEOUT* is returned, which acts like a *NULL* value but can explicitly be fetched at a later time. Similarly, a *RATELIMIT* token can be returned for rate-limited UDFs.

2.1.3 Storing Data and Generating Streams

It is often useful for TweepQL developers to break their workflows into multiple steps and to write final results into a table. To support both of these operations, we allow the results of *SELECT* statements over streams to write data to named tables. In this way, intermediate steps can be named to allow subsequent queries in a workflow to utilize their results.

Output to a table and temporarily naming tuples is accomplished via the *INTO* operator. To save results, a programmer can add an *INTO TABLE tablename* clause to their query. To name a set of results that can be loaded as a stream by another query, the programmer can add an *INTO STREAM streamname* clause to their query. For example, consider the following three queries:

```

CREATE STREAM sampled
FROM twitter_sample;

SELECT text, sentiment(text) AS sent
FROM sampled
INTO STREAM text_sentiment;

SELECT text
FROM text_sentiment
WHERE sent > 0
INTO TABLE positive_sentiment;

SELECT text, sent
FROM text_sentiment
WHERE text contains 'obama'
INTO TABLE obama_sentiment;

```

The first query creates an unfiltered sampled stream called *sampled*. The second query retrieves all tweet text and its sentiment (described in Section 2.1.4), and places that text in a stream called *text_sentiment*. The third query stores all positive-sentiment tweet text from the *text_sentiment* stream in a table called *positive_sentiment*. The final query stores all tweet text from the *text_sentiment* stream containing the term *obama* in a table called *obama_sentiment*.

For testing purposes, it is also possible to select a stream *INTO STDOUT*, which outputs the contents of a stream to a user’s console.

2.1.4 Structure Extraction UDFs

One key feature of our TweepQL implementation is that it provides a library of useful UDFs. One important class of operators are those that allow programmers to extract structure from unstructured content. The functions are described below.

String Processing. String functions help extract structure from text. We have already described one such UDF, *tokenize* in Section 2.1.2 that splits strings into a list of tokens. Other UDFs allow more complex string extraction, such as regular expressions that return lists of matches for each string.

Location. Tweets are annotated with location information in several ways. GPS-provided coordinates are most accurate, but only a small fraction of tweets are annotated with such precision (0.77% in mid-2010 [6]). More common is a self-reported location field, with values ranging from the nonsensical “Justin Bieber’s heart” [6] to a potentially accurate “Boston, MA.”

To extract structure from self-reported location strings, we offer a *geocode* UDF. The following query extracts the sentiment of tweets containing the term *obama* as well as the coordinates of the self-reported location:

```

SELECT sentiment(text) AS sent,
       geocode(loc).latitude AS lat,
       geocode(loc).longitude AS long
FROM obama_tweets
INTO STREAM obama_sent_loc;

```

The query also displays another feature of TweepQL UDFs. In addition to being able to return lists of fields to be flattened into a resultset, UDFs can return tuples rather than fields. In the example above, *geocode* returns a tuple of coordinates that the query projects into two fields, *lat* and *long*, in the result set.

Classification. Classifiers can be used to identify structure in unstructured text content. For example, social science researchers explore various ways to use the tweet stream as a proxy for public sentiment about various topics. TweepQL provides a *sentiment* UDF for classifying tweet text as expressing positive or negative sentiment. An example of this UDF can be seen in the *obama_sent_loc* stream example above. Other classifiers might identify the topic, language, or veracity of a tweet.

Named entity extraction. So far, we have identified tweets about President Obama by filtering tweets whose text contains the term *obama*. This approach may be unacceptable when two people with the same name might be confused. For example, searching for tweets containing the term *clinton* combines tweets such as “Secretary Clinton accepts Crowley resignation” and ones such as “Former President Clinton undergoes heart surgery.”

To reduce ambiguity, TweepQL provides a *namedEntities* UDF that identifies potential entities in context. For example, *namedEntities*(“Secretary Clinton accepts Crowley resignation”) returns a list of fields [*Hillary Clinton*”, *P.J. Crowley*”] that can be filtered.

With the *namedEntities* UDF, we can refine our original *obama_tweets* example to identify tweets specifically involving Barack Obama.

```

CREATE STREAM obama_tweets
FROM twitter_stream
WHERE text contains 'obama';

SELECT text,
       FLATTEN(namedEntities(text)) AS entity
FROM obama_tweets
INTO STREAM obama_entities;

SELECT text
FROM obama_entities
WHERE entity = "Barack Obama"
INTO STREAM barack_obama_tweets;

```

The current implementation of *namedEntities* is an API wrapper around OpenCalais ², a web service for

²<http://www.opencalais.com/>

performing named entity extraction and topic identification. OpenCalais was designed to handle longer text blobs (e.g., a newspaper article) for better contextual named entity extraction. One area of future work is to develop named entity extractors for tweets, which are significantly shorter.

2.1.5 Windowed Operators

Like other stream processing engines, TweepQL supports aggregates and joins on streams. Because streams are infinite, we attach sliding window semantics to them, as in other streaming systems [2, 1]. Windows are defined by a *WINDOW* parameter specifying the time-frame during which to calculate an aggregate or join.

Any streaming source must include a *__created_at* timestamp field. By default, tuples are timestamped with their creation time. On aggregates, an *EVERY* parameter specifies how frequently to emit *WINDOW*-sized aggregates. If an *EVERY* parameter is smaller than the *WINDOW* parameter, overlapping windows are emitted. The *__created_at* field of a tuple emitted from an aggregate is the time that the window begins.

The query below provides an example of the *WINDOW* and *EVERY* parameters for aggregates:

```
SELECT AVG(sent) AS sent,
       floor(lat) AS lat,
       floor(long) AS long
FROM obama_sent_loc
GROUP BY lat, long
WINDOW 3 hours
EVERY 1 hour
INTO STREAM obama_sent_by_area;
```

The query converts the *obama_sent_loc* stream of sentiment, latitude, and longitude into an average sentiment expressed in a 1°x 1° area. This average is computed over the course of three hours, and is emitted every hour.

2.1.6 Event Detection

As we explore with *TwitInfo* in Section 3, the number of tweets per minute mentioning a topic is a good signal of peaking interest in the topic. If the number of tweets per minute is significantly higher than recent history, it might suggest that an event of interest has just occurred.

To support event detection, we provide a *meanDeviation* UDF. The UDF takes a floating-point value as an argument. It returns the difference between the value and an exponentially weighted moving mean (EWMA) of recent values. This difference is called the mean deviation. Before returning the EWMA, it updates the EWMA with the floating-point value for future calls. The details of this algorithm are spelled out in [7]. The following example illustrates its use:

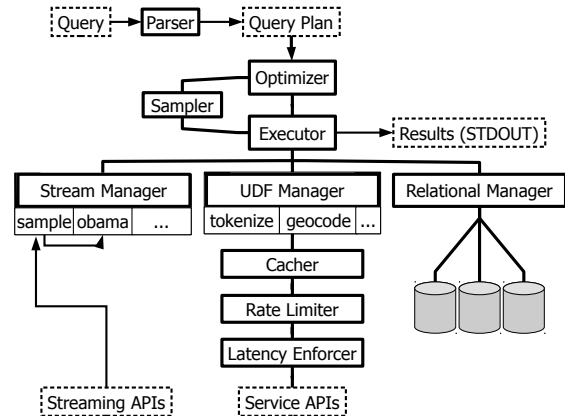


Figure 1: TweepQL architectural components.

```
SELECT COUNT(text) as count,
       __created_at as time
FROM obama_tweets
WINDOW 1 minute
EVERY 1 minute
INTO STREAM obama_counts;

SELECT meanDeviation(count) AS dev,
       time
FROM obama_counts
WHERE dev > 2
INTO TABLE obama_peaks;
```

The first query uses windowed aggregates, described in Section 2.1.5, to calculate the tweets per minute mentioning the term *obama*. The second query calculates the mean deviation of each tweets-per-minute value, and stores the time of deviations above 2 in a table *obama_peaks*.

The *meanDeviation* UDF is unique in that it stores state that is updated between calls. This makes the semantics of the UDF difficult to define, as calling *meanDeviation(count)* on the same *count* value with different histories will result in a different return value. In practice we found that the simple interface to the *meanDeviation* UDF makes it usable for event detection.

2.2 System Design

Figure 1 illustrates the key architectural components of the TweepQL stream processor.

TweepQL offers its SQL-like query language through a traditional query prompt or in batched query mode. All of the queries that make up a workflow (e.g., *sampled*, *text_sentiment*, *positive_sentiment*, and *obama_sentiment* in Section 2.1.3) are handled together and sent to the **Query Parser** to be processed at the same time.

The parser generates batches of dependent query trees, some of which store records in tables while others generate streams that other query trees depend on. The

Optimizer reorders operators as informed by selectivity and latency statistics collected by the **Sampler**. In addition to reordering operators, the optimizer also decides which filters to send to streaming APIs such as Twitter's to reduce the number of tuples returned. The sampler keeps statistics on all APIs and tables known to the database.

Optimized query tree batches are sent to the **Executor**. The query executor is iterator-based. As streaming sources asynchronously generate tuples, the streams are buffered by streaming access method operators that allow iterator access. Streamed tuples appear as tuples with a fixed schema to the rest of the query tree. As we see in Section 2.1.3, a stream (such as *text_sentiment*) can be used by multiple downstream query trees. Downstream query trees register themselves as listeners to named streams that send batches of tuples generated at their root to the streaming buffer of each query tree.

There are three data source managers from which the executor retrieves data: a stream manager, a UDF manager, and a relational manager.

The **Stream Manager** manages all streams generated with *CREATE STREAM* or *INTO STREAM*. It communicates with streaming APIs such as Twitter's, and informs streaming access method operators in query trees when new batches of tuples arrive from streaming sources.

The **UDF Manager** manages all UDF invocations. While traditional UDFs are executed as they are in traditional RDBMSs, the UDF Manager has special logic for handling UDFs which wrap web services. In addition to providing adapters that generate relational data from nonrelational services, it contains components that apply to all requests. The **Cacher** ensures that frequent service requests are cached, and supports age- and frequency-based cache eviction policies. The **Rate Limiter** enforces service-based rate limiting policies. These policies generally limit the number of requests per minute, hour, or day. Finally, the **Latency Enforcer** ensures that requests that run for too long are returned with *TIMEOUT* as discussed in Section 2.1.2. The latency enforcer still allows requests returned after a timeout to be cached for future performance benefits.

The **Relational Manager** simply wraps traditional relational data sources for querying, and stores tables generated with *INTO TABLE* syntax.

2.3 Current Status

TweeQL is implemented in Python, using about 2500 lines of code. The implementation is available as an open source distribution³. The distribution includes most of the features described in this paper. We are working to add the rate-limiting and latency-enforcing logic to web service UDF wrappers. The *CREATE*

³<https://github.com/marcua/tweeql>

STREAM and *INTO STREAM* statements, which we realized were necessary as we wrapped streams for services other than Twitter, are available in experimental versions of TweeQL. Finally, we intend to add *FLATTEN* syntax in the next TweeQL release.

2.4 Challenges

In this section, we describe a number of challenges and open issues we encountered when building TweeQL.

Uncertain Selectivities. When creating a stream, TweeQL users may issue multiple filters that can be passed to the streaming API. Only one filter type can be submitted to the API, and selecting the most efficient one to send is difficult. For example, consider a user issuing the query:

```
CREATE STREAM obama_nyc
FROM twitter_stream
WHERE text contains 'obama';
AND location in [bounding box for NYC];
```

The user wants to see all tweets containing the word *obama* that are tweeted from the New York City area. TweeQL must select between requesting all *obama* tweets, or all *NYC* tweets.

We benefit from having access to Twitter's historical API in this case. We can issue two requests for recent tweets with both filters applied, and determine which stream is less frequent. We are also exploring Eddies-style [3] dynamic operator reordering to adjust to changes in operator selectivity over time.

High-latency Operators. As discussed in Section 2.1.2, TweeQL UDFs can return *TIMEOUT* and *RATELIMIT* for long-running or rate-limited web services. Still, the high latency of operations is in tension with the traditional blocking iterator model of query execution.

Web service API requests such as geolocation can take hundreds of milliseconds apiece, but incur little processing cost on the query executor. Though the operations incur little computational cost, they often bottleneck blocking iterators. Caching responses and batching multiple requests when an API allows can reduce some request overhead.

We are exploring modifying iterators to operate asynchronously as described by Goldman and Widom [5]. This, in combination with a data model that allows partial results as described by Raman and Hellerstein [9], might be a sufficient solution.

Aggregate Classifiers are Misleading. In the development of *TwitInfo*, described in Section 3, we ran into an issue with running aggregates over the output of classifiers such as the *sentiment* UDF. We describe the problem and one solution in detail in [7].

One such example can be seen in the *obama_sent_by_area* stream in Section 2.1.5. Consider the case where the

sentiment UDF simply outputs 1 for positive text and -1 for negative text. It is possible that the classifier powering *sentiment* has different recall (e.g., the fraction of text identified as positive in situations where the text is actually positive) for positive and negative classifications. In this case, $AVG(sent)$ will be biased toward the class with higher recall. The solution described in [7] is to adjust for this bias by learning the positive and negative recall values ($recall_{positive}$ and $recall_{negative}$) on training data. With these values, we can return $\frac{1}{recall_{positive}}$ for positive text, and $\frac{-1}{recall_{negative}}$ for negative text. These values, when summed or averaged, adjust for overall recall differences.

3. TWITINFO

TwitInfo [7] is an application written on top of the TweepQL stream processor. TwitInfo is a user interface that summarizes events and people in the news by following what Twitter users say about those topics over time⁴. TwitInfo offers an example of how aggregate data extracted from tweets can be used in a user interface. Other systems, such as Vox Civitas [4], allow similar exploration, but TwitInfo focuses on the streaming nature of tweet data and uses event detection to relay a story.

3.1 Creating an Event

TwitInfo users define an *event* by specifying a Twitter keyword query. For example, for a soccer game, users might enter search keywords *soccer*, *football*, *premier-league*, and team names like *manchester* and *liverpool*. Users give the event a human-readable name like “Soccer: Manchester City vs. Liverpool” as well as an optional time window. When users are done entering the information, TwitInfo saves the event and begins logging tweets containing the keywords using a TweepQL query like the following:

```
CREATE STREAM twitinfo
FROM twitter_stream
WHERE text contains `soccer`
      OR text contains `football`
      OR text contains `premierleague`
      OR text contains `manchester`
      OR text contains `liverpool`;
```

This query results in some irrelevant tweets (e.g., tweets about American Football). In [7], we discuss how to remove noisy terms and rank tweets by relevance.

3.2 Timeline and Tweets

Once a user has created an event, TwitInfo creates a page on which the user can monitor the event. The TwitInfo interface (Figure 2) is a dashboard summarizing

⁴The TwitInfo website with interactive visualizations is accessible at <http://twitinfo.csail.mit.edu/>

the event over time. The dashboard displays a timeline for the event, raw tweet text sampled from the event, an overview graph of tweet sentiment, and a map view displaying tweet sentiment and locations.

The event timeline (Figure 2.2) reports tweet activity by volume. The more tweets that match the query during a period of time, the higher the y-axis value on the timeline for that period. When many users are tweeting about a topic (e.g., a goal by Manchester City), the timeline spikes. TwitInfo’s peak detection algorithm is implemented in a stateful TweepQL UDF described in Section 2.1.6. The algorithm identifies these spikes and flags them as peaks in the interface.

Peaks appear as flags in the timeline. TwitInfo automatically generates key terms that frequently appeared in tweets during a peak, and displays them to the right of the timeline. For example, in Figure 2.2, TwitInfo automatically tags one of the goals in the soccer game as peak “F” and annotates it on the right with representative terms in the tweets like ‘3-0’ (the new score) and ‘Tevez’ (the soccer player who scored). Users can perform text search on this list of key terms to locate a specific peak.

As users click on peaks, the map, tweet list, links, and sentiment graph update to reflect tweets in the period covered by the peak.

The Relevant Tweets panel (Figure 2.4) contains the tweets that have the highest overlap with the event peak keywords. These tweets expand on the reason for the peak. The relevant tweets are color-coded red, blue, or white depending on whether the sentiment they display is negative, positive, or neutral.

3.3 Aggregate Metadata Views

A user may wish to see the general sentiment on Twitter about a given topic. The Overall Sentiment panel (Figure 2.6) displays a pie chart with the proportion of positive and negative tweets during an event.

Twitter users share links as a story unfolds. The Popular Links panel (Figure 2.5) aggregates the top URLs extracted from tweets in the timeframe being explored.

Often, opinion on an event differs by geographic region. The Tweet Map (Figure 2.3) displays tweets that provide geolocation metadata. The marker for each tweet is colored according to its sentiment, and clicking on a pin reveals the associated tweet.

3.4 Uses and Study

As we developed TwitInfo, we tested its ability to identify meaningful events and its effectiveness at relaying extracted information to users.

We have tracked events of different duration and content using TwitInfo. In soccer matches, TwitInfo successfully identifies goals, half-time, the end of a game, and some penalties. The system successfully identified

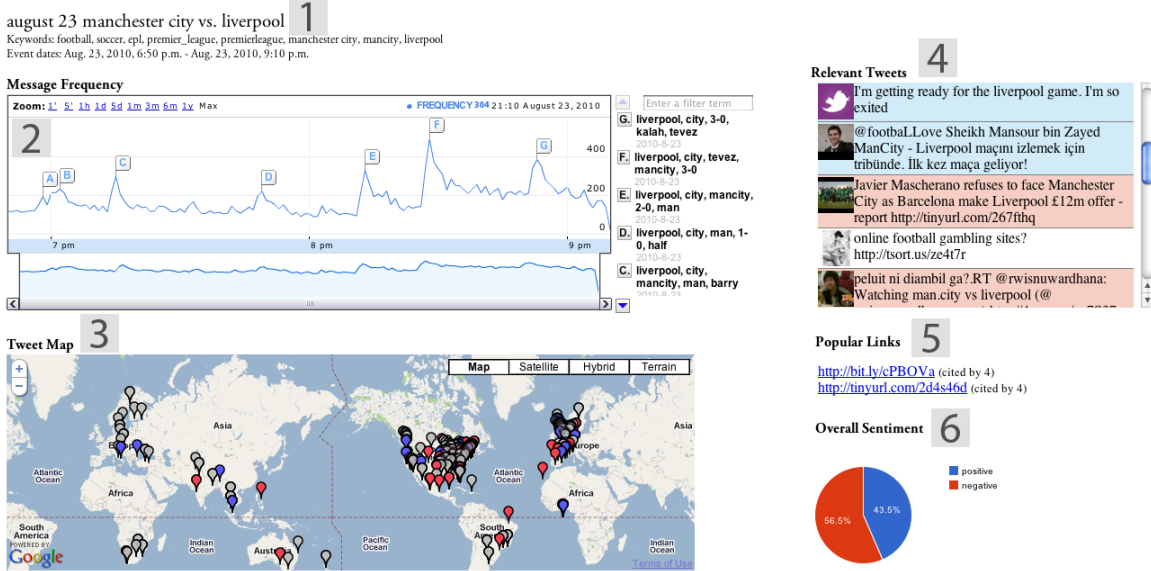


Figure 2: The TwitInfo user interface summarizing a soccer game.

all major earthquakes over a 1-month timespan. Finally, we visualized sixteen days in Barack Obama’s life and policymaking, and identified most newsmaking events in the interface. Examples of these visualizations can be found on the TwitInfo website.

We tested the TwitInfo interface with twelve users. We asked them to reconstruct either a soccer game or sixteen days in Barack Obama’s life based solely on the TwitInfo user interface. Participants found the interface useful for such summaries, with one participant recounting in detail Obama’s every activity over the timespan without having read any other news on the topic [7].

While users explained that TwitInfo provides them with a good summary of an event, they often described the summary as shallow. This is in part due to the short, fact-oriented nature of tweets.

A Pulitzer Prize-winning former Washington Post investigative reporter thought of two use-cases for TwitInfo in journalism. The first was in *backgrounding*: when a journalist starts researching a topic, it helps to have an overview of recent events of note. The second use was in finding eyewitnesses. While reporters are generally averse to trusting tweets at face value, a location-based view of tweets can help identify Twitter users that may have been at or near an event to follow up with in more detail.

4. CONCLUSION

Twitter offers a diverse source of timely facts and opinions. In order for the information in unstructured tweets to be useful, however, it must be tamed. We described two tools, TweepQL for programmers and Twit-

Info for end-users, to make this information more accessible. More broadly, social streams offer the database community an opportunity to build systems for streaming, unstructured data, and social networks in the wild.

5. ACKNOWLEDGEMENTS

We thank Eugene Wu, who discussed the mechanics of stream creation and wrapping syntax.

6. REFERENCES

- [1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 2003.
- [2] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. Technical Report 2003-67, Stanford InfoLab, 2003.
- [3] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *In SIGMOD 2000*.
- [4] N. Diakopoulos, M. Naaman, and F. Kivran-Swaine. Diamonds in the rough: Social media visual analytics for journalistic inquiry. In *VAST*, 2010.
- [5] R. Goldman and J. Widom. WSQ/DSQ: a practical approach for combined querying of databases and the web. *SIGMOD Record*, 2000.
- [6] B. Hecht, L. Hong, B. Suh, and E. H. Chi. Tweets from justin beiber’s heart: the dynamics of the location field in user profiles. *CHI*, 2011.
- [7] A. Marcus, M. S. Bernstein, O. Badar, D. R. Karger, S. Madden, and R. C. Miller. Twitinfo: aggregating and visualizing microblogs for event exploration. *CHI*, 2011.
- [8] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. *SIGMOD*, 2008.
- [9] V. Raman and J. M. Hellerstein. Partial results for online query processing. In *SIGMOD*, 2002.