SocialTrove: A Self-summarizing Storage Service for Social Sensing

Md Tanvir Al Amin*, Shen Li*, Muntasir Raihan Rahman*, Panindra Tumkur Seetharamu*, Shiguang Wang*, Tarek Abdelzaher*, Indranil Gupta*, Mudhakar Srivatsa†, Raghu Ganti†, Reaz Ahmed‡, and Hieu Le§
*University of Illinois at Urbana-Champaign, Urbana, Illinois 61801
Email: {maamin2, shenli3, mrahman2, tumkurs2, swang83, zaher, indy}@illinois.edu
†IBM Research, Yorktown Heights, NY 10598, Email: {msrivats, rganti}@us.ibm.com
‡University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, Email: r5ahmed@uwaterloo.ca
§Facebook Inc., Menlo Park, CA 94025, Email: hieu@fb.com

***Abstract*—The increasing availability of smartphones, cameras, and wearables with instant data sharing capabilities, and the exploitation of social networks for information broadcast, heralds a future of real-time information overload. With the growing excess of worldwide streaming data, such as images, geotags, text annotations, and sensory measurements, an increasingly common service will become one of data summarization. The objective of such a service will be to obtain a *representative sampling* of large data streams at a configurable granularity, in real-time, for subsequent consumption by a range of data-centric applications. This paper describes a general-purpose self-summarizing storage service, called SocialTrove, for social sensing applications. The service summarizes data streams from human sources, or sensors in their possession, by hierarchically clustering received information in accordance with an application-specific distance metric. It then serves a sampling of produced clusters at a configurable granularity in response to application queries. While SocialTrove is a general service, we illustrate its functionality and evaluate it in the specific context of workloads collected from Twitter. Results show that SocialTrove supports a high query throughput, while maintaining a low access latency to the produced real-time application-specific data summaries. As a specific application case-study, we implement a fact-finding service on top of SocialTrove.**

I. Introduction

This paper describes the design, implementation, and evaluation of SocialTrove; a self-summarizing storage service for social sensing applications. The service offers an API that allows applications to access their data at different degrees of summarization in a configurable manner. SocialTrove is motivated by the advent of an age of data overload, brought about by the increasing availability of smart devices with instant data collection and sharing capabilities, as well as by the growth of social network broadcast, such as microblog upload on Twitter. Early autonomic computing envisioned machines with self-* properties that independently meet application needs. The rise of social networks in the present decade, together with the proliferation of smart devices and other digital data sources, suggests that an increasing application need in the foreseeable future will be one of summarizing large volumes of redundant data for subsequent processing. This motivates development of a *general-purpose summarization service*.

In this paper, we focus on *social sensing* applications. We refer by social sensing to those applications, where humans share information on themselves or their environment, either directly (e.g., by blogging) or using sensing devices in their possession (e.g., sensing on a smart phone). The application features a back-end, where collected data is stored, which is the focus of our work. Social sensing applications encompass participatory sensing [1]–[4], opportunistic sensing [5]–[7], and use of humans as sensors [8]–[13]. For example, smartphone users on a participatory sensing campaign might run a geotagging application that allows them to upload GPS locations of items of interest via the phone. The application might also allow them to describe these items using text tags, or to supply images. For another example, Internet-connected vehicles may upload speed information periodically from on-board navigation systems, allowing the back-end servers to compute city traffic speed of different streets. In recent work, the authors explored the use of social networks, such as Twitter, as sensor networks, observing that many tweets can be viewed as bits of information about the state of the physical world. For such sensor networks, an application might construct physical state estimates from "human sensor" observations [8], [10], [11], [14]. A common characteristic of social sensing systems exemplified above is that they generate large amounts of redundant data. The underlying data objects may be different, depending on the application. The simplest way to summarize data is to reduce redundancy by offering a *sampling* of the original data set, where the selected samples are minimally redundant. We call such a sampling policy, *representative sampling*. A challenge, therefore, is to develop a representative sampling service that is agnostic to the data type.

SocialTrove is an exercise in building a *general-purpose* representative sampling service that reduces redundancy in large data sets. The service allows application designers to specify an *application-specific* distance metric that describes a measure of similarity relevant to this application among data items. Based on that application-specific measure, the service hierarchically clusters incoming data streams in real time, and allows applications to obtain representative samples at arbitrary levels of granularity by returning cluster heads (and member counts) at appropriate levels of the cluster hierarchy.

An important design consideration in developing our service is scalability. When data are large, if the observations are stored in a cluster-agnostic manner, retrieving a representative summary would require scanning the entire set of observations, thereby communicating with many machines and

decreasing throughput. Instead, SocialTrove stores content in a *similarity-aware* fashion, according to the application-specific similarity metric. We implement SocialTrove and evaluate its performance in the context of summarizing Twitter data. We demonstrate that it outperforms the alternate mechanisms in terms of both (summary) query latency, and maximum query throughput. To demonstrate an application that uses Twitter data summaries, we built a fact-finding service [8] that uses the produced summaries to determine which observations are more credible in the presence of noise, errors, and conflicts. We observe that the fact-finder implementation on top of SocialTrove required significantly fewer lines of code than a standalone service.

The rest of this paper is organized as follows. Section II describes the main interface exported by SocialTrove as a self-summarizing storage service. In Section III we present the distributed architecture of the SocialTrove runtime. Section IV presents microbenchmarks and a performance evaluation. We review the related work in Section V. This paper concludes with a discussion in Section VI.

II. A Self-Summarizing Storage Model

Our goal in this paper is to build a (data storage) service that allows an application to retrieve summaries of their data at arbitrary levels of granularity based on an application-specific redundancy metric. We call such a service, *self-summarizing* storage. The main purpose of summarization is to reduce data redundancy by selecting data samples that are minimally redundant. Towards that end, SocialTrove employs a hierarchical clustering scheme and returns data samples constituting cluster-heads at a configurable granularity (together with the sizes of corresponding clusters). Finally, we aim to design the service that is agnostic to the data type, so that it may be reused in different application contexts. Hence, we allow applications to define their own application-specific distance metric between data objects, and cluster objects in the corresponding feature space. The SocialTrove API is carefully designed not to make assumptions regarding the feature space in which application objects live, and yet perform clustering, store clusters, and serve summary queries in an efficient manner at different levels of granularity.

In accordance with the above design requirements, the fundamental abstraction and main "citizen" of SocialTrove is the abstract *data object*. It is an opaque data type that SocialTrove itself does not interpret. Instead, it stores object records that are tuples of (`ObjectSource`, `ObjectHandle`, `FeatureVector`), where `ObjectSource` specifies the ID of the input source (e.g., sensor ID, camera ID, or social network user ID) from which the object was obtained, `ObjectHandle` is a handle to the abstract data type, and `FeatureVector` is a placeholder for the object's application-specific feature vector (not computed by SocialTrove).

Further, the service offers two interfaces; (i) a *customization interface* that allows applications to define their application-specific features and distance metrics for objects, and (ii) a *summary query interface*, that allows applications to retrieve data summaries at different degrees of granularity. We begin the paper by describing those interfaces first to give the reader, respectively, an understanding of (i) the way we attain independence of the service from the application-specific data type, and (ii) the functionality we offer to the application.

A. The Customization Interface

To customize SocialTrove to the summarization needs of a particular application, two application-specific callback functions must be written by the application developer. These functions will be called by SocialTrove. Namely:

- `Vectorize(u)`: SocialTrove expects applications to implement a callback function, called `Vectorize()`. SocialTrove passes an object handle, `u`, to this function. The function returns a corresponding feature vector, `FeatureVector`. Note that, SocialTrove never interprets the incoming objects themselves or assumes their format. Rather, only `Vectorize()` is aware of what an object means. Similarly, SocialTrove does not interpret the output feature vector. It is stored as an opaque data type in the object's record.
- `Distance(u.FeatureVector, v.FeatureVector)`: SocialTrove expects applications to implement a callback function, called `Distance()`, that computes the distance between two objects, `u` and `v`, based on their feature vectors. As mentioned above, SocialTrove itself never interprets the feature vectors, as they are application specific. Instead, it treats the feature vectors generated by the `Vectorize` function as an opaque data type. A handle to the data type is stored in the object's record. The `Distance()` function operates on these vectors and returns a scalar distance value. We require that the scalar distance value obey the triangle inequality. In other words, we require that $distance(u, v) + distance(v, w) \geq distance(u, w)$.

The above interface is flexible and supports the needs of very different applications. For example:

- *Scalar measurements*: In applications involving scalar sensor values, `Vectorize()` trivially returns the scalar sensor measurement. `Distance()` returns the difference between two measurements.
- *Vector measurements*: In applications where objects such as, environmental measurements, are associated with metadata, such as time and location, `Vectorize()` might focus on metadata elements of objects, viewed as a feature vector. `Distance()` might then return a weighted Cartesian distance between feature vectors, where weights reflect the relative impact of differences in the corresponding dimension on the likelihood of similarity between objects. For example, say, we know that a particular variable does not change much over time, but has large spatial variations. Hence, the weight of the location dimension is set larger and the weight of the time dimension is set smaller. This allows computing a scalar similarity measure between any two objects and estimating measurements at one time and location

using a nearby object in the feature space (albeit from a different time and location).

- *Pictures*: In applications involving visual objects, `Vectorize()` might apply a library of image processing tools to extract relevant image features. `Distance()` may compute visual similarity between images based on these features.
- *Text and tags:* In applications where objects constitute small amounts of text (such as tweets or tags associated with images), `Vectorize()` might split the text entry on whitespaces into different tokens (words). `Distance` may be applied on pairs of vectors (token lists) by counting the proportion of similar tokens. The Tanimoto distance and the Angular distance are suitable distance metrics in this space [15], [16].

The point of the above discussion is to demonsrate versatility. Many application domains (e.g., vision and speech) already have well-defined distance metrics between objects. The definition of vector spaces and distance metrics is thus out of scope for SocialTrove. In our case study, we demonstrate a distance defined on short text (tweets), showing how it leads to meaninful summaries of human observations.

B. The Summary Query Interface

Using the above two application-specific callback functions, SocialTrove has all it needs to perform hierarchical clustering in real time, as will be described later in this paper. With clusters at different levels of granularity constructed, SocialTrove exports an interface to retrieve data summaries at different degrees of granularity. A summary in our service is given by a list of cluster-heads. For each cluster-head, the service allows one to optionally retrieve a member count (i.e., count of objects in the same cluster) or a member list (list of object record handles for objects in the same cluster). Remember that an object record is a tuple, (`ObjectSource`, `ObjectHandle`, `FeatureVector`), specifying the source ID, feature vector and object handle. Hence, given a list of record handles, the application can retrieve the corresponding objects, sources, or features, depending on how much data they need.

For example, an application interested in the degree of data corroboration only, might retrieve a summary that consists of cluster heads and member counts only. An application that also needs to know which sources reported the observations in the cluster (e.g., in case some are trusted more than others), can retrieve the member (handle) list and inspect the sources. An application interested in statistics over clusters may also inspect the feature vectors. The SocialTrove runtime is described in the following sections.

III. SocialTrove Runtime

SocialTrove is designed for large-scale social sensing services where collected data is too big for a single machine. Hence, we design and implement SocialTrove on a machine cluster. In this section, we describe the design of the runtime environment that makes SocialTrove scalable. The design is based on two observations:

- Latency and throughput are improved by limiting global state updates to only once per a configurable interval, called the *batching interval*. Hence, incoming data are buffered until enough of it is present, then a batch process makes an update to existing clusters, once per batching interval. Batching amortizes runtime overhead across a larger body of input data. The batching interval (e.g., 5 minutes) is thus a configurable parameter that offers a trade-off between data freshness and update overhead.
- Availability is improved by noting that social sensing content is likely to exhibit temporal locality. Hence, state does not change significantly across batching periods, making further optimizations possible.

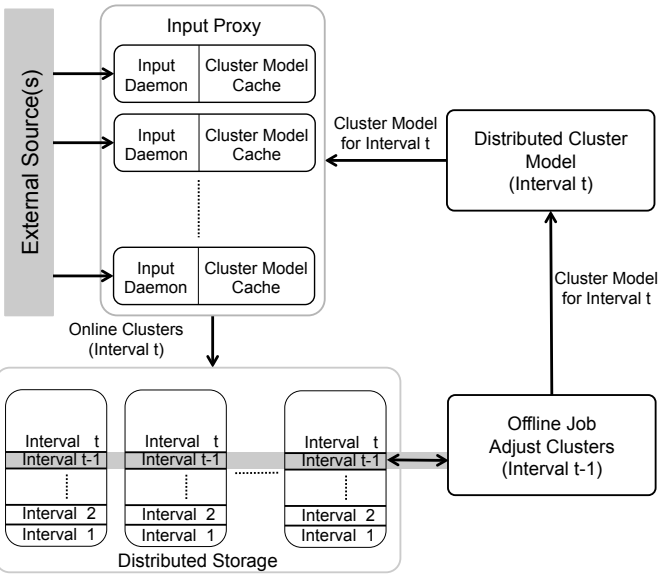

Fig. 1. SocialTrove System Design

A. System Components

Figure 1 shows the components of SocialTrove and their interactions, described below.

1) Data Input Proxy: We envision SocialTrove to sit on top of a data collection service. This service will interact with the various sources and will supply a stream of real-time data to SocialTrove for summarization. In the current implementation, the input is supplied as a set of tuples (`ObjectSource`, `ObjectHandle`) in JSON [17] format. In our particular application example, we replace the data collection service with Twitter and write a simple interface that uses Twitter API to stream tweets. In this instantiation, `ObjectSource` is a Twitter user ID, and `ObjectHandle` is a handle to a tweet object (including text and metadata).

The input proxy is composed of several data input daemons that receive streaming objects and must resolve where to store each. This resolution is done by consulting a *Cluster Model*, which keeps track of the existing clusters for the present batching interval and the mapping from these clusters to individual storage machines.

2) Client Query Proxy: Similarly to the input daemon nodes, are the client query proxies. (The proxies are not shown in Figure 1 to keep it simple.) They function like

input daemons and cache the cluster model as well. Instead of clustering collected data from external sources, the query proxies receive queries from SocialTrove clients, and fetch the matching data summaries from the storage nodes using their locally cached cluster model.

3) Cluster Model: The Cluster Model is a data structure that contains the set of cluster centroids, along with their hierarchical relationships. Computing an accurate cluster model requires knowledge of all the data objects, including those that would be arriving in future. Because the data objects arrive as a stream, having an accurate model is often not possible. Maintaining a streaming cluster model that updates the existing clusters as the data objects arrive would be close to accurate [18]. In this scenario, the input daemons would require exclusive locks to update the model at every insertion, and all the proxies would need to synchronize the updates to maintain consistency. Such a write-heavy scheme would greatly reduce both throughput and response time of the system, and would not be scalable as a service.

To solve this problem, SocialTrove maintains a system wide batching interval of Δ minutes. A new cluster model is computed using the recently collected data objects, and advertised at the beginning of every interval. The input daemons and the client query proxies cache the cluster model (or portions of it) in their main memory that remains consistent until the interval ends. In later sections, we discuss different solutions to organize and update the cluster model.

4) Storage: The storage nodes store actual data objects in a clustered form. The objects are received from the input daemon nodes that cluster incoming data objects using the cluster model. The clusters stored in the storage nodes are partitioned and indexed according to the interval they were received. For a particular interval, the union of the respective partitions over all the storage nodes constitutes the 'sensed universe' for that interval.

5) Model Update Routine: The model update routine is run every batching interval of Δ minutes. During interval t, it considers the data objects received in interval $t-m$ to $t-1$ (the previous m intervals), and computes the cluster model that the data input and the client query proxies will use during interval $t+1$ (the next interval). As an option, output of this routine can be fed back to the storage nodes so that the data objects received during interval $t-1$ could be readjusted.

B. Cluster Model Management

The Cluster Model is a key part of SocialTrove. It maintains a set of centroids as the cluster heads of the existing clusters. For an incoming data object, the input daemons traverse the cluster model to find the centroid of the cluster this object belongs to. Similarly, to serve the applications running on top of SocialTrove, the client query proxies traverse the cluster model to find matching vectors. Depending on how the distributed cluster model is organized and maintained, there can be different trade-offs and flexibilities the system can offer.

SocialTrove is scalable by virtue of efficient realization of these insert and lookup queries. If there are k centroids and n incoming data objects in an interval, the naive and most versatile implementation requires a query object to be compared with all the centroids to find the nearest match, resulting in a $\mathrm{O}(nkd)$ algorithm when the whole cluster model fits in the cache and the comparisons take $\mathrm{O}(d)$ time. The comparison time can be considered a constant. We also observe that the cluster sizes in socially sensed data objects approximately follow a long tail distribution, and k is roughly of the same order as n. Hence, the naive algorithm requires $\mathrm{O}(n^2)$ time when the entire cluster model fits in the cache. This naive solution would not be scalable.

If object distances, however, follow the triangle inequality, some distances can be inferred from others and hence the above extensive comparison is an overkill. Given a metric distance space, we thus build a nearest neighbor data structure (tree) during clustering. The insert and search operations on the clustered data objects can then be performed efficiently using the tree. Disjoint partitions of the tree are mapped to different storage nodes, so that an input daemon can quickly decide which storage node to forward the incoming data object, and a query proxy can quickly decide which storage node(s) to forward the user query to.

If the distance function satisfies all the properties of a metric (non-negativity, small self-distance, isolation, symmetry, and triangle inequality) [19], it enables us to use rich Nearest Neighbor data structures like M-tree [20] or Ball-tree [21] to perform k-`means` [22] clustering efficiently. It is trivial to satisfy the first four properties. The triangle inequality may not be satisfied by all distance measures. However, if any of the last three conditions fail; provided the other four are satisfied, it is possible to find a function through transformation, which is a metric function [19].

The euclidean distance function follows triangle inequality and is a metric function. In fact, all normed vector spaces are metric spaces, if we define $d(\mathbf{x},\mathbf{y}) = \|\mathbf{x}-\mathbf{y}\|$. Some distance measures like KL-Divergence or Mahalanobis Distance do not follow triangle inequality, but instead follow another property called Bregman Divergence. There are Nearest Neighbor data structures inspired by Ball-tree; for example Bregman Ball-tree [23] that can be used in this case for efficient clustering. These, however, are currently not implemented on SocialTrove.

In SocialTrove, the cluster model is represented as a binary tree of centroids. The tree is constructed using a divide and conquer paradigm. At every stage, the current set of vectors is partitioned into two sets, using a 2-`means`[1] clustering algorithm. The centroids of the two sets are considered as the two children of the centroid of the original set. This process continues until we arrive at a set of vectors with diameter less than a threshold, which is considered as a single indivisible cluster. The data objects are separable in this way, provided the distance function satisfies the triangle inequality (and all the properties of a metric). The tree is generated by the model update module, and synchronized every interval to the input daemon and the proxy nodes that cache it. As an illustrative example, Figure 2 shows a set of points in two dimensional space and maps those to a corresponding binary tree that divides the space using the euclidean distance among the points as a distance metric.

For each collected object, the input daemons find the closest centroid from the tree, and assign it to the corre-

[1] k-`means` clustering with $k = 2$

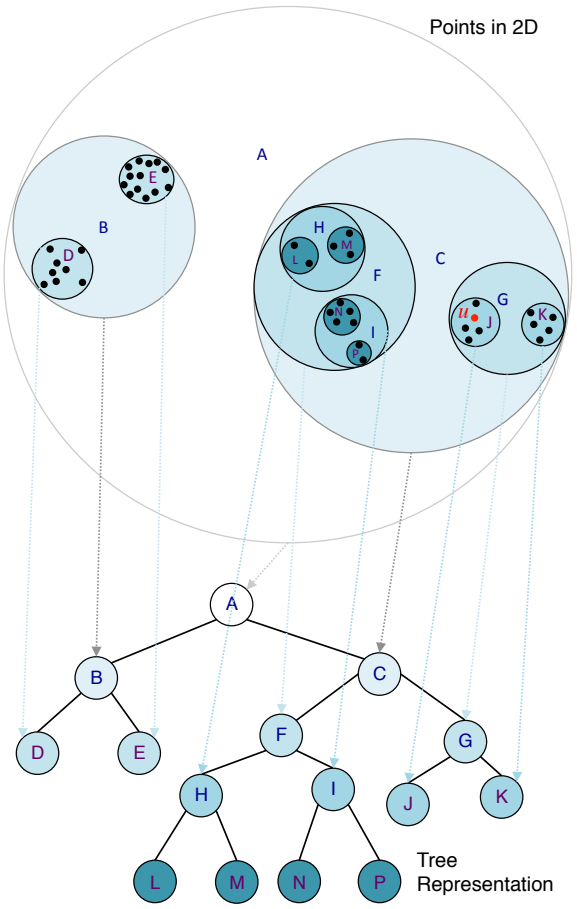

Fig. 2. Mapping a set of points in two dimensions to a tree

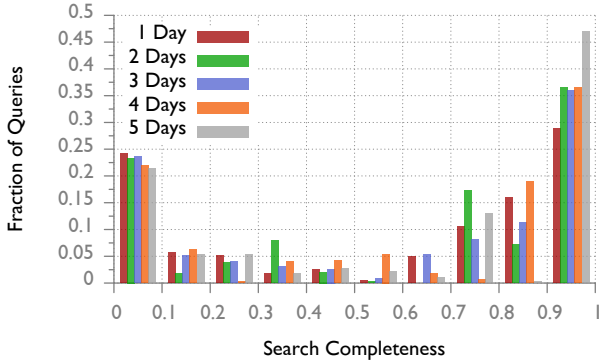

Fig. 3. Distribution of Search Completeness

sponding cluster. If there are k centroids, this operation can be performed quickly, in $O(\log k)$ time. However, for the clustering performed this way to be correct and the lookup operations to succeed, the nodes of the tree requires perfect centroids for all objects that would be collected during the current interval, which is not possible.

We assume that objects collected in the present interval are correlated with those collected in past intervals. Hence, we estimate the cluster tree for interval t during interval $t-1$ by clustering the objects collected in last m intervals (i.e., intervals $t-m-1$ to $t-2$).

To check the validity of our assumption, experiments were performed using Twitter data as the input by clustering past tweets to build the cluster tree, and inserting new tweets using it. The objective was to check how complete the lookup operations would be, if a scheme for quickly clustering recent tweets based on a past model is used. The hashtags present in the current set of tweets were then used as search queries. We used a very large Δ, of 1 day, as a very extreme case. Figure 3 shows the distribution of search completeness for the newest tweets for different values of m from 1 day to 5 days.

The plot confirms that tweets from present and past intervals are correlated. The plot also reveals one potential limitation of this method; false negatives. Over 20% of user queries could not find any match at all, and 40% could only find at most 50% of the desired results. This problem is visible with high dimensional data like text or tweets, where some dimensions were not known when the summary model was generated. As the cluster tree is computed and circulated to the input daemon nodes in synchronous intervals, it fails to look up using query terms that are unique to the present interval.

As a solution, we add an asynchronous component to the cluster model using Bloom filters [24]. There are Bloom filters corresponding to every node of the cluster tree. Dimensions (keywords, in case of tweets) unique to the present interval are locally inserted to the Bloom filters corresponding to the tree nodes visited by an incoming data object. Crawlers use a gossip protocol [25] to propagate their local updates to the Bloom filters. These updates are not expensive because only relative changes are sent over the network, which are easily merged using bitwise ORing. Lookups are performed using the Bloom filter. A Bloom filter has a 100% recall rate; hence it solved the aforementioned problem of false negatives when searching with the query terms unique to the present interval.

1) Insertion: New object insertions use the cluster model to find the correct cluster for incoming objects. Here, we illustrate using Figure 2 how insertions are performed. Suppose an incoming object u (the red point in Figure 2) arrives. To assign the nearest cluster to this point, it is at first compared with centroids B and C. Distance from centroid C is found to be lower. Thus, the object is pushed down that branch of the tree. Centroid C has two children, namely F and G. Again, object u is compared to both. The distance from G is found lower. Thus, the object is pushed down that branch. The two children of G (namely J and K) are compared to u next. The incoming object is closter to J, which is a single cluster. Hence, u is assigned to cluster J.

The pseudo-code for insertion using the cluster tree is shown in Figure 4. $node_{sync}$ corresponds to the synchronized component of $node$ that is updated every interval, and $node_{async}$ corresponds to the asynchronous components that are maintained through Bloom filters. Lines 4–19 push the incoming object u down the tree. Lines 9–15 compare the new point with the two children of the presently considered node of the tree and decide which branch to take next. As the incoming object traverses down the tree, the local Bloom filters of the corresponding nodes are updated (which would be later propagated to the data input and the client query proxies). In Line 20, the cluster that u belongs to has been decided and the corresponding cluster summary is pushed to the in-memory distributed cache at this point.

```
1: procedure INSERT(u)                                  ▷ Data object u
2:     node_sync ← root(CM_sync)                        ▷ Global cluster model
3:     node_async ← root(CM_async)                      ▷ Local cluster model
4:     while node_sync is not leaf do
5:         l_sync ← left(node_sync)                     ▷ Left child
6:         l_async ← left(node_async)
7:         r_sync ← right(node_sync)                    ▷ Right child
8:         r_async ← right(node_async)
9:         if dist(u, l_sync) < dist(u, r_sync) then
10:            node_sync ← l_sync
11:            node_async ← l_async
12:        else
13:            node_sync ← r_sync
14:            node_async ← r_async
15:        end if
16:        for all token ∈ u do
17:            Set node_async[token]                    ▷ Update Bloom filter
18:        end for
19:    end while
20:    Append u to the cluster node_sync                ▷ Invoke RPC
21: end procedure
```

Fig. 4. Algorithm to insert an object

```
1: procedure LOOKUP(w, d_q)                             ▷ Query object w
2:     node_async ← root(CM_async)                      ▷ Local cluster model
3:     result ← ∅                                       ▷ Set of matching objects
4:     if dist(w, node_async) ≤ d_q then
5:         EXPLOREBRANCH(w, d_q, node_async, result)
6:     end if
7:     return result
8: end procedure

9: procedure EXPLOREBRANCH(w, d_q, node, result)
10:    if node is not leaf then
11:        l ← left(node)                               ▷ Left child
12:        if dist(w, l) ≤ d_q then
13:            EXPLOREBRANCH(w, d_q, l, result)
14:        end if
15:        r ← right(node)                              ▷ Right child
16:        if dist(w, r) ≤ d_q then
17:            EXPLOREBRANCH(w, d_q, r, result)
18:        end if
19:    else
20:        Append cluster node to result
21:    end if
22: end procedure
```

Fig. 5. Algorithm to lookup cluster summaries

2) Lookup: Lookups use the asynchronous component of the cluster model to find the correct cluster summaries related to an incoming query. Please note that, for an insertion, the incoming data object is assigned to only one cluster, which is nearest from it. The incoming data items are expected to follow the trend of the existing clusters, so that the summary model can be used to find the nearest cluster. However, for a lookup, the queries can be any point in space. The response is a set of cluster summaries within a mentioned distance from the query.

Figure 5 presents the pseudo code. Lines 4–5 decide if the query object w is within a specified distance d_q of the root node. If it is not, it is decided that the query does not match any of the existing summaries in the model. If the distance is within d_q, Lines 10–21 traverse the tree, taking the branches for which the distance of the centroid is less than the specified threshold d_q, and pruning when it is not.

```
1: procedure GENERATEMODEL(S, d_c)                      ▷ Set of objects S
2:     root ← mean(S)                                   ▷ Calculate centroid of S
3:     if diameter(root) > d_c then
4:         TWOMEANSMODEL(root, d_c)                     ▷ Non-blocking
5:     end if
6: end procedure

7: procedure TWOMEANSMODEL(node, d_c)
8:     ▷ node must be divisible in atleast two clusters.
9:     ▷ TWOMEANS uses 2-means clustering to
10:    ▷ partition node into two clusters l and r.
11:    (l, r) ← TWOMEANS(node)                          ▷ MapReduce job
12:    left(node) ← l                                   ▷ Assign l as left child
13:    right(node) ← r                                  ▷ Assign r as right child
14:
15:    ▷ Calls to TWOMEANSMODEL are independent,
16:    ▷ asynchronous, and can be scheduled in parallel.
17:    if diameter(l) > d_c then
18:        TWOMEANSMODEL(l, d_c)                        ▷ Non-blocking
19:    end if
20:    if diameter(r) > d_c then
21:        TWOMEANSMODEL(r, d_c)                        ▷ Non-blocking
22:    end if
23: end procedure
```

Fig. 6. Algorithm to generate summary model

3) Model Update: Model update is an offline job that runs once per batching interval. It considers the objects collected in the previous m intervals, and constructs the cluster model by repeatedly performing 2-means clustering. Because the distance function satisfies triangle inequality, divisions performed at each stage are independent, and are scheduled in parallel for further division.

Figure 6 presents the pseudo code. d_c is a threshold parameter the algorithm uses to decide if the current set of objects are distant enough to be partitioned into two clusters. Line 2 initializes the root node of the tree. Line 11 calls the TWOMEANS procedure to perform a 2-means clustering. In reference to Figure 2, if C is the current set of points, F and G are calculated in line 11. For a large set of data objects, this is an expensive operation, and we use a MapReduce framework to parallelize the workload [26]. Lines 12–13 updates the tree with the newly calculated centroids. At this point, the problem has been divided into two independent subproblems. Line 18 and 21 schedule new invocations of TWOMEANSMODEL in parallel, and the process continues until the diameter of the current set is less than d_c.

C. Implementation

SocialTrove runs in UIUC Green Data Center [27]. We use Python to implement a data collection service to provide input data. Apache Thrift [28] is used as a Serialization and

RPC framework. Memcached [29] is used as a distributed in-memory cache layer for the input data and the client query proxies. Apache Hadoop [30] and Spark [31] are used for offline analytics.

The input data objects are sent by input data daemons to be stored in a Hadoop Distributed File System (HDFS) [30]. There are 23 machines with one 6-core 2.0 GHz processor (Intel Xeon E5-2620), 16 GB memory, and 1 TB of storage. The model update routine has been implemented using Java, which runs on Apache Spark [32], in a subset of the available machines. Spark has been configured to run in Standalone mode (i.e., without Yarn [30] or Mesos [33]). Each of the Spark slaves runs one worker process using 12 GB memory. Due to higher memory requirements of the Spark tasks, 30% of the memory is reserved for caching the RDDs (Resilient Distributed Datasets) [34] instead of the default allocation of 60%. The remaining 9 GB is available for the Java heap.

A machine with two 10-core 3.0 GHz processors (Intel Xeon E5-2690 v2) and 128 GB memory works as the driver machine. The driver machine commands the worker machines to build RDD (Reslilient Distributed Datasets) using the data that has been collected over the past interval. It uses the GENERATEMODEL algorithm (Figure 6), which generates a summary model by repeated use of 2-`means` clustering as a subroutine to bisect the distributed dataset. At each step, two objects are randomly selected from the present dataset that act as initial centroids. These two centroids are broadcast to all workers. After this broadcast, the driver machine initiates a map phase (known as RDD Transformation in Spark) so that the workers calculate the distance of each object in its collection from the two broadcast centroids, and assigns it to the centroid with the smaller distance. After that, the driver issues a reduce phase (RDD Action) that calculates two new centroids from the previous assignments. The new centroids are broadcast to the workers again, and the process continues until it converges and results in two clusters. The parent RDD is then partitioned into two child RDDs corresponding to the newly formed clusters, each of which are scheduled to run TWOMEANSMODEL (Figure 6), in parallel. Due to the overhead of small jobs, once an RDD becomes small enough, we start bisecting it in a single thread, instead of spawning new Spark jobs. Once clustering is complete, the data input proxies and the client query proxies update their cluster models accordingly.

IV. EVALUATION

We evaluate SocialTrove in the context of summarizing tweets. Each tweet is represented as a high dimensional vector of tokens in our vector space model. We use Tanimoto distance [16], which obeys triangle inequality [15], and can be considered as a vector expansion of the Jaccard distance [16]; a good measurement of similarity between high-dimensional sets. The objective of the evaluation is to answer the following questions:

- For summarization to be a service, is it necessary to precalculate a summary model? Instead of building a summary model in advance, can we generate the summaries only for the related tweets on demand as the queries arrive?
- Where and how much do we gain by organizing the summary as a hiearchy? Instead of building a tree, can we build a reverse index from keywords to list of tweets (or tweet summaries)? Such techniques are used by the Internet search engines, and are able to support a high request throughput.
- How well does SocialTrove scale out?

Live tweets crawled from Twitter via its search API are subject to rate limits. Hence, we merge tweets collected during several events in the physical world, and play those back to SocialTrove. The events include Crimean Crisis (February 2014), Sochi Winter Olympics (February 2014), Syria Chemical Attack (August 2013), Boston Marathon Bombings (April 2013), Hurricane Sandy (October 2012), Hurricane Irene (August 2011), England Riots (August 2011), Fukushima Nuclear Disaster (March 2011), Egyptian Revolution (January 2011), etc. This combined set contains 4 142 586 tweets.

To the best of our knowledge, SocialTrove is the first system to offer summarization of social streams as a cloud-backend service. We did not find any corresponding system in the state of the art. Hence, we compare the performance by replacing SocialTrove components and algorithms with the following options:

- **Baseline** In this scheme we do not build periodic summary models. The input daemon randomly picks a storage machine for an incoming tweet even without deserialization (JSON parsing). To perform a lookup, the client proxies broadcast the query to all machines. The machines collect matching tweets, cluster those, and return a representative sample.
- **Indexing** This scheme does not precalculate a summary model. However, it maintains a keyword-to-storage map in memory. An incoming tweet is scanned to find keywords. For every keyword occuring in the tweet, we consult the map, and assign the corresponding storage machines. To perform a lookup, we cluster the matching tweets collected from the storage machines, and present a representative summary.
- **Summary Baseline** This scheme precalculates a summary model. It opts for a flat organization of the cluster summaries. At every interval, the model is pushed to the data input, and the client query proxies, just like SocialTrove. To assign an incoming tweet to an existing cluster, this scheme computes the distance to all the existing clusters and finds the nearest one. To perform a lookup, this scheme again needs a linear search through the list of summaries.
- **Summary Indexing** This scheme is derived from techniques used by the Internet search engines. It computes the summary model, and organizes those by reverse indexing from keywords to the list of summaries. To insert a new tweet, a data input proxy extracts the keywords from it, searches only the reverse indexes corresponding to those keywords, and finds the nearest summary to assign the tweet. To perform a lookup, this scheme needs to scan the list of reverse indexes corresponding to the given keywords, and find the matching summaries.

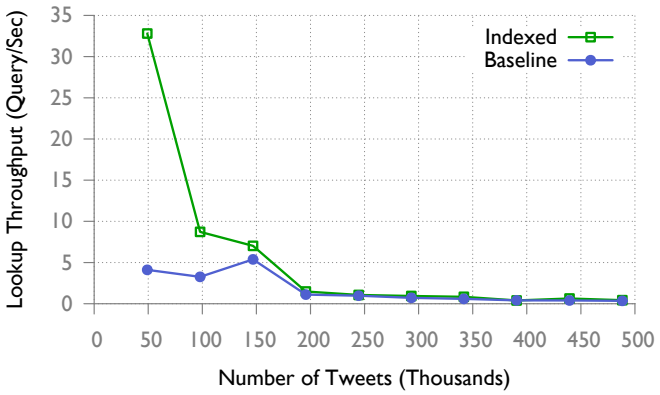

Fig. 7. Without a summary model, lookup throughput is very low

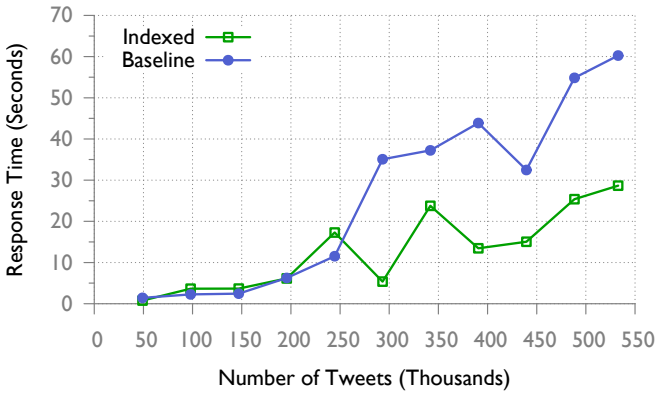

Fig. 9. Response time for a request is often high without a summary model

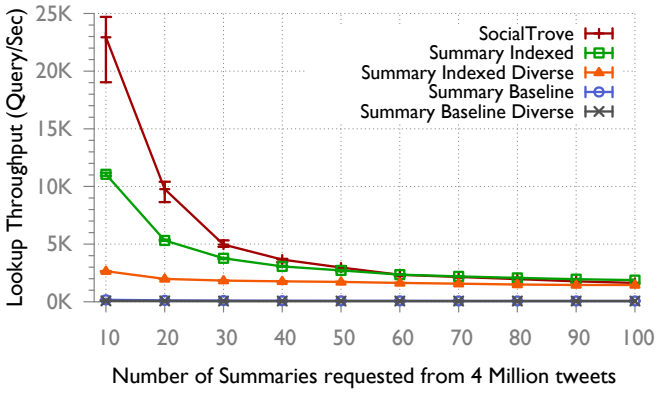

Fig. 8. SocialTrove offers high throughput for small to medium sized requests

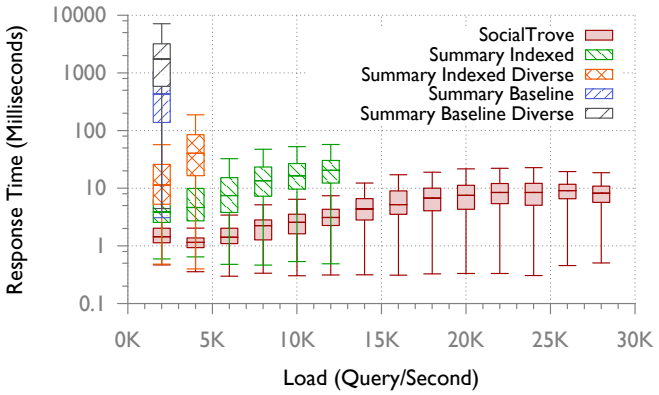

Fig. 10. SocialTrove has lower response time compared to the other methods

A. *Query Throughput*

In our application, a query is a set of keywords. In response to a query from the application, the Client Query Proxy prepares a representative summary of the tweets that contain the given keywords. Please note the difference between returning all the results and returning a *representative summary*. The former is the application of known data structures and storage systems that can return all the matching objects. However, SocialTrove is a summarization service to deal with information overload, and as such, returns a representative sample (i.e., cluster-heads). The queries can also include an optional distance parameter, which specifies the minimum diversity among the returned samples.

Figure 7 shows that the throughput is very low for the Baseline and the Indexed methods that do not pre-calculate a summary model. These methods calculate a summary on demand, in response to a query. The baseline option suffers the most in lookup throughput as it is putting load on every worker machine for every query. The indexed meachanism would have a high throughput if the queries would ask for all matching data objects instead of a representative summary. It performs better than the baseline because of the underlying indexing that provides it the set of candidate tweets without searching. However, the throughput quickly falls off towards zero as the number of tweets in the universe increases.

SocialTrove client query proxies cache the summary model in their memory once it is generated. For every query request, it traverses the tree according to the Algorithm in Figure 5 and finds the corresponding leaves. To answer queries, it consults the distributed in-memory cache (Memcached) to fetch a sample tweet from each of the clusters. Figure 8 compares SocialTrove with the other methods that prepare a summary model in advance, in a universe of 4 Million tweets. SocialTrove can sustain a much higher throughput compared to the other methods, because of the hierarchical organization of the summary model. It can also incorporate the distance parameter (diversity) without any overhead because the tree had already calculated and cached the necessary distance information. The Summary Indexing method offers roughtly 50% of SocialTrove throughput when the number of objects requested is small (around 20). On the other hand, the Summary Indexing method suffers when the diversity parameter is included. Baseline Indexing has the lowest throughput with and without the diversity parameter because of the lack of organization in the cluster summaries.

The evaluation presented here shows that when serving small requests like updating a web-page with the cluster summaries, or showing a set of tweets on a cellphone screen, SocialTrove allows high throughput. This is particularly a useful aspect of SocialTrove, because the user-facing applications often need a ‘concise’ amount of useful information.

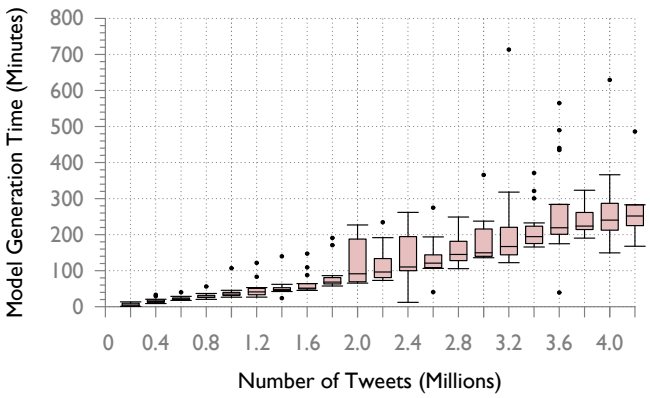

Fig. 11. Time to generate summary model using 8 worker machines

TABLE I. EFFECT OF PARALLELIZING MODEL GENERATION

Number of Workers	Model Generation Time
4	337 minutes
8	239 minutes
11	195 minutes

B. Query Response Time

In this section we measure and compare the query response time of SocialTrove and the alternate mechanisms. Figure 9 shows that the Baseline and Indexed methods that do not precompute a cluster summary do not scale. These mechanisms are acceptable as a service only when the number of data objects that pass through the system at every interval is very low. Twitter receives around 500 Million tweets per day [35] (or 20 Million per hour), so clearly precomputing a summary model is necessary.

Figure 10 compares the response time between the methods that precompute the summary model in advance. We measure the response time at various levels of load (number of requests per second) in a universe of 4 Million tweets, and observe that SocialTrove responds in $10\,\mathrm{ms}$ under heavy load, and nearly in $1\,\mathrm{ms}$ under light load. The summary indexing method is acceptable only when the system is lightly loaded (around 5K queries per second). If the diversity parameter is added, the summary indexing method suffers even more due to the additional distance calculations to ensure diversity.

We conclude that SocialTrove performs best, because it (1) precalculates the cluster summaries, (2) organizes the summaries as a tree, which prunes many options and reduces the search space, and (3) makes it possible to cache the summary model in main memory. If the summary model was not cached, traversing the tree would require at least one RTT (round trip time) in the network, reducing both throughput and response time. On the other hand, caching the summary model has been possible by allowing updates to the model only in synchronous intervals. This is how SocialTrove avoids a write-heavy data structure and cache consistency issues.

C. Cluster Model

We now present the time it takes SocialTrove to generate a summary model using Spark. Figure 11 shows the time in minutes, for differernt number of tweets as input, using 8 worker machines. k-means (in our case, $k = 2$) clustering algorithms sometimes converge to local minima, which in our case translates to unbalanced partitioning at some stages, requiring more time to finish. This is the reason for the variability in the summary generation time. Table I shows the effect of parallelizing the clustering workload by comparing the median model generation time for 4 Million tweets with different number of worker machines.

TABLE II. TOP SUMMARIES

Thousands at Moscow rally against Russian intervention in #Ukraine: http://t.co/6U0AIOOgQv http://t.co/kobbd7KzXY
Man in Ukraine plays the piano to help calm down a riot. http://t.co/fdNAc0cfJ2
For Crimea, Google Shows Different Borders Based on Your Location: Russia's Minister of Communications and Mass Media http://t.co/vIHGYlibOC
Militants in eastern #Ukraine were equipped with Russian weapons and the same uniforms as those worn by Russian forces that invaded Crimea.
50,000 #Ukraine supporters march in Moscow to protest Russia's intervention in #Crimea. http://t.co/qMJjYgPNxI
Some russian tanks on ukrainian border already painted with 'peacekeeping' slogans. How much longer until the 'humanitarian intervention'?
I've been speaking to @BarackObama about the situation in Ukraine. We are united in condemnation of Russia's actions. http://t.co/7Rk2k8iOIK
Ukrainian Defense Ministry says its lone submarine has been taken by Russians. http://t.co/lj1XP4q1BX http://t.co/mDDhQ2lqAO
Ukraine prepares armed response as city seized by pro-Russia forces http://t.co/ahVX7lKftT
Ukraine crisis: Nato warns Russia against further intervention - BBC News http://t.co/GtdmRMxAPI

Table II shows sample output of the summary tweets ranked by a fact-finder application on top of SocialTrove. Our application queried SocialTrove for the set of summaries related to the keywords {Crimea, Ukraine, Russia} and ranked them according to the algorithm in [8]. Note how the tweets offer a quick insight into the highlights of the current event. A detailed evaluation of this specific application is beyond the scope of this paper.

V. RELATED WORK

SocialTrove is motivated by the needs of data-intensive applications that handle sensor or social media data. We consider social sensing applications where redundant data are collected from people or sensors in their possession. For exampe, CabSense [36] is a crowd-sourced service that collects information on taxi cab fleets. Mediascope [37] describes a media retrieval service to query and retrieve photos taken by people directly from their mobile devices. Another recent service uses Twitter as a sensor network and models humans as noisy sensors to report and summarize ongoing events [8].

To reduce the inherent redundancy in data reported by such services, a clustering algorithm is needed. A very common one is k-means clustering [22]; an iterative method that repeats between selecting k means as centroids, assigning the rest of the points to the means based on similarity, and recalculating the means. Our paper uses a special form of the k-means algorithm, where $k = 2$, repeatedly bisecting a data set to form a hierarchy. The k-means algorithm is sensitive to its initialization. Different efforts have addressed this problem. For example, k-means++ [38] avoids the issue and can be applied to SocialTrove in a straightforward way. For another example, the Buckshot Clustering algorithm [39] combines HAC (Hierarchical Agglomerative Clustering) and k-means.

It selects $O(n)$ points randomly and runs a group average on this sample, which takes $O(n)$ time. Using the result of HAC as initial seed for k-means avoids the bad seed selection problem.

There are also streaming versions of k-means. Ailon et al. [40] run online facility location algorithm on a data stream of size n, to arrive at a partial solution with $k\log(n)$ clusters. The partial solution is followed by a ball k-means step to reduce the number of clusters to k. Shindler et al. [18] simplify the algorithm, which results in a better approximation guarantee. DS-means [41] describes a distributed algorithm to cluster data streams in a p2p environment. This system mainly uses the distributed k-means algorithm described by Bandyopadhyay et. al. [42], along with local instances of X-means [43] and gossip propagation to converge to the actual number of clusters in the system. We do not directly incorporate streaming algorithms in SocialTrove due to the need for model updates, required upon insertions. Instead, we use a batching interval to update the summary model, and exploit the "slow-changing" nature of social sensing observations in between updates.

To attain scalable implementations of data processing services, one common execution model is MapReduce [44]. MapReduce, however, is not efficient for a large class of vertex parallel iterative algorithms that have a substatial data shuffling phase. Pregel [45] is another scalable and fault-tolerant platform to implement the vertex parallel iterative algorithms in a distributed environment using the Bulk Synchronous Parallel (BSP) [46] execution model. A limitation of MapReduce or Pregel is that the results of each round are stored on disk to be read again in the next step. Spark [31], in contrast, is an in-memory cluster computing framework that uses Resilient Distributed Datasets (RDD) to record the lineage of operations on the datasets instead of storing the data. Once a fault occurs, the lineage can be traversed to recover from the fault. Trinity [47] is another in-memory distributed platform for iterative computation. It partitions the dataset over the main memory of individual machines and essentially implements a distributed in-memory key-value store. Other systems include Storm [48], a distributed and fault-tolerant framework for processing streams in real-time, and SparkStreaming [31], which uses RDDs for streaming workloads. SocialTrove uses Spark to generate the summary model because the main building block of that algorithm is 2-means clustering, which is a data parallel iterative algorithm. Typically many rounds of iterations on many subsets of the data are necessary, along with back and forth communications with the driver machine. The in-memory computation reduces the inter-round overhead and latencies.

Memcached [29] is an in-memory key-value store, often utilized to mask latencies from external data sources by caching results [49]. SocialTrove uses Memcached [29] as the distributed in-memory cache for the data input and the client query proxies, which improves throughput and response time of the queries. Druid [50] is a distributed column-oriented real-time OLAP system that uses a combination of real-time nodes and historical nodes to answer both real-time queries and historical aggregate queries. Compared to Druid, SocialTrove is not limited to structured time-series data. Rather SocialTrove can process arbitrary streaming data, aimed at social sensing services. Druid emphasizes fast ingestion for real-time queries, whereas SocialTrove provides a flexible summarization service by allowing the users to define a summarization criteria.

VI. Conclusions

In this paper, we described SocialTrove; an information summarization service for social-sensing. The design of the service is motivated by the advant of an age of information overload, where much data is generated in real-time, and where redundancy is common. SocialTrove delivers data summaries at arbitrary levels of granularity by reducing redundancy through clustering. Evaluation shows that SocialTrove is scalable in serving data summaries because it caches a cluster summary model in memory for a predefined interval, which allows it to provide high throughput, low-latency lookups for real-time social sensing data, without incurring signficiant insertion overheads. It outperforms traditional indexing methods, which incur a heavier latency and suffer from lower throughput. A limitation of the current evaluation is that it tests SocialTrove only in the context of Twitter data summarization. Future work of the authors will focus on exploring the benefits and performance of SocialTrove in summarizing other types of large streaming data.

Acknowledgements

Research reported in this paper was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement W911NF-09-2-0053, DTRA grant HDTRA1-10-1- 0120, and NSF grants NSF CNS 13-29886, CNS 09-58314, and CNS 10-35736. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] D. Estrin, "Participatory sensing: applications and architecture [internet predictions]," *Internet Computing, IEEE*, vol. 14, no. 1, pp. 12–42, 2010.

[2] M. Mun, S. Reddy, K. Shilton, N. Yau, J. Burke, D. Estrin, M. Hansen, E. Howard, R. West, and P. Boda, "Peir, the personal environmental impact report, as a platform for participatory sensing systems research," in *Proceedings of the 7th international conference on Mobile systems, applications, and services*. ACM, 2009, pp. 55–68.

[3] R. K. Ganti, N. Pham, H. Ahmadi, S. Nangia, and T. F. Abdelzaher, "Greengps: a participatory sensing fuel-efficient maps application," in *MobiSys '10: Proceedings of the 8th international conference on Mobile systems, applications, and services*. New York, NY, USA: ACM, 2010, pp. 151–164.

[4] S. Reddy, D. Estrin, and M. Srivastava, "Recruitment framework for participatory sensing data collections," in *Pervasive Computing*. Springer, 2010, pp. 138–155.

[5] A. T. Campbell, S. B. Eisenman, N. D. Lane, E. Miluzzo, R. A. Peterson, H. Lu, X. Zheng, M. Musolesi, K. Fodor, and G.-S. Ahn, "The rise of people-centric sensing," *Internet Computing, IEEE*, vol. 12, no. 4, pp. 12–21, 2008.

[6] N. D. Lane, S. B. Eisenman, M. Musolesi, E. Miluzzo, and A. T. Campbell, "Urban sensing systems: opportunistic or participatory?" in *Proceedings of the 9th workshop on Mobile computing systems and applications*. ACM, 2008, pp. 11–16.

[7] M. Shin, C. Cornelius, D. Peebles, A. Kapadia, D. Kotz, and N. Triandopoulos, "Anonysense: A system for anonymous opportunistic sensing," *Pervasive and Mobile Computing*, vol. 7, no. 1, pp. 16–30, 2011.

[8] D. Wang, M. T. Amin, S. Li, T. Abdelzaher, L. Kaplan, S. Gu, C. Pan, H. Liu, C. Aggarwal, R. Ganti, X. Wang, P. Mohapatra, B. Szymanski, and H. Le, "Humans as sensors: An estimation theoretic perspective," in *Proceedings of the ACM/IEEE Conference on Information Processing in Sensor Networks (IPSN'14)*, 2014.

[9] D. Wang, L. Kaplan, H. Le, and T. Abdelzaher, "On truth discovery in social sensing: A maximum likelihood estimation approach," in *ACM/IEEE Conference on Information Processing in Sensor Networks (IPSN)*, 2012.

[10] M. T. A. Amin, T. Abdelzaher, D. Wang, and B. Szymanski, "Crowdsensing with polarized sources," in *Proceedings of the 2014 IEEE International Conference on Distributed Computing in Sensor Systems*, 2014, pp. 67–74.

[11] M. Uddin, M. Amin, H. Le, T. Abdelzaher, B. Szymanski, and T. Nguyen, "On diversifying source selection in social sensing," in *9th International Conference on Networked Sensing Systems (INSS)*, 2012.

[12] M. F. Goodchild, "Citizens as sensors: the world of volunteered geography," *GeoJournal*, vol. 69, no. 4, pp. 211–221, 2007.

[13] P. Giridhar, M. Amin, T. Abdelzaher, L. Kaplan, J. George, and R. Ganti, "Clarisense: Clarifying sensor anomalies using social network feeds," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2014 IEEE International Conference on*, March 2014, pp. 395–400.

[14] D. Wang, T. Abdelzaher, L. Kaplan, R. Ganti, S. Hu, and H. Liu, "Exploitation of physical constraints for reliable social sensing," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, Dec 2013, pp. 212–223.

[15] A. H. Lipkus, "A proof of the triangle inequality for the tanimoto distance," *Journal of Mathematical Chemistry*, vol. 26, pp. 263–265, 1999.

[16] S.-H. Cha, "Comprehensive survey on distance/similarity measures between probability density functions," *City*, vol. 1, no. 2, p. 1, 2007.

[17] "Introducing json," http://www.json.org.

[18] M. Shindler, A. Wong, and A. Meyerson, "Fast and accurate k-means for large datasets," in *Proceedings of Neural Information Processing Systems*, 2011, pp. 2375–2383.

[19] K. Clarkson, "Nearest-neighbor searching and metric space dimensions," in *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice*, G. Shakhnarovich, T. Darrell, and P. Indyk, Eds. MIT Press, 2006, pp. 15–59.

[20] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *Proceedings of the 23rd International Conference on Very Large Data Bases*, ser. VLDB '97, 1997, pp. 426–435.

[21] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[22] C. C. Aggarwal and C. K. Reddy, *Data Clustering: Algorithms and Applications*, 1st ed. Chapman & Hall/CRC, 2013.

[23] L. Cayton, "Fast nearest neighbor retrieval for bregman divergences," in *Proceedings of the 25th International Conference on Machine Learning*, ser. ICML '08, 2008, pp. 112–119.

[24] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[25] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, 1987, pp. 1–12.

[26] W. Zhao, H. Ma, and Q. He, "Parallel k-means clustering based on mapreduce," in *Proceedings of the 1st International Conference on Cloud Computing*, 2009, pp. 674–679.

[27] (2014, Sep) Green server farm. [Online]. Available: http://greendatacenters.web.engr.illinois.edu/

[28] (2015, Jan) Apache thrift. [Online]. Available: https://thrift.apache.org

[29] (2015, Jan) Memcached. [Online]. Available: http://memcached.org

[30] (2015, Jan) Apache hadoop. [Online]. Available: http://hadoop.apache.org

[31] (2014, Sep) Spark: Lightning Fast Cluster Computing. [Online]. Available: http://spark.apache.org/

[32] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[33] (2015, Jan) Apache mesos. [Online]. Available: http://mesos.apache.org

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Networked Systems Design and Implementation*, 2012.

[35] (2014, Oct) New Tweets per second record, and how! [Online]. Available: https://blog.twitter.com/2013/new-tweets-per-second-record-and-how

[36] X. Yu, Q. Fu, L. Zhang, W. Zhang, V. Li, and L. Guibas, "Cabsense: creating high-resolution urban pollution maps with taxi fleets," *ACM MobiSys, Taipei*, 2013.

[37] Y. Jiang, X. Xu, P. Terlecky, T. Abdelzaher, A. Bar-Noy, and R. Govindan, "Mediascope: Selective on-demand media retrieval from mobile devices," in *Proceedings of the 12th International Conference on Information Processing in Sensor Networks*, ser. IPSN '13, 2013, pp. 289–300.

[38] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable k-means++," *Proc. VLDB Endow.*, vol. 5, no. 7, pp. 622–633, Mar. 2012.

[39] D. R. Cutting, D. R. Karger, J. O. Pedersen, and J. W. Tukey, "Scatter/gather: A cluster-based approach to browsing large document collections," in *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1992, pp. 318–329.

[40] N. Ailon, R. Jaiswal, and C. Monteleoni, "Streaming k-means approximation," in *Proceedings of Neural Information Processing Systems*, 2009.

[41] A. Guerrieri and A. Montresor, "DS-means: Distributed data stream clustering," in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par'12, 2012, pp. 260–271.

[42] S. Bandyopadhyay, C. Giannella, U. Maulik, H. Kargupta, K. Liu, and S. Datta, "Clustering distributed data streams in peer-to-peer environments," *Inf. Sci.*, vol. 176, no. 14, pp. 1952–1985, July 2006.

[43] D. Pelleg and A. W. Moore, "X-means: Extending k-means with efficient estimation of the number of clusters," in *Proceedings of the Seventeenth International Conference on Machine Learning*, ser. ICML '00, 2000, pp. 727–734.

[44] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, 2008.

[45] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *International Conference on Management of data (SIGMOD)*, 2010.

[46] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, 1990.

[47] B. Shao, H. Wang, and Y. Li, "The trinity graph engine," Microsoft Research, Technical Report, 2012.

[48] (2014, Sep) Apache Storm. [Online]. Available: http://storm.apache.org/

[49] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013, pp. 385–398.

[50] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A real-time analytical data store," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014, pp. 157–168.