Top-k Item Identification on Dynamic and Distributed Datasets

Alessio Guerrieri[1], Alberto Montresor[1], and Yannis Velegrakis[1]

University of Trento, via Sommarive 5, Trento (Italy)

Abstract. The problem of identifying the most frequent items across multiple datasets has received considerable attention over the last few years. When storage is a scarce resource, the topic is already a challenge; yet, its complexity may be further exacerbated not only by the many independent data sources, but also by the dynamism of the data, i.e., the fact that new items may appear and old ones disappear at any time. In this work, we provide a novel approach to the problem by using an existing gossip-based algorithm for identifying the k most frequent items over a distributed collection of datasets, in ways that deal with the dynamic nature of the data. The algorithm has been thoroughly analyzed through trace-based simulations and compared to state-of-the-art decentralized solutions, showing better precision at reduced communication overhead.

1 Introduction

One of the classical problems in computer science is the development of efficient algorithms to compute statistical functions over a dataset. Among these, identifying the most frequent items has attracted considerable attention over the last years. In particular, two challenging scenarios have been considered: very large but static datasets [12] and continuous streams of data [15].

Recent advances in information and communication technology have dramatically changed the computational landscape in which these problems are applied: useful information is now often found across many physically distributed and independent sources. For retrieving the most frequent items, one needs to collect and integrate information from multiple, dynamic datasets, posing challenges on the computation of a global function over the data located at distant nodes.

Computing the most frequent items over a collection of dynamically changing and independent data sets is part of the problem of *continuous distributed monitoring* [9]. This problem finds application in many different scenarios, such as computing the popularity of topics in social services like Twitter and Facebook, discovering global security attacks in communication networks, or identifying popular web pages for ranking search results.

The straightforward solution to this problem is to send all the information to a central node, which can in turn compute the statistics. As this approach is impractical for very large and dynamic datasets, a number of variations to this idea have been proposed aiming to reduce both the traffic and the load on

the central node. One approach is to perform periodic polling or more sophisticated random sampling [20] instead of continuous monitoring. Since the interest is only on the k most popular elements, one can send information only when the local set of top-k items changes, or when there is a number of changes above a threshold [11]. An intermediate solution tries to predict the interesting items and communicate them to the central node, using either knowledge of the data distribution [5], entropy statistics [3] or sketches as a form of a compact data representation [10]. This centralized approach can be applied when this information must be gathered in a single location to use it as a reference when needed.

However, using a central node may not be preferable in all applications. Individual sources may not be willing to send all their information to a central node and allow it to acquire a global view of the entire system that goes well beyond the original goal – identifying the most frequent items. If the number of sources is really large, the central node may become a serious bottleneck, not only in terms of communication but also in terms of computation. Finally, in a highly distributed environment, individual sources may need to have always available the information that the central node has computed and use it for their own purposes.

We advocate here a completely decentralized approach for computing the top-k most frequent items in a large collection of independent dynamically changing datasets, based on the idea of gossip-based protocols for information propagation [13]. Intuitively, each source has an estimate of what are the most frequent items globally. Initially, the only information a source has is the set of local frequencies. Periodically, each source performs a random gossip exchange with another source, sending and receiving their current estimates. Both sources then update their estimates using the old local estimate and the estimate received from the other source. This process is repeated until the estimates converge to the actual top-k items.

This idea has been recently applied to the identification of top-k items [19] in a collection of static datasets. The algorithm in [19] is shown to be very efficient, converging to the correct top-k items in a logarithmic time with respect to the size of the network. In this work we push this technique even further, by considering dynamic datasets where new items may be added – while existing items may be removed – at any time.

The contributions of this work are the following: (i) we formally define the problem of computing the top-k most frequent items in a distributed, dynamic environment (Section 2); (ii) we extend the algorithm presented in [19] by considering the case in which the collection of data is not fixed but varies over time (Section 3); (iii) we prove that our novel algorithm converges with very high probability despite the modifications to the original one (Section 4); (iv) we experimentally test our solution on trace-driven datasets, showing that, even without a central node, our approach manages to achieve a very good precision at the expense of a communication overhead which is shared among all sources (Section 5). We conclude the paper by analyzing related work (Section 6) and summarizing our results (Section 7).

2 Problem Statement

We consider a finite collection $\mathcal{P}$ of networked nodes. Each node can communicate – if it chooses to do so – with any other node in $\mathcal{P}$, provided they know its identifier; process identifiers may be obtained either through a static list, or through a peer sampling service [14]. We consider a universe $\mathcal{I}$ of items, a time domain $\mathcal{T}$ and a function $F : \mathcal{P} \times \mathcal{I} \times \mathcal{T} \to \mathbb{N}$, referred to as the *local frequency* of an item $i \in \mathcal{I}$ in a node $p \in \mathcal{P}$ at a time $t \in \mathcal{T}$, and denoted as $F_p^t(i)$ for brevity. Intuitively, the function represents the number of times an item has been observed in a node until a specific moment.

We define the *global frequency* of an item i at a time t, denoted as $F^t(i)$, to be the cumulative frequency in all the nodes, i.e.,

$$F^t(i) = \sum_{p \in \mathcal{P}} F_p^t(i)$$

We are interested in finding the k most frequent items across the whole node network. Let i_k^t denote the k-th item in the sequence of all the items in the node network sorted in decreasing order of global frequency at the time t. The set we are interested in is the set $MF^t \subseteq \mathcal{I}$ of the items with global frequency more than or equal to $F^t(i_k^t)$, i.e.,

$$MF^t = \{i \mid i \in \mathcal{I} \wedge F^t(i) \geq F^t(i_k^t)\}$$

Note that the cardinality of MF^t may be larger than k since there may be several items with the same global frequency as i_k^t.

We consider two different cases of the problem. In the first we assume that the frequency of the items can only increase in time. This finds application in scenarios where one is interested in the number of times the items have appeared in the nodes since the beginning of the operation of the system. We refer to this case as the *streaming scenario*. In the second case, we are interested in counting the appearances of items within a recent time window. This applies in scenarios where one needs to ignore appearances of items that have occurred long time ago and take into consideration only the recent appearances. This means that the function F for an item may increase or decrease in time. We refer to this case as the *sliding window scenario*.

3 Gossip-based top-k discovery

Since we assume no centralized authority or node with global knowledge, we would like every node of the network to be able to provide an answer to the top-k problem. Each node will estimate the *average global frequency*, i.e. the global frequency of an item divided by the network size; given that the network size is constant, this estimate can be used instead of global frequency to compute the top-k set.

We adopt a solution that is based on a gossip-based aggregation protocol [13], where the local knowledge of a node is expanded with knowledge collected from

Algorithm 1: Gossip algorithm executed by node p

```
Data: Nodes 𝒫, int k, int sleep, int s, int Δ_round
MAP est_p ← ∅
SET old ← ∅
int rounds ← 0

function main()
    repeat every Δ_round time units
        rounds ← rounds + 1
        if extractTop(est_p, k) ≠ old then
            rounds ← 0
            old ← extractTop(est_p, k)
        if rounds ≤ sleep then
            NODE q ← random(𝒫)
            send ⟨REQUEST, extractTop(est_p, s)⟩ to q

upon receive ⟨REQUEST, MAP est_q⟩ from q do
    MAP Δ ← ∅
    foreach i ∈ est_q do
        Δ[i] ← 1/2 (est_p[i] − est_q[i])
        est_p[i] ← est_p[i] − Δ[i]
    send ⟨REPLY, Δ⟩ to q

upon receive ⟨REPLY, MAP Δ⟩ do
    foreach i ∈ Δ do
        est_p[i] ← est_p[i] + Δ[i]

function modifyLocalFrequency(ITEM i, int δ_{F_p(i)})
    est_p[i] ← est_p[i] + δ_{F_p(i)}

function updateWindow(list activeItems)
    cutoff ← currentTime − windowSize
    while activeItems.head().timestamp < cutoff do
        modifyLocalFrequency(activeItems.head().timestamp, −1)
        activeItems.removeHead()
```

other nodes in the network. The nodes try to estimate the average global frequency of each item by updating any local estimate they may have to reflect also the estimates of the other nodes. If this is repeated continuously in a gossip fashion, then the information about the frequency of the most frequent items is epidemically propagated to all the nodes in the system. Previous work [19] has shown that not only this approach makes the frequencies of the various items in the individual nodes to converge to the true *average global frequencies* of the respective items, but also that they do so at an exponential rate [13].

The results of previous works [13,19] are based on the assumption that the local frequencies are not changing, i.e., that the input remain static while the

gossip algorithm is applied. We are interested in the case in which the local frequencies of the items are continuously changing, making the global frequencies continuously increase (in the case of the streaming scenario) or continuously fluctuate (in the case of the sliding window scenario). Our gossip-based algorithm is an extension of the one for the static case [19]. It propagates the changes that occur in a distributed fashion all over the network, using the parallel participation of the nodes to obtain a very good approximation of the average global frequencies.

The algorithm is shown in Algorithm 1. Each node p maintains a map structure $est_p : \mathcal{I} \rightarrow \mathbb{R}$ that represents p's estimate for the average global frequency of each item i, i.e., an estimate for the value $F^t(i)/|\mathcal{P}|$. Since $|\mathcal{P}|$ is constant, the top-k items in the map structure should coincide with the top-k among the estimated global frequencies. The node does not need to keep the local frequencies of the items in a different structure from the estimates. Whenever there is a change in the local frequency of an item, it is enough to record it in est_p by changing the estimate for the respective item accordingly. This is implemented by calling the function modifyLocalFrequency and providing to it the item and the change in its local frequency. Furthermore, we consider a function extractTop that given a map structure M and a number s, returns a new map structure with only the entries of M with a frequency in the top s values.

Each node p works in periodic *rounds*, during which it may initiate a gossip exchange with a random node q. A gossip exchange consists of a REQUEST message sent from p to q, followed by a REPLY message sent by q to p. In the request message, the node p includes the $s \geq k$ items from est_p with the s highest frequencies, alongside their estimated frequencies. Sending more than k items in the request results in faster convergence; this is a trade-off, however, as higher values of s result in larger communication costs.

When the request is received by node q, q updates its own estimates by subtracting $\Delta[i] = \frac{1}{2}(est_q[i] - est_p[i])$ from the estimate $est_q[i]$ of every item i that the received message contained. It then responds to the request by sending a reply message to p containing a map with the value $\Delta[i]$ of every item i whose estimate frequency was modified. Upon receipt of the response from q, for every item i for which the value $\Delta[i]$ is contained in the response message, the value est_p is updated to the value $est_p + \Delta[i]$. As a result, when the gossip exchange between the two nodes is completed, both nodes will have their estimates for the top-k items of p, updated to the average of the values that these two nodes had before the gossip.

$$est_q[i] \leftarrow est_q[i] - \Delta[i] = \frac{1}{2}(est_q[i] + est_p[i])$$
$$est_p[i] \leftarrow est_p[i] + \Delta[i] = \frac{1}{2}(est_q[i] + est_p[i])$$

In other words, a gossip exchange between any two nodes p and q substitutes the old values $est_p[i]$ and $est_q[i]$ with their average $\frac{1}{2}(est_q[i] + est_p[i])$.

Since it is possible that the global top-k items remain unchanged for potentially long periods, our algorithm communicates only when nodes observe varia-

tions in their current top-k lists, thus using fewer messages and bandwidth. We allow our nodes to be in one of two different states: *active* or *dormant*. Active nodes periodically initiate gossip exchanges with other nodes. Dormants only participate in exchanges initiated by other nodes. An active node becomes dormant when the last *sleep* number of exchanges have not changed its set of top-k items. A dormant node becomes active again whenever a variation in the set of top-k items occurs, either because of information received from other nodes, or because of variations in the local frequencies. The above approach ensures that the number of exchanges is reduced whenever there are no important changes, but can automatically and rapidly increase when needed.

For the case of a sliding window scenario each node keeps in a list the sequence of items it has received. When the topmost item in the list is out of the window, it is removed from the queue and its frequency in the local frequency table is decreased by 1. The sum of the local frequencies for that item is thus kept equal to the number of its active instances in the network. If this approach requires too much memory, we can divide the window into smaller time chunks and keep, for each of these chunks, the frequencies of all items the node has received in that time chunk. The window will not move continuously, but in chunk-steps: each time a chunk has become obsolete all its contents will be thrown away. In our experiments we assume that each node has enough memory to store the sequence of items it has received during the window and will update the local frequency table every time an item has become obsolete.

4 Protocol convergence analysis

Previous work [19] has computed a probabilistic upper bound on the number of rounds in the static case, showing that the convergence time grows logarithmically with the network size. If we assume that the local frequencies of the items do not change, then our problem is reduced to the case of [19].

In our case, the presence of the *sleep* parameter plays an important role. When the top-k of a node has not changed for *sleep* consecutive rounds, it will become dormant and will stop initiating exchanges until either it meets a node with a different top-k set, or its local top-k changes because of the arrival of new local data. The introduction of the *sleep* parameter creates the possibility, however low, that part of the network might converge to a wrong answer.

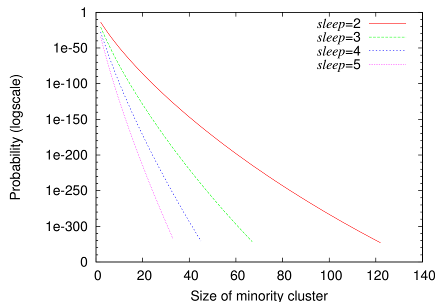


Fig. 1: Probability of convergence to a wrong answer, with different values of *sleep* ($n = 1000$).

To study the probability of such a situation, we devised the following scenario. Let $\mathcal{C}$ be the set of nodes containing a wrong top-k; furthermore, consider the

case where all top-k sets maintained by nodes in $\mathcal{C}$ are equal. Let $n = |\mathcal{P}|$ and $c = |\mathcal{C}|$. If, for *sleep* rounds of the protocol, nodes only contact nodes of their kind (nodes in nodes in $\mathcal{C}$ only contact those in $\mathcal{C}$, and those in $\mathcal{P} - \mathcal{C}$ only contact those in $\mathcal{P} - \mathcal{C}$), the entire network might become dormant before a common answer is reached. The probability of this event to occur in a complete graph is the following:

$$\left(\frac{c-1}{n-1}\right)^{c \cdot sleep} \cdot \left(\frac{n-c-1}{n-1}\right)^{(n-c) \cdot sleep}$$

As shown in Figure 1, the probability of the network becoming dormant while a group of nodes still contain a wrong answer get exponentially small with the size of the disagreeing group. Since nodes are also prone to exit from their dormant state whenever the arrival of new data changes the composition of the local top-k, we can conclude that the network will converge to the correct top-k with very high probability.

5 Results

We performed extensive simulations of our algorithm using PeerSim [18], a peer-to-peer simulator written in Java. If not stated otherwise, each experiment is repeated 20 times.

Our objective is to design a protocol that identifies the items in MF^t as accurately as possible. Unfortunately, it is impossible to guarantee that the output of our protocol corresponds exactly to MF^t at each time t, because of the delay occurring between the arrival of an item and the discovery of this fact by all nodes in the network. We will therefore measure the quality of a proposed protocol by checking for each node in the network, at each time t, the number of items that appear both in its output and in MF^t. We then compute the average across the entire network and, when needed, average across all time instants to get the average precision of the network across the entire experiment.

Evaluation framework We tested the algorithm on two different scenarios. The WCUP dataset contains timestamped URL requests to the 1998 World Cup servers across 90 days, covering a timeframe starting from a month before the first match to a few days after the final [4]. The LAST.FM dataset records the playing history of users across an entire year on the Last.fm website, a music discovery service that provides personalized recommendations based on the listening habits [1]. Our protocol computes the top-k most accessed pages in WCUP and the top-k most listened artists in LAST.FM. Each single data item is delivered to a node chosen uniformly at random. Different policies have been studied, without any impact on the quality of the solution.

The distributions of the frequencies of our chosen datasets follow a power law, the few top ranked items having very large frequency while all the many lower ranked items have very small frequency. This property guarantees a certain degree of separation between the top-k items and all the lesser frequent items.

Table 1 contains all the default parameters for the experiments listed in the current section. In our experiments, the d nodes that form the neighbor set of each node are chosen uniformly at random, property that could be achieved by using a peer sampling protocol. The amount of data items s sent per round is set to $2k$. Such value has been experimentally valitated in a previous paper [19] as a good compromise between convergence speed and bandwidth. Larger values for s do not induce a very large improvement in convergence speed (and thus precision), but have a much steeper cost in terms of message size.

N	number of nodes in the network	100
d	degree of nodes in the network	20
k	number of most frequent items	40
s	amount of data items per round per node	2k
Δ_{round}	length of each round	1 hour
sleep	sleeping factor	5
W	size of the sliding window	1 day

Table 1: Default values of our parameters, where not explicitly stated otherwise

Streaming results We first analyze how does our algorithm behave in the streaming model, when it has to compute the top-k over all the items that have arrived since the start of the experiment. To measure precision, in each instant t we compare the top-k of each node in that instant against the global top-k computed using all data delivered from instant 0 to instant t.

In Figure 2 we show the precision of our algorithm against the size of the network, using a round length of one hour. The larger the system, the more time is needed for information to reach all nodes; consequently, a slightly lower precision is obtained. Still, since an increase from 100 to 1000 nodes causes a decrease in just 0.5% in precision, the algorithm remains highly scalable.

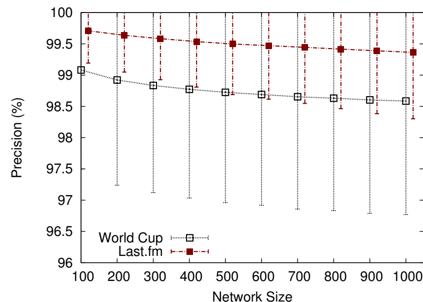
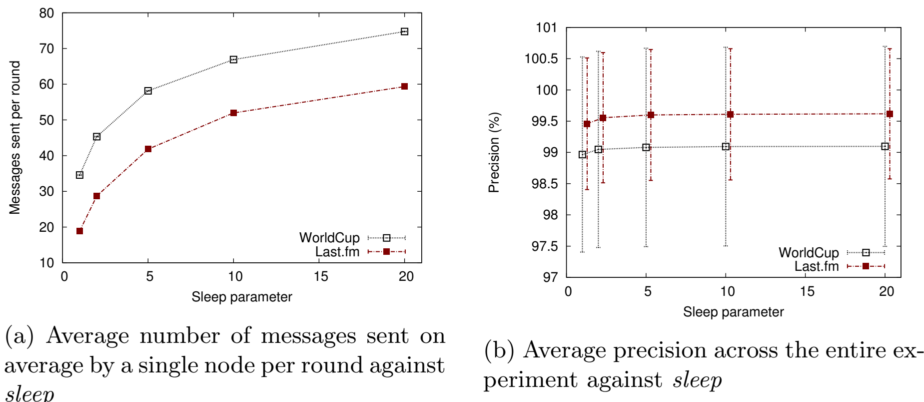


Fig. 2: Algorithm precision using variable network size

The *sleep* parameter is very influential in decreasing both the amount of messages and the workload of each node. Figure 3a shows that a small value for *sleep* can decrease the amount of messages sent by a huge margin across the entire experiment. Figure 3b instead shows that the highest the value for *sleep*, the slower the nodes will become dormant. If *sleep* is too low, the nodes will quickly become dormant and the algorithm will be slower to react to changes to the global top-k. By choosing the value for this parameter it is possible to achieve the desired trade-off between precision and bandwidth.



(a) Average number of messages sent on average by a single node per round against *sleep*

(b) Average precision across the entire experiment against *sleep*

Fig. 3: Analysis of the *sleep* parameter in the two datasets.

Sliding window results This second group of experiments illustrate the performance and behavior of our approach in a sliding window scenario, when each occurrence of a data item is deleted after W rounds have passed. We assume that each node has enough memory to store all the local items that are still within the time window and updates the local frequency table whenever one local item expires.

Figure 4 shows how the algorithm behaves with a sliding window 1-day long. By decreasing the round length of the protocol we can achieve almost perfect precision while using low bandwidth. Since each node will send around 1KB of data during each round, even with a round length of one minute the amount of bandwidth used is extremely small.

Real world scenario Since the WCUP dataset also contains the identification number of the server that served each page request, we can test our algorithm in a real world scenario by simulating the network of 20 servers that managed the web site during the 1998 World Cup. Figure 5 shows the precision of our algorithm when replicating the exact same setting, with a window length of 1 day and k equal to 40. Again, with a small round length the algorithm achieves almost perfect precision.

Comparison To our knowledge, there are no other decentralized top-k algorithm that work on sliding windows. We therefore compare our approach with another decentralized top-k algorithm in the basic, streaming scenario.

In Figure 6 we compare our approach with the gossip-based decentralized sampling approach in [16]. Since both approaches use gossip, it is possible to directly compare their performance by using the same round length and measuring the amount of bandwidth used. Figure 6 shows that our approach obtains better results using only a very small fraction of the bandwidth. When compared on the LAST.FM dataset, the larger difference is caused by the larger dataset.

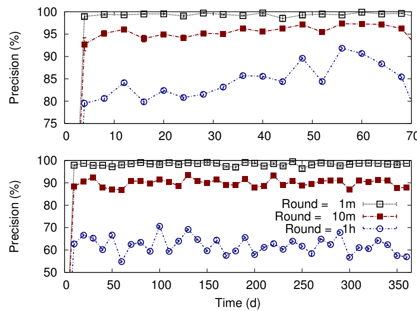


Fig. 4: Precision across time, using different window sizes (WCUP on top, LAST.FM on the bottom)

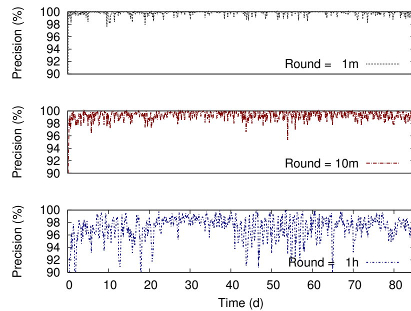
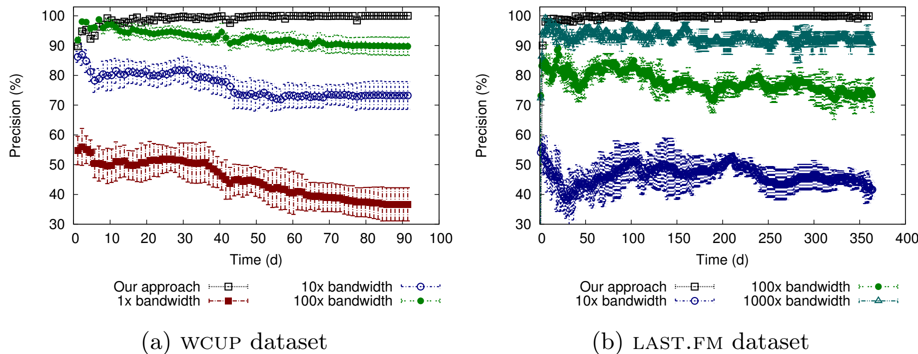


Fig. 5: Precision in the realistic WCUP scenario with $k = 40$, using different round lengths.



(a) WCUP dataset

(b) LAST.FM dataset

Fig. 6: Comparison of our algorithm with the gossiping sampling approach

6 Related work

Finding the most frequent items is a classic problem with applications in many different fields. According to the specific application and the properties of the dataset, wildly different requirements need to be satisfied. In the most basic case, where the dataset is stored in a single machine and the amount of memory available is enough to store the frequencies of each item, the problem becomes quite trivial. If the amount of memory is not large enough, the problem moves into the streaming scenario. Theoretical work [2] proved that it is possible to estimate items frequency to a constant factor in logarithmic space. The most common approach is to define compressed data structures to store the approximated frequency of the interesting items, the items that may be part of the top-k set. Among the many synopsis in the literature [12], the "COUNT SKETCH" data structure [7] allows a single pass algorithm that is able to compute an arbitrarily close approximation of the top-k in logarithmic space. Other algorithms [8] can also work in a sliding window scenario, by keeping track of both frequent items

and items that might become frequent in the future, with different degrees of precision. These algorithm cannot be directly applied when the dataset itself is distributed across different machines.

A common approach to the distributed computing of top-k sets is having a number of *slave* nodes that analyze their data and a *master* node that collects the partial findings and computes the final solution. The main drawbacks of this approach are clear: the system has a single point of failure and may cause an excessive amount of computation on the master node. Cao and Wang's algorithm [6] is an example of this type of solutions. Each slave computes its own top-k, all of which are collected by the master node to compute a lower bound on the frequency of the k-most frequent item. This information is given to the slaves, that recompute their solution to include only those items that have local frequencies above the threshold. Babcock and Olston's approach [5] instead computes a starting approximation of the top-k set in each slave node and in the master node. The temporary solution is then sent back to each slave, that starts analyzing the entirety of its data as it arrives. When a slave sees that its own solution is "different enough" from the global solution, it sends an update to the master node. It will be the master node's job to then notify all slave nodes if the global solution has changed.

One possible approach to avoid putting too much stress on the master node is using an hierarchical structure. There still is a root node that computes the final solution, but the costs of aggregation of temporary solution are spread between all the inner nodes of the topology. The construction and maintenance of the topology creates additional overhead on the system. Manjhi presents an interesting algorithm [17] based on compressed synopsis. This data structure offers an approximation of the frequencies of a datasets. Synopsises can be joined together at the different level of the hierarchical topology to obtain in the root node an estimate of the top-k set. This simple approach is then enhanced by the idea of a precision gradient. The level of compression of the synopsis is not kept constant in the system, but is adapted at each different level of the topology to minimize the communication costs.

A completely decentralized algorithm is inherently more robust and should guarantee better subdivision of work between the nodes. Lahiri and Tirthapura [16] presented a gossip algorithm based on uniform random sampling. The intuition behind this algorithm is that the top-k of a dataset should be similar to a large enough random sampling of the dataset. The algorithm thus computes a random sampling of all the data in the distributed system via repeated aggregation. Since the entire sample must be sent around, the amount of data sent is much bigger than in our algorithm.

7 Conclusions

In this work we have extended an existing approach to find the k most frequent items across a distributed collection of datasets, without relying on a central node that collects global knowledge about the data. The method we discussed is based

on a gossip protocol that allows local information in a node to be epidemically propagated to other sources. The algorithm presented has special features to deal with continuously changing data. Trace driven experiments illustrate that despite the dynamic changes in the global frequencies, the system is able to react quickly and provide a good approximation from any node of the network.

References

1. Last.fm. *http://www.lastfm.com*.
2. N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. of STOC'96*, pages 20–29. ACM, 1996.
3. C. Arackaparambil, J. Brody, and A. Chakrabarti. Functional monitoring without monotonicity. In *ICALP (1)*, pages 95–106, 2009.
4. M. Arlitt and T. Jin. 1998 World Cup web site access logs, Aug. 1998. Available at `http://www.acm.org/sigcomm/ITA/`.
5. B. Babcock and C. Olston. Distributed top-k monitoring. In *Proc. of SIGMOD'03*, pages 28–39, 2003.
6. P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *Proc. of PODC'04*, pages 206–215. ACM, 2004.
7. M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.
8. Y. Chi, H. Wang, P. Yu, and R. Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *Proc. of ICDM'04*. IEEE, 2004.
9. G. Cormode. Continuous distributed monitoring: a short survey. In *Proc. of AlMoDEP'11*, pages 1–10. ACM, 2011.
10. G. Cormode and M. N. Garofalakis. Sketching probabilistic data streams. In *Proc. of SIGMOD'07*, pages 281–292, 2007.
11. G. Cormode, S. Muthukrishnan, and K. Yi. Algorithms for distributed functional monitoring. *ACM Transactions on Algorithms*, 7(2):21, 2011.
12. P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. In *External memory algorithms*, pages 39–70. American Mathematical Society, 1999.
13. M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM TOCS*, 23(3):219–252, 2005.
14. M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM TOCS*, 25(3), Aug. 2007.
15. R. Karp, S. Shenker, and C. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28(1):51–55, Mar. 2003.
16. B. Lahiri and S. Tirthapura. Identifying frequent items in a network using gossip. *J. Parallel Distrib. Computing*, 70(12):1241–1253, Dec. 2010.
17. A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (recently) frequent items in distributed data streams. In *Proc. of ICDE'05*. IEEE, 2005.
18. A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *Proc. of P2P'09*, pages 99–100, Sept. 2009.
19. J. Sacha and A. Montresor. Identifying frequent items in distributed data sets. *Computing*, 95(4):289–307, 2013.
20. S. Tirthapura and D. P. Woodruff. Optimal random sampling from distributed streams revisited. In *Proc. of DISC'11*, pages 283–297, 2011.