Performing Real-time Analytics with Apache Storm over Streaming Data from meetup.com

Shilpa Chaturvedi, *S. R. No. 12899* and Md. Imbesat Hassan Rizvi *S. R. No. 12214*

***Abstract* — We use the data from the streaming APIs of meetup.com, an online social networking platform facilitating offline group meetings, to generate meaningful real-time analytics about the platform and observe its user behavior. In particular, we define use cases and implement some of the approximation algorithms that focus on achieving operations such as sampling, filtering, counting and trending, without having to store huge amounts of data from the streams. Real-time analysis of streaming data is the focus, and the implementation is carried out with the help of Apache Storm platform. In addition to finding the approximation results for the defined use cases, we evaluate the results of each of these algorithms against their accurate counterparts and attempt to benchmark the error rate of the algorithms for the data considered.**

***Index Terms* — Real-time analytics, streaming data, evaluation of approximation algorithms.**

I. Introduction

MEETUP.COM [1] is an online social networking platform that facilitates offline group meetings at various venues around the world. Meetup allows members to find and join groups of their interest, ranging across politics, games, health, technology, careers, socializing or hobbies.

Organizing members can create public or private events within their Meetup groups and other members on meetup.com can browse, search for, and register to events based on their interest and locality. Meetup.com exposes useful streaming APIs that provide information about events, and RSVPs to these events by members. We use these streams in this experiment to generate useful analytics about the platform and observe user behavior on the platform. In addition, we implement some of the approximation algorithms centered on sampling, filtering, counting and popular elements, with the help of Apache Storm platform.

Apache Storm is a free and open source distributed real-time computation framework, written predominantly in Clojure. Storm makes it easy to reliably process unbounded streams of data, with many use cases including real-time analytics, online machine learning, continuous computation and more. Storm is fast and fault tolerant, and uses custom created "*spouts*" and "*bolts*" to define information sources and manipulations to allow distributed processing of streaming data [2].

A Storm application is designed as a "*topology*" in the shape of a directed acyclic graph with spouts and bolts acting as the vertices of the graph, and streams that direct data from one node to another can be seen as the edges. Together, the topology acts as a data transformation pipeline.

There are five streaming API endpoints of meetup.com. One exposes all the open events, another tracks RSVPs, another shares the photos shared from the events in real time, one more of venues being registered for open events and another for tracking online comments being made about specific open events. These APIs are exposed in long-polling, WebSockets and chunked HTTP formats. We consider the first two of these for our experiments, open events and RSVPs. Whenever there is any update to an open event's metadata, including their creation, deletion and update of venue, or change of description or timings, the open_events stream sends out the details about the event. Also, with respect to RSVPs, whenever an RSVP response is sent or its status changes, it is added to the stream. These two streams are the most active of the ones exposed, with over 200 RSVPs per second among events happening across the world.

In the upcoming sections, the details of implementing the sampling, filtering, counting, counting the number of 1s, and trending approximation algorithms are elaborated. Results specific to these approximation algorithms and their performance when compared to the accurate versions are tabulated. At the end, some interesting domain specific statistics of meetup.com and some scaling behavior are highlighted, along with the concluding remarks.

II. Sampling

One of the most obvious and important algorithms when it comes to approximation of real time streaming data is sampling. Extracting reliable samples from a stream becomes necessary in calculating various data points, as the actual data and the rate at which it is generated prohibits many exact calculations. In this experiment, we generate some sampling data from the meetup.com streams and evaluate their accuracy against the exact numbers.

A. Use Case

Open events data from the meetup.com stream contains quite a few additional information about the events being planned. Events are organized in many cities across the world at specific venues and they are classified based on different categories such as technical, socializing, health, and many more. Some of the events happen to be paid events where the interested

members will have to first pay a specified amount to register.

As part of the sampling use case, we chose to find out the exact count of open events happening across the world, and then generate samples at various rates to compare the overall accuracy of the sampling with respect to top cities hosting events, percentage of events in "tech" category, and percentage of paid events. Given that the actual open_events stream produces a lot of data, an approximate count of these sampled versions help in getting these important analytics without having to store all the data.

B. Implementation

Sampling is achieved on the open events data the way it is described in section 4.2.2 of Leskovec et. al.[3], with the help of random number generation and hashing. A 10 bucket hash is assumed and sampling data is obtained for 40%, 50% and 60% of all events, by generating random numbers and using 4, 5 and 6 buckets out of 10 respectively. The open events data is counted by applying multiple filters too, sampling per event city, and sampling based on whether they are paid events and whether they are "tech" events. Category filter is kept configurable so that we can get the counts of events of any desired category. For the experiments on sampling, I have considered 'tech' category.

C. Topology Details

Sampling implementation for the described use case involves a topology where we keep track of both the exact counts and the sampled counts. This helps us evaluate the accuracy of sampling. The spout reads from the open_events stream and passes on the important fields from each event, that include location, event id, whether the event is paid or if it is a tech event, to the bolts for consumption and report generation. In the sampling spout, care is taken to maintain a hash map and a random number generation mechanism that allows the generation of specified percentage of samples. An additional field sent out by the spout is an indicator of whether the tuple being sent is considered in the sample or not. Based on this information, the following counter bolt updates the appropriate counters.

The samples thus generated by the spout are consumed by the counter bolt, with field grouping based on the location. Counting is supported by maintaining a counter map per location, which is designed to store in a user defined data structure, different types of counts for a given event id and location – for both the exact and sampled cases. The counter bolt emits across the location and the counts, for the reporting purpose.

A dedicated report bolt takes care of reporting of the location based sampling data into an output file, along with the overall exact and sample counts.

D. Results

As part of proving the effectiveness of the sampling scenario, we conduct a few experiments with the data returned by the sampling algorithm. One is to check whether the order of top N cities in terms of popularity in conducting events changes much, and the other is to see if the percentages of different filters applied also sticks to the sampling percentages chosen. If the top N order is not affected much, and if the filtered data too is at the same sampling rates, it proves the true random behavior of sampling, and increases the confidence in choosing the samples for any analysis.

TABLE I
SHUFFLE IN THE ORDER OF TOP 20 CITIES FOR VARYING SAMPLING RATES

Accurate Data	40% Sampling	50% Sampling	60% Sampling
London	London	London	London
New York	New York	New York	New York
Los Angeles	Singapore	Los Angeles	Los Angeles
Toronto	Los Angeles	Singapore	Toronto
Singapore	Toronto	Sydney	Singapore
Chicago	Sydney	Toronto	Sydney
Sydney	Barcelona	Melbourne	Melbourne
Melbourne	Melbourne	San Francisco	Barcelona
Barcelona	Chicago	Barcelona	San Francisco
San Francisco	San Francisco	Denver	Chicago
Denver	Denver	Chicago	Denver
Washington	Washington	Seattle	Washington
Seattle	San Diego	Washington	San Diego
San Diego	Tokyo	Tokyo	Seattle
Tokyo	Seattle	Paris	Portland
Paris	Paris	Portland	Paris
Portland	Austin	San Diego	Tokyo
Austin	Portland	Austin	Austin
Hong Kong	Hong Kong	Hong Kong	Vancouver

In case of the top N listing, across sampling experiments, we consider the data to be sorted in descending order of number of unique events per city. For cities in the top 20 of such a listing, If we consider only sliding down from their original ranking as the action that leads to shuffling of orders (which would in turn push other countries up to take the place evacuated by the cities that are sliding down), we observe the following as shown in Table I:

For the 40% sample, the overall top 20 most active cities in terms of organizing events remain the same as those in the original data. Shuffle among them is created by 6 cities sliding down within the top 20.

Even for the 50% sample, the overall top 20 most active cities remain the same. Top 3 cities are exactly the same as in the accurate case. 5 cities slide down to create a shuffle.

As for the 60% sample, only one change is observed among the overall top 20 most active cities, and that too is at position 20. Top 5 cities are exactly the same as the accurate case, in the same order - London, New York, Los Angeles, Toronto and Singapore. 5 cities slide down within the top 20 leading to a shuffle in the order.

TABLE II
ACTUAL SAMPLING RATES FOR DIFFERENT GIVEN RATES

Actual Percentages of Data after Sampling	40% Sample	50% Sample	60% Sample
Overall Event Count	39.84%	50.03%	60.17%
Count of Paid Events	38.73%	49.26%	61.85%
Count of Tech Events	39.24%	49.10%	60.44%

In Table II, we see the data of actual percentages of data

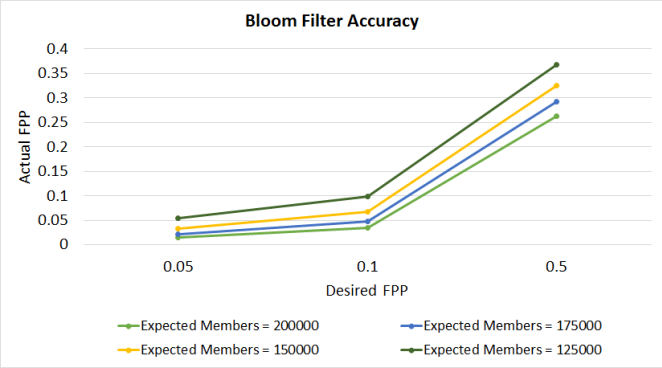


Fig. 1. A graph plot showing the desired false positive percentage (fpp) against the actual fpp, for each expected member count considered (125000, 150000, 175000 and 200000) and desired fpp rates (0.05, 0.1 and 0.5).

generated through the samples for the given sampling rates. The data is as close to the ideal as possible when run on events data of more than a week. Also, the sampling rates remain close to the overall sampling rates expected even in case of filters applied. That is, even the percentage of paid events and tect events show the same sampling rate as the overall sampling rate. This proves the equal distribution of the sampling data across all the incoming entries and no skew is observed, giving credibility to the sampled data.

III. Filtering: Bloom Filter Implementation

A Bloom filter [4] is a probabilistic data structure that can indicate quickly, in a memory-efficient manner, whether an element is present in a given set or not [5]. Bloom filter performs this activity, not by storing the actual elements of the set, but by storing large enough array of bits (a bit vector) corresponding to the entries of the set. Due to its probabilistic nature, a Bloom filter can tell us that the element either "definitely is not in the set" or "may be in the set". So, when the Bloom filter claims that an element may be in the set, it could be wrong, leading to false positives. No case of false negatives is possible though, which means, if the Bloom filter claims an element is not present, it is definitely not present in the set.

A. *Use case*

We take up a hypothetical scenario with respect to Meetup data, where we classify a set of members as "well-behaved members" based on their historical RSVP activities. We also presume that some of the questionable users conduct a burst of activities once in a while, where they send RSVP responses to mislead the genuine audience, or conduct a denial of service attack. In such a scenario, the stream of RSVP responses arriving would mostly contain activity from such users, interspersed with some genuine RSVP responses.

Meetup.com can address this issue by blocking out these questionable members. For this, they can maintain a Bloom filter of well-behaved users, and check the members from the incoming bursts of RSVP stream against this filter. Only those who are identified as being present in the filter can be let in, while others' IP can be blocked for a while. While this might still let through some of the misbehaving members, it is guaranteed that all the genuine users' responses are honored, and a majority of the bad users are stalled.

B. *Implementation*

As part of the filtering algorithm use case, we implemented this hypothetical scenario with Bloom filters, with multiple cases of *expected number of elements* and *desired false positive probability (fpp)*. We then evaluated the actual false positive rate in each case against the fully accurate behavior. Exact values for the fully accurate behavior (with zero false positives) are obtained by maintaining the data of well-behaved members also in a HashMap, and checking against them on the incoming stream. We highlight the inevitability of going for a probabilistic data structure like Bloom filter as the data size grows large, and show the efficiency of Bloom filter for this problem in terms of accuracy levels and storage space.

Apache Storm (version 0.9.4, which is being used all through the experiments on the local cluster, and version 0.9.3.2.2 on the Azure cluster) comes with an implementation of Bloom filter (from *Google Guava* libraries) [6]. But this implementation does not contain the support for serialization and deserialization. So, we have used a newer version of the Google Guava library (version 19.0) that also has serialization support. This becomes necessary as we create a Bloom filter of well-behaved members through one Storm topology and use it in another Storm topology that checks for non-members of the filter.

C. *Topology Details*

To train the Bloom filter (and the accurate HashMap) with the well-behaved members data, over 500,000 RSVPs from the corresponding stream of meetup.com were considered. It was identified that among them, there were nearly 250,000 unique members. We divided this stream into 3/5ths of the data to identify the good users, and the rest to check against the potential bad ones. That meant about 300,000 RSVP entries were used during the creation of the Bloom filter. Exactly 151,320 unique members were found. These were identified as the good users and were put into the Bloom filter as expected elements.

Among the remaining RSVPs (over 200,000 of them), nearly 100,000 unique users had not yet been encountered, and these would be considered as bad users for this use case. To ensure that we get significant number of repetitions from the good users list for our accuracy check, we also added about 10% of the RSVPs from the training phase (about 30,000 records) to this phase of streaming input.

The next step was to conduct the experiments by varying the expected number of members (n) and the desired false positive percentage (fpp), which can both be parameterized through the Bloom filter implementation of the Guava libraries. The library evaluates the ideal number bits required in the bit vector (m) as well as the number of hash functions needed (k) to stick to an actual fpp that is upper bounded by the desired fpp given. Hash functions are implemented using MurmurHash, which is both fast and independent. The actual false positive probability is

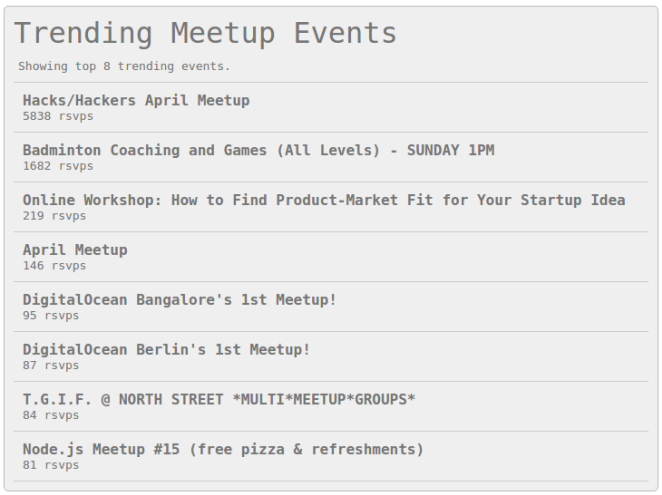


Fig. 2. A screenshot of the trending Meetup events visualization, with popularity calculated in terms of number of positive RSVPs.

obtained by $(1- e^{- kn / m})^{k}$.

The first topology, "*Acceptable Users Filter Preparation*", prepares the Bloom filter and the HashMap that helps in finding out accuracy later. The spout consumes the RSVP stream and emits member ids, and the corresponding Bolt stores the member ids in the Bloom filter as well as in the HashMap. At the end, as part of the cleanup, the Bolt also serializes and stores both the data structures back to the secondary memory.

The second topology, "*Check For Acceptable Users*", reads in the test RSVP stream, compares against the existing filters (deserialized Bloom filter and HashMap) and pronounces the presence or absence of incoming member ids. The overall counts from both data structures are shared by the Bolt so that the actual false positive percentage can be calculated.

D. Results

Experiments were conducted with the scenario stated in the previous section, where the number of well-behaved members were 151,320 from the training phase. Expected number of members were tested for the counts of 125,000, 150,000, 175,000 and 200,000. False positive percentage were also varied from 0.05 to 0.1 to 0.5 in each of the above cases. A plot of the desired fpp against the actual fpp, for each expected member count is shown in Fig. 1.

The plot clearly indicates that in most cases, the actual fpp is upper bounded by the desired fpp, and that the error rate decreases as the expected member count is increased. Also, if we decide to allow higher fpp, actual fpp values also deteriorate within the limit of the desired fpp. Memory allocation varies based on the size of the Bit Vector needed to satisfy the given parameters.

On observing the memory usage by the topology, it is clear that the Bloom filter version of the solution behaves far more efficiently than the HashMap version. For the experiments conducted above, while the HashMap size remained the same across all cases, size of the Bloom filter changed based on the expected number of elements and the desired fpp. These experiments indicate that the HashMap is at least 20 times larger than the Bloom filter implementation, while in the worst case, it is 125 times larger (23 K Bytes of the Bloom filter vs 2.8 M Bytes of the HashMap).

IV. Trending Events

A word, phrase or topic that is tagged at a greater rate than other tags is said to be a trending topic. Trending topics become popular either through a concerted effort by users or because of an event that prompts people to talk about one specific topic. These topics help Twitter, Facebook and other such social networks and their users to understand what is happening in the world.

A. Use case

Inspired by the Twitter trends widget, we have created a trending events algorithm based on the RSVPs stream of meetup.com. The system is designed in such a way that it considers Event A to be more popular than Event B (for a given time span) if Event A has received more positive RSVPs than Event B. Eventually, we want our topology to periodically produce the top N most trending events. The logic for trending is the implementation of a *sliding window* and ranking the events based on the aggregate count of event windows, and periodically advancing the window as well as emitting the trending events.

B. Implementation

The calculation of trending events is carried out by maintaining a sliding window data structure. A window similar to a queue is maintained for every event and data collected at some interval for each event is pushed to the queue. The sliding window provides total counts of the occurrences of "events" present in the current window, that is, a sliding window count for each tracked object. As the sliding window advances, the slice of its input data (the events tracked, in this case) changes. Mathematically, we can represent the sliding window count at any time instant t as

$$\text{Sum} = \sum_{i=t}^{i=t+m} \text{element}(i)$$

where m is the size of the window. At every N time units, the total count of the window for each event is returned as the event count and the window is advanced. We are using a modified version of the sliding window, where the count of the whole window is returned, not limited by any size.

For storing and visualizing the trending event details generated by the topology, Redis [7], an open source in-memory data structure store, is used along with a small web application written in Python. We use Redis to provide a *PubSub* queue and the topology publishes the trending events to that queue at regular intervals. The Python web application present at the subscriber end reads the trending events and displays it on the user interface, making the whole trending event visualization near real time.

C. Topology Details

The calculation of trending events involves a series of steps. The RSVPs are collected for a time interval at the counting stage and the event RSVP counts are emitted periodically to the

next level.

The *aggregator logic* will keep intercepting the counts and maintains a data structure similar to a sliding window, where for each event, a queue is maintained for storing the counts which are received from the count stage. This aggregation stage will emit the window counts periodically to the next ranking stage and windows for each event will be advanced after every emit. This means that the data collected for last F seconds is emitted where F is the emit frequency of aggregator.

The *global ranking stage* will keep collecting the event counts from previous stage for some time. This stage ranks the current events based on collected counts and emits on top N events periodically. The counters are reset to zero again at this stage. This makes sure that we get trending topic for every T seconds where T is the emit frequency of the ranker.

The final stage is the *reporting stage* which publishes the top N trending events to the subscribers for visualizations or storage. An important aspect to note is that the emitting intervals of counting, aggregation and ranking stages are different.

D. Results

A snapshot of the trending events visualization is provided as the result in Fig. 2. As can be observed, the trending Meetup events at a specific time instant are ranked based on the number of RSVPs the events received, and they move further up or down in the trend based on the regularity of RSVPs across time frames.

V. DGIM: Counting ones in a window

Counting ones, or in general, counting a specific type of elements in a window could be very helpful in many use cases, especially on streaming data. So, given a binary stream window of size N, the ability to answer the question "how many 1s are there in the last k bits?", for any $k \leq N$ is useful. Getting the exact count of this would require storing all the N bits of the window to ensure accuracy. However, there are approximation solutions like the *Datar – Gionis – Indyk - Motwani (DGIM)* algorithm [8] that can approximate this count by reducing the storage significantly. In fact, the DGIM algorithm can estimate the count with just having $O(\log^2 N)$ storage space for a window of N bits, with an error rate of no more than 50% [3].

A. Use case

Even here, the meetup.com stream of RSVPs is considered. The use case is of having a mechanism to count the number of "yes" RSVPs in any given time frame, or a window. Capturing this information regularly could help meetup.com analyze peak traffic periods in the day. This could in turn help them decide scaling of their systems for peak load and even device marketing strategies where the usage is low.

B. Implementation

For the implementation of the DGIM algorithm on RSVPs data, all "yes" responses are considered as 1s and every other type of response is assumed to be depicted with 0s. We are interested in counting only the 1s. As part of the implementation, we again take the approach of calculating the approximate count with the help of the DGIM algorithm and the corresponding accurate count in parallel, and observe the deviation in accuracy for different runs (by varying the window size N and the interval size k) of the topology.

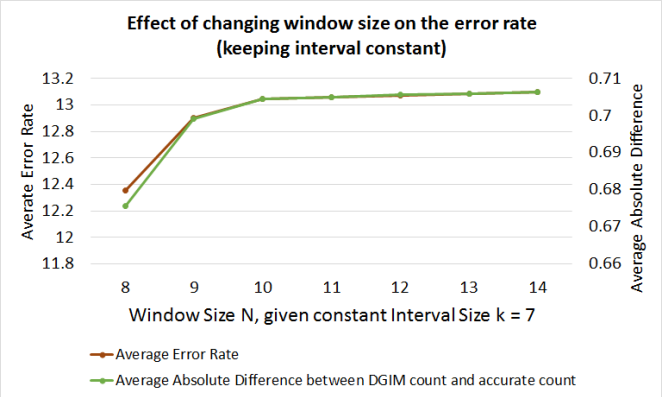


Fig. 4. Effect of changing the window size (N) on the average error rate, keeping the interval size (k) constant in DGIM algorithm. Here, k is kept at 7, and N is varied from 8 till 14.

All the logic related to the DGIM algorithm is coded into the *DGIM Container* class. The class exposes three methods – one to consume an incoming bit and update the internal data structures appropriately according to the DGIM algorithm, another to get the approximate count for the given interval and window sizes using the DGIM approach, and one more to fetch the accurate count of 1s for the same interval and window sizes.

A Bucket object is designed to keep two pieces of information – the timestamp of the most recent 1 seen in the bucket, and the number of 1s in the bucket (in base 2 logarithmic scale, as we only have bucket sizes in powers of 2). The DGIM container implements the Bucket Sequence using a Deque [9] (double ended queue) of Buckets. The choice of a Deque was obvious as there are use cases of both insertion and deletion from both ends of the Bucket Sequence, and there is a need to iterate both in an ascending and descending manner. All these are supported by the Deque data structure in Java.

The *consumeInputBit* method takes the input bit from the stream, puts it in the window-sized array maintained for generating accurate counts and performs the Bucket Sequence updates as required by the DGIM algorithm. This involves removing stale bucket if any, adding the bit to the Bucket Sequence if it is a 1, and combining buckets if required after this addition. The *getCount* method returns the approximated count of 1s for the given interval and window sizes, by querying the Bucket Sequence.

Programming languages like Java implement the % operator as the reminder operation instead of the modulus operation.

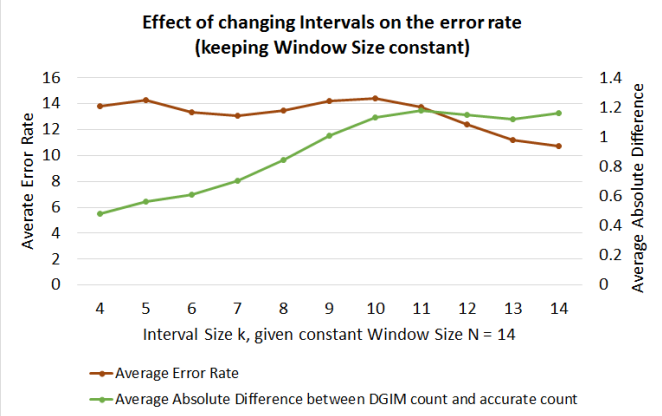


Fig. 3. Effect of changing the interval size (k) on the average error rate, keeping the window size (N) constant in DGIM algorithm. Here, N is kept at 14 and k is varied from 4 till 14.

That means, when there is a negative number whose modulus has to be found, the result will also be negative. But for the purpose of DGIM, we would just need the positive modulus (for example, we want -1 MOD 8 to be 7, not -1). This is taken care in the local logic, by building on top of the % operator.

C. *Topology Details*

The data structure required for the Bucket, as well as the logic involved in insertion and querying of the Bucket Sequence as per the DGIM algorithm, are both implemented through a container class. So, the responsibility of the Storm Topology is limited to bringing in the logic of implementing the identified use case with the help of the DGIM container. The topology also has to generate data that helps in analyzing the deviation from accurate counts.

While the spout takes the responsibility of reading the RSVPs stream and extracting the RSVP response field and converting it into 1s and 0s, the only bolt present in the topology takes care of generating useful data by using the DGIM algorithm. The bolt reads every tuple that come in through the spout, and calls the three exposed methods of the DGIM container. For each input bit, it inserts the bit into the Bucket Sequence and gets the approximate and accurate counts of 1s for the given window and interval sizes. This way, we can calculate the accuracy of the algorithm at the end of each input bit. The bolt outputs the data of timestamp, the DGIM count of 1s, and the accurate count of 1s to an external file.

D. *Results*

We evaluate the DGIM algorithm in two ways. First, the window size N is kept constant and the interval size k is changed around. In the second scenario, interval size k is kept constant for changing values of the window size N. In both the scenarios, we calculate the average variation in absolute difference between the approximate and accurate counts of 1s, and the average error rate. Error rate is calculated by finding the percentage error with respect to the corresponding accurate count. The experiments are carried out on nearly 100,000 RSVPs, generating these numbers as an average across those many tuples.

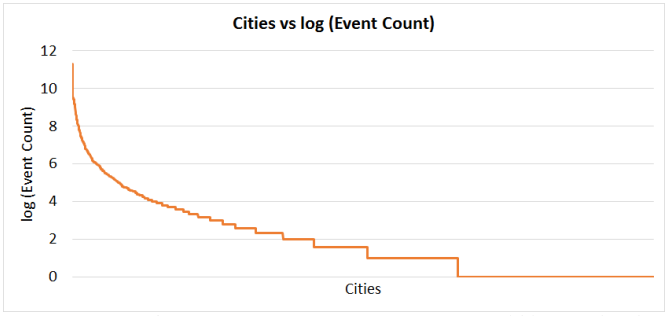


Fig. 5. Number of events happening across various cities within a week's data of meetup.com. Plotting is done by taking the base 2 logarithm of the event count.

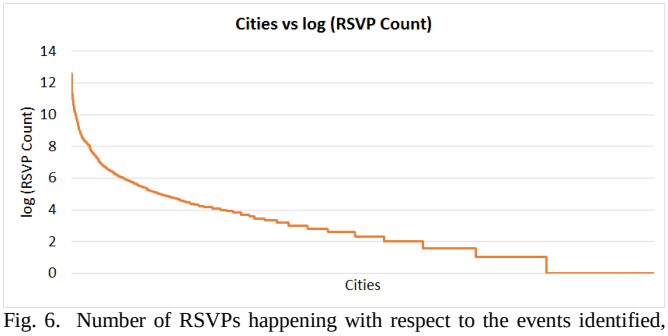


Fig. 6. Number of RSVPs happening with respect to the events identified, across various cities within a week's data of meetup.com. Plotting is done by taking the base 2 logarithm of the RSVP count.

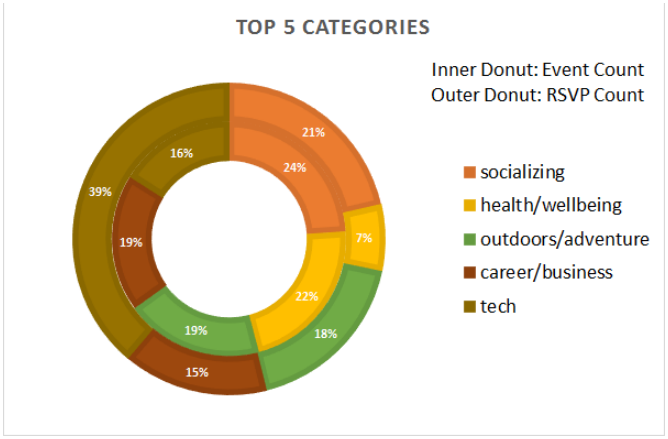


Fig. 7. Top 5 categories in terms of events created and RSVP responses sent to those events. Inner donut depicts the event count, and the outer one depicts the RSVP count.

For the experiments carried out, the average error rate stood between 11% to 14.5%. Plotting this data on a graph reveals expected behavior. In the case of keeping the window size constant (N = 14) and varying the interval size k, as shown in Fig. 3, the average absolute difference between the DGIM count and the corresponding accurate count increases as the interval size increases. Error rate, however, hovers around 14% and

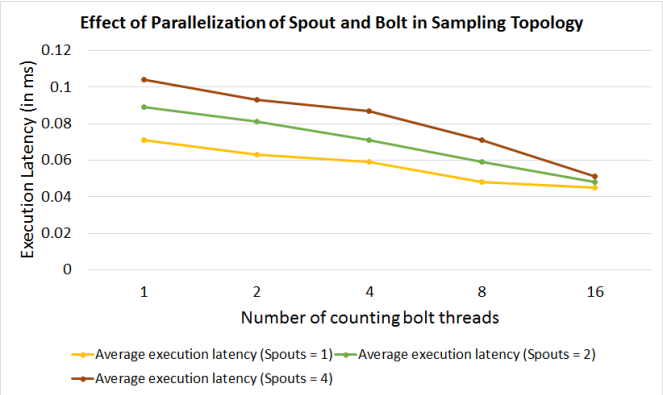


Fig. 8. A plot of average execution latency of the counting bolt for varying values of number of spouts (1, 2 and 4) and bolt threads (1, 2, 4, 8 and 16), indicating speedup / slowdown with respect to increase / decrease of resources.

decreases as the interval size approaches window size.

In the case where the interval size is kept constant (k = 7) and the window size is varied, a correlation can be observed between the average absolute difference and the average error rate. As can be seen in Fig. 4, both of them increase steeply when the window size is closer to the interval size, but settles for minor changes as the window size grows further away from the interval size.

VI. Counting Distinct Element

Meetup Streams comprise of several fields which are of interests to us. One particular interest may be to know how many distinctly different elements have appeared in the stream, counting either from the beginning of the stream or from some known time in the past. This nature of problem is referred to as "*Counting distinct elements in a stream*".

A. Use Case

The counting distinct elements in meetup streams may be used in scenarios where we would like to know the unique number of peoples attending events in a given location. In case there are several meetups happening in a location, same member / person may post rsvp for each of these events. Thus simply counting the attendees after filtering the streams based on location will overestimate the count. The location may be given as a country, or a city or a state etc. whatever be the target of our analysis. We have considered determing the unique number of persons who are attending events given a country, e.g. US.

B. Implementation

There are two approaches for the above mentioned problem. The exact and straightforward method which, after filtering based on location, saves member id to check for repetitions of meetup rsvps done by same member. The other approach which is approximate in nature but gives good accuracy still maintains less memory footprint is the Durand-Flajolet Algorithm. Both the approaches have been implemeted and compared.

C. Exact Distinct Counting Method

The exact distinct counting method is quite straight forward. Once the meetup rsvps are filtered, the bolts receiving the filtered output, maintains a HashSet of memeber ids witnessed so far. In execution function, the bolt checks whether the member id of incoming tuple is already present in the HashSet or not. In case it is not present, the counter which is maintaining unique count is incremented and the updated count is forwarded for further processing, e.g. writing to file. In case the member id is present, the tuple is ignored and the counter is not incremented.

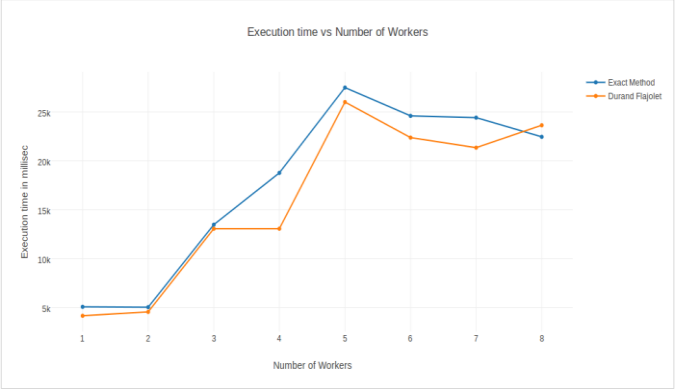
D. Durand-Flajolet distinct count Algorithm:

As good and simple as the exact counting method may be, it suffers from the drawback of excessive memory usage for storing all the previously witnessed member ids. For streams with very small number of distinct elements/members, this may not be a problem, but it is surely not expected to scale.

To deal with this, algorithms belonging to the family of Probabilistic Distinct Counting are well suited as they offer excellent estimate of the actual unique counts but comparatively use much less memory. It is assumed that the stream elements are chosen from some universal set.

The Durand-Flajolet Algorithm uses a hash function to turn a set of data into an evenly-distributed random values. A particular number of trailing bits of hashed value (e.g. 6) are used to divide the member ids into different buckets. Thereafter, the Durand-Flajolet uses the maximum number of trailing zeros in the remaining bits of hashed value to estimate the unique Count as *2^(sum(max_zeros) / num_buckets) * num_biuckets * Durand-Flajolet_Magic Number*. This unique count estimate is returned.

Fig 9. A plot comparing both counting strategies and shows deviation of Durand-Flajolet Algorithm counting w.r.t. exact.



E. Topology Details

The storm topology is designed as described below:

Stream generation Spouts – Spouts read stream data from hdfs files and emit tuples having fields like, stream generation timestamp, tuple id, country of event, city of event, member id, event name, event time etc.

Filter Bolts – The filter bolts takes input tuple and checks its value against filtering field, in this case country. In case of positive match it forwards the tuple to subsequent levels, otherwise drops the tuple.

Field Counter Bolts – The outputs from filter bolts are field grouped w.r.t member id into field counter bolts. Thus several

bolts will keep track of unique counts of distinct set of member ids. These counts are forwarded to a unique count bolt which receives count from all these field count bolts and returns/writes their sum as output or for display.

F. Results

The comparison of both the counting strategies is provided as the result in Fig. 9. As can be observed, the approximation counting is behaving similar to exact counting with some deviation. Also Fig 10 shows accuracy of Probabilistic estimate of distinct count by Durand-Flajolet Algorithm against the Exact Method implementation.

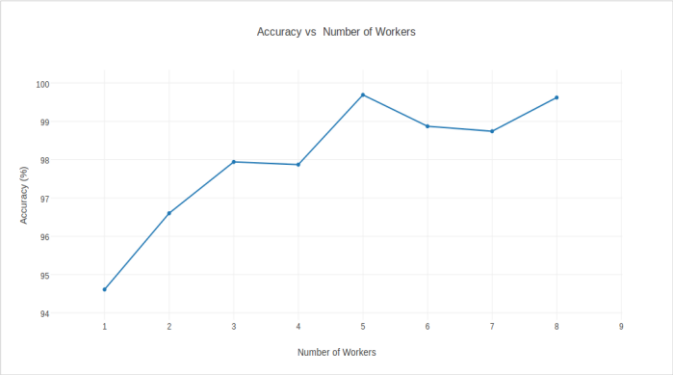


Fig 10. Accuracy of Probabilistic estimate of distinct count by Durand-Flajolet Algorithm against the Exact Method implementation.

VII. List of upcoming events in a given location:

One may be interested to know all the upcoming events in a given location w.r.t. a given date. This is a useful case when a person wants to browse events happening in a given location of his/her interest. This is a simple topology that provides filtering over open events stream data which simply requires two level filtering, one based on location and the other based on upcoming status. The implementation finally outputs only those upcoming events that are happening in a given location and on a given date.

VIII. Statistics, Observations and Results

In addition to the implementation of the well-known approximation algorithms on streaming data, we wanted to generate some domain specific analytics with respect to meetup.com data and observe any interesting patterns. Counting the events and event RSVPs with respect to host cities and categories had the potential to reveal insights. A separate set of topologies were created to get this information. Following are some of the observations from about one week's data of both the open events and RSVPs streams.

A. Domain Statistics

Although 3484 different cities have Meetup activity within the chosen time period, top 10 cities (0.29%) make up for as many as 18.2% of the events. In terms of the RSVPs to these events in the same time period, top 10 cities (0.35%) contribute 23.6% of RSVPs despite members from 2890 distinct cities showing interest in the events planned. This indicates the presence of a long tail in both these statistics, leading to a power law distribution. This is clear from Fig. 5 (number of events with respect to cities) and Fig. 6 (number of event RSVPs with respect to cities). The plots leave out the city names in the X-axis to avoid the clutter.

TABLE III
CATEGORY DIVERSITY ACROSS CITIES BASED ON EVENTS CREATED

Category	Percentage of cities with events of this category
health/wellbeing	27.01
career/business	26.69
socializing	24.60
outdoors/adventure	22.68
new age/spirituality	21.38

Event and RSVP counts for the 33 distinct categories are more evenly distributed than the distribution over cities. The proportion of RSVPs to events of a specific category are in direct correlation with the number events in that category. However, we observe a significant deviation in a couple of cases.

When we consider only the top 5 categories in terms of number of events, Fig. 7 shows the percentage of events (inner donut) and the percentage of RSVPs (outer donut) with respect to these categories. While the percentages are comparable for 3 of the top 5 categories, 22% of the events in "health / wellbeing" category only led to 7% RSVPs. Also, 16% "tech" category events led to a massive 39% RSVPs. Capturing this data for a longer period and analyzing could lead to interesting patterns in terms of demand and supply of events of specific categories.

In terms of the diversity, there are 5 categories that feature across more than 20% of the cities in terms of events conducted, as can be seen from Table III.

B. Scaling Observations

The topology created for the sampling use case is used to observe the execution behavior on parallelization.

Storm provides parallelism with the help of three entities - worker processes, executors and tasks. A worker process executes a subset of a topology, and runs in its own JVM [10]. A worker process may run one or more executors for one or more components (spouts or bolts) of this topology. A worker process spawns an executor as a thread that runs within the worker's JVM. An executor may run one or more tasks for the same component. A task performs the actual data processing and is run within its parent executor's thread of execution.

A running topology consists of many worker processes running on many machines within a Storm cluster, each with their own executors and tasks in turn. In this experiment, we are showing the effect of thread level parallelism. As we increase the number of executors of count bolts, the average execution latency decreases. This is because the load gets divided into more number of threads.

We can also increase the number of spouts if the data rate is very high. When we increase the number of spouts, we can handle more data from the stream for the same time period. This means bolts chained together in the topology getting fed from the spout will all get more data to handle when the number of

spouts are increased. This should increase the latency of the bolts.

When the bolt threads are increased, the incoming spout data can be handled in parallel, thus effectively reducing the latency at the bolt stages. Fig. 8 shows the behavior of the topology when the number of spouts and the counter bolt threads of the sampling topology are varied. The behavior establishes the theory described above, with the execution latency of the counting bolt increasing with the increase in the number of spouts, and decreasing in each case when more bolt executors were initiated.

IX. Conclusion

The objective of the work, which was to implement some of the well-known stream processing approximation algorithms including sampling, filtering, counting the number of 1s and trending, were implemented on specific use cases identified for the meetup.com data. In addition, for the experimental and benchmarking purposes, accurate versions were also implemented, which helped in each case, to evaluate the accuracy of the approximation algorithms.

Additional domain specific analytics were explored, but not many interesting conclusions could be drawn in the limited data available during the time frame of the execution of the work. There were some promising behaviors identified as described in the earlier sections, but more insights could be brought out by running the topology for longer periods and capturing more information about these data points. Further slicing and dicing could be carried out on this data on other facets not considered, like payment modes, duration of meetups, days of the week found more congenial for meetups, and so on.

Also useful for a future enhancement is to implement streaming machine learning algorithms on the meetup.com data. Well established algorithms like the streaming k-means could be used to group similar events together in a cluster and observe further statistics around it. Based on the kind of RSVPs being performed by the members and the type of events, streaming collaborative filtering algorithms can be implemented to come up with recommendations on members with similar interests and suggestions on which events to join.

X. Contributions

The project was the collaborative effort of *Shilpa Chaturvedi* (12899, CDS) and *Md. Imbesat Hassan Rizvi* (12214, CDS).
Shilpa has contributed to exploring various ways of domain analytics using ways like sampling, filtering, trending etc. She has been experimenting with lot of concepts like rolling counts, bloom filter and DGIM to propose her solutions as well as looking into how to make effective domain analytic and how her solutions scale in the given scenerio. The sections Shilpa contributed to are I, II, III, IV, V, VIII.
Md. Imbesat has focused more on implementing approximate counting and comparing the results with exact counting. He has also contributed a solution to list all the upcoming events at a particular location. He contributed to section VI, VII.

References

[1] Meetup.com: http://www.meetup.com/
[2] Apache Storm: http://storm.apache.org/
[3] Anand Rajaraman, Jure Leskovec, and Jeffrey D. Ullman, *Mining of Massive Datasets*, 2014, pp. 131–162.
[4] Burton H. Bloom, "Space/time trade-offs in hash coding with allowable errors". *Communications of the ACM, Volume 13 Issue 7, July 1970*, pp. 422–426. Available: https://dl.acm.org/citation.cfm?doid=362686.362692
[5] Bloom Filters by Example: http://billmill.org/bloomfilter-tutorial/
[6] BloomFilter class in Google Guava Libraries: http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/hash/BloomFilter.html
[7] Redis: http://redis.io/
[8] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows". *Proc. of 13th Annual ACM-SIAM Symp. on Discrete Algorithms*, January 2002. Available: http://www-cs-students.stanford.edu/~datar/papers/sicomp_streams.pdf
[9] Java Deque Interface for double ended queue: https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html
[10] Understanding the Parallelism of a Storm Topology: http://www.michael-noll.com/blog/2012/10/16/understanding-the-parallelism-of-a-storm-topology/
[11] Anand Rajaraman, Jure Leskovec, and Jeffrey D. Ullman, "Lecture notes of Mining of Massive Datasets - Mining Data Streams (Part 1)," Stanford University, 2014.
[12] Anand Rajaraman, Jure Leskovec, and Jeffrey D. Ullman, "Lecture notes of Mining of Massive Datasets - Mining Data Streams (Part 2)," Stanford University, 2014.
[13] Bloom Filters - The Math: http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html