The VLDB Journal (2015) 24:849–866
DOI 10.1007/s00778-015-0404-3

REGULAR PAPER

S^3-TM: scalable streaming short text matching

Fuat Basık[1] · **Buğra Gedik**[1] · **Hakan Ferhatosmanoğlu**[1] · **Mert Emin Kalender**[1]

Received: 2 October 2014 / Revised: 14 August 2015 / Accepted: 15 September 2015 / Published online: 24 September 2015

Abstract Micro-blogging services have become major venues for information creation, as well as channels of information dissemination. Accordingly, monitoring them for relevant information is a critical capability. This is typically achieved by registering content-based subscriptions with the micro-blogging service. Such subscriptions are long-running queries that are evaluated against the stream of posts. Given the popularity and scale of micro-blogging services like Twitter and Weibo, building a scalable infrastructure to evaluate these subscriptions is a challenge. To address this challenge, we present the S^3-TM system for streaming short text matching. S^3-TM is organized as a stream processing application, in the form of a data parallel flow graph designed to be run on a data center environment. It takes advantage of the structure of the publications (posts) and subscriptions to perform the matching in a scalable manner, without broadcasting publications or subscriptions to all of the matcher instances. The basic design of S^3-TM uses a scoped multicast for publications and scoped anycast for subscriptions. To further improve throughput, we introduce publication routing algorithms that aim at minimizing the scope of the multicasts. First set of algorithms we develop are based on partitioning the word co-occurrence frequency graph, with the aim of routing posts that include commonly co-occurring words to a small set of matchers. While effective, these algorithms fell short in balancing the load. To address this, we develop the SALB algorithm, which provides better load balance by modeling the load more accurately using the word-to-post bipartite graph. We also develop a subscription placement algorithm, called LASP, to group together similar subscriptions, in order to minimize the subscription matching cost. Furthermore, to achieve good scalability for increasing number of nodes, we introduce techniques to handle workload skew. Finally, we introduce load shedding techniques for handling unexpected load spikes with small impact on the accuracy. Our experimental results show that S^3-TM is scalable. Furthermore, the SALB algorithm provides more than 2.5× throughput compared to the baseline multicast and outperforms the graph partitioning-based approaches.

Keywords Short text matching · Stream processing · Publish/subscribe

1 Introduction

Micro-blogging has enjoyed wide adoption among Internet users and became a popular form of communication. Services like Twitter and Weibo enable users to create and share short updates to the public or to a selected group of contacts. Micro-blog posts, known as tweets, are up to 140 characters in length and short in comparison with regular blog posts. Users of these services can subscribe to the posts of other users, which is known as following a user. The content of a post is irrelevant to the subscription event and that means a user receives all the posts from the users it follows, no matter what the content is. In this respect, micro-blogging services resemble the traditional topic-based publish/subscribe (pub/sub) systems [7], in which tweets correspond to publications and user ids are analogous to topics.

Micro-blogging services also provide APIs for subscribing to streams of posts, where the matching is based on the content. For instance, Twitter has a *Streaming API* [27], which takes subscriptions in the form of a set of words and delivers matching tweets in a streaming manner. This model

✉ Fuat Basık
fuat.basik@bilkent.edu.tr

[1] Computer Engineering Department, Bilkent University, Ankara, Turkey

of service resembles the content-based pub/sub systems [7]. However, the backbone for this kind of service is typically implemented within a data center [2], and not using brokers over a wide-area network as in pub/sub systems [1,4,8]. Considering that the popular micro-blogging services receive hundreds of millions of posts per day, implementing this matching in a scalable manner is a key requirement. In this work, we present S^3-TM–a stream processing-based solution to scalable short text matching under the content-based subscription model. We develop effective techniques and algorithms for publication routing and subscription placement, which yield an overall scalable solution.

While current services are typically targeted toward a user-centric flow of information, S^3-TM provides the ability to filter messages based on their content. An example usage scenario would be subscribing to all micro-blog posts that contain the words *white* and *house* together, rather than following the official *White House* micro-blog account. This model can capture a broader range of relevant information, with less effort on the part of the subscriber.

S^3-TM is organized as a stream processing application in the form of a data parallel flow graph designed to be run on a data center environment. The system aims at parallelizing the task of matching publications against the subscriptions. For this purpose, it creates multiple instances of the matcher module and performs smart routing to avoid broadcasting publications or subscriptions to the matchers, so that scalability can be achieved as the number of replicas is increased in response to increasing volume of publications.

There are a number of challenges faced by S^3-TM.

1.1 Publication routing

The core issue in achieving scalability for streaming short text matching within a data center environment is the routing of publications and placement of subscriptions to the machines where the matching is to be performed. Previous attempts at this have been limited to publication unicast—subscription broadcast, publication broadcast—subscription unicast, or a combination of these two fundamental approaches [2]. However, in order to achieve good scalability as the workload (and thus the number of machines) increases, we need to avoid any kind of broadcast. To address this challenge, we take advantage of the problem domain. In particular, the word-based publications and subscriptions in micro-blogging enable us to apply hashing to multicast (as opposed to broadcast) publications to the machines responsible for matching the words they contain. This way, subscriptions can be placed on any one of the machines that are responsible for one of the words forming the subscription. However, this brings an additional challenge, which is to minimize the number of machines a publication is multicast to, which we refer to as the *spread*. To address this challenge, we develop effective word partitioning algorithms (which replace the hashing-based partitioning) that keep the spread low.

1.2 Load balancing

Another major obstacle to scalability is load imbalance. At one extreme, one way to minimize spread is to assign all words to a single machine. Obviously, this is the worst case scenario for load balance. In general, there is a trade-off between reduced spread and better load balance. To address this challenge, we integrate load awareness into our word partitioning algorithms. We develop several graph partitioning-based solutions that work on the co-occurrence frequency graph of words, where vertex and edge weights are used to create balanced partitions (words to be assigned to machines). However, graph partitioning- based approaches fell short, as they cannot accurately represent the load of a partition as the sum of edge or vertex weights. Therefore, we develop the SALB algorithm, which works on the word-to-post bipartite graph, rather than the word co-occurrence graph. SALB incorporates mechanisms to create a spread-aware load-balanced word partitioning.

1.3 Subscription placement and matching

The word partitioning-based routing leaves open the problem of placing subscriptions to machines, as a subscription can be placed on any one of the machines that is responsible for at least one of the words in it. Furthermore, given a number of subscriptions assigned to a machine, publications need to be matched efficiently against them. To solve the subscription placement problem, we first model the load imposed on a machine for handling the subscriptions placed on it, using a trie-based subscription matching technique. We then use this model to develop a placement algorithm that attempts to minimize the load, while at the same time keeping the load imbalance under control. Importantly, the subscription placement algorithm is incremental by nature, making it easy to admit streaming subscriptions.

1.4 Skew handling

While the SALB algorithm we introduce strives to balance the load, as the number of machines keeps increasing, the skew in the word frequencies starts inhibiting scalability. For instance, when the load due to a particular hot word exceeds the average load on a machine (average load reduces as the number of machines increases), it becomes increasingly difficult to achieve good load balance. We solve this problem by detecting hot words and applying a word splitting mechanism, which is adaptive to the number of machines, to break the hot words apart.

1.5 Overload and load shedding

Finally, under unexpected spikes in load, such as during rare events causing significant increase in post traffic, the streaming text matching service can experience overload. To address this, we develop simple yet effective techniques to limit the load, with little impact on the matching accuracy. We achieve this by putting a hard limit on the spread and selectively multicasting posts based on the expected value of their words in terms of the matching accuracy and the amount of load shed.

We evaluate S^3-TM through an extensive experimental study using real-world datasets. Our evaluation showcases the system's scalability, as well as the effectiveness of our publication routing and subscription placement algorithms. We provide insights about the behavior of the system at different scales, under different kinds of subscription workloads, and for changing publication contents (concept drift). Our results show that the SALB algorithm is the most effective among all and can increase throughput by a factor of 2.5× compared to a baseline multicast approach.

In summary, we make the following contributions:

- We present the S^3-TM system for scalable streaming short text matching, which relies on a distributed stream processing architecture to run at scale in a data center environment.
- We present algorithms for smart publication routing, including variants based on partitioning of the word co-occurrence graph and a novel algorithm called SALB that uses the word-to-post bipartite graph to perform spread-aware load-balanced word partitioning.
- We develop a subscription placement algorithm, called LASP, that takes into account the trie-based matching to minimize load, while at the same time preserving load balance.
- We develop simple yet effective techniques to handle skew in the publication workload, as well as load shedding techniques to handle overload situations.

2 Architecture

In this section, we present the general architecture of the S^3-TM system, which is illustrated in Fig. 1. We mainly focus on the scalable matching infrastructure that receives publications and subscriptions, and performs the matching between the two. Publications are the micro-blogging posts, which are treated as sets of words. An example is a tweet. Subscriptions are continuous queries [14] that are long-running requests to receive all publications that match a given monitoring condition. Specifically, the monitoring condition is a set of words. For instance, if a subscription is ["Obama," "health"],

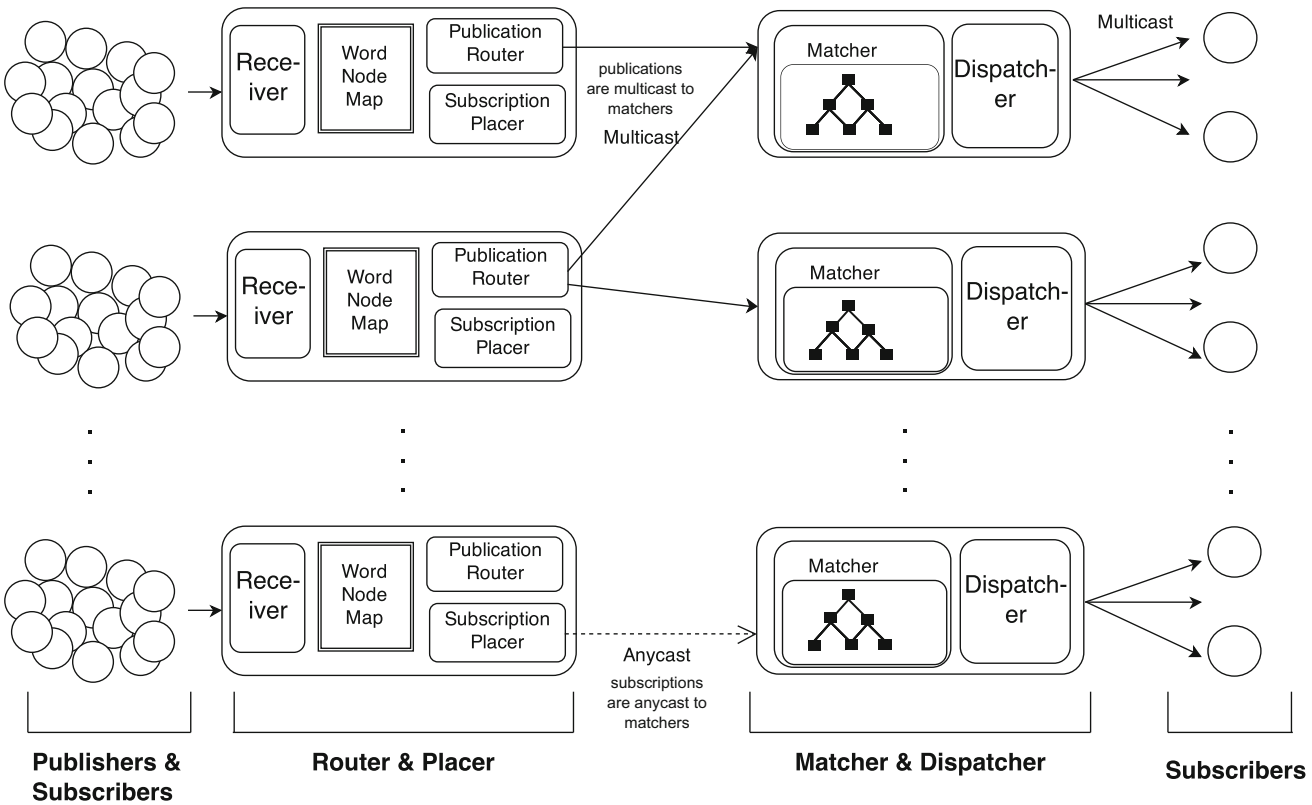


Fig. 1 Overall architecture of the S^3-TM system

then any post that contains both of the words "Obama" and "health" will be considered a match for this subscription. The results for a subscription constitute a stream, and this stream is delivered to the subscriber client that owns the subscription, as new matches take place. We assume that the publications arrive at a much higher rate compared to subscriptions, which is typical in practice for micro-blogging applications. As such, the system aims at maximizing the publication processing throughput.

S^3-TM is organized as a distributed data stream processing application that runs on a data center with multiple machines. The main flow of the application consists of two unique stages, namely the Router and Placer stage and the Matcher and Dispatcher stage. These are shown in the middle of Fig. 1. The system is designed to scale via data parallel execution; thus, there will be many copies of these stages, depending on the scale of the deployment (dashed lines in the figure).

On the left-hand side of the figure, we see the clients of the system: publishers and subscribers. We assume that each client sends its publications and subscriptions to one of the Router and Placer stages. This assignment can change at any time, as any stage instance can handle any client request. This kind of load balancing is typical for all large-scale Internet services. Note that publications flow through the system and are discarded once they are fully processed. The subscriptions, on the other hand, are stored for performing matches against future publications and are only removed upon explicit request by the subscribers. On the right-hand side of the figure, we see the subscribers again, which receive their matching publications as a stream.

In what follows, we detail the two stages that constitute the core of the scalable matching logic.

2.1 Router and Placer

This stage contains three operators within. The first one is called the Receiver, which receives publications and subscriptions from the clients. Recall that both publications and subscriptions consist of words. The Receiver operator performs stemming and stop word removal on both publications and subscriptions. Publications are then forwarded to the Publication Routing operator, whereas the subscriptions are forwarded to the Subscription Placement operator.

The Publication Routing operator is responsible for multicasting each publication to a set of Matcher and Dispatcher stages. It routes a publication to those stages that are responsible for one or more of the words contained in the publication. As an optimization, only subscribed words, that is words contained in at least one subscription, are used for the multicast. For the purpose of routing, words are partitioned over the Matcher and Dispatcher stages, such that for a given word, there is one stage responsible for it. The default partitioning policy is to hash words to stages. This default scheme has two undesirable properties. First, the spread of a hashing-based approach can be high, as it does not take into account the co-occurrence frequency of words. Ideally, words that commonly appear together should be assigned to the same stage. Second, the words might exhibit high skew, as some words are highly popular. Under skew, it becomes difficult for hashing to maintain load balance. As a result, we develop several alternative techniques for partitioning words over stages. The partitioning of words is kept as a mapping in memory as part of the Router and Placer stage and is used by the Publication Routing operator to quickly determine the target stages of a multicast for a given publication. This mapping is computed off-line and is kept as a read-only replicated copy in memory.

The Subscription Placement operator is responsible for anycasting each subscription to a set of Matcher and Dispatcher stages. A given subscription can be sent to any one of the stages that are responsible for at least one of the words in the subscription. For example, if a subscription is $[x, y]$, then the stage that is responsible for x, say S, would receive all the publications that contain the word x. Since the subscription is interested publications that contain both x and y, S is capable of evaluating the subscription. Similarly, if stage P is responsible for word y, it is also capable of evaluating the subscription. As a result, anycasting the subscription to one of the eligible stages is sufficient. The default anycast policy is to send the subscription to one of the eligible Matcher and Dispatcher stages at random. However, this policy suffers from two problems as well. First, it may not balance the load properly, as the set of eligible downstream stages is often a subset of the entire set of Matcher and Dispatcher stages and it is possible that this eligible set is skewed. Second, to reduce load, we should group together similar subscriptions as much as possible [2,11,13].

To address these issues, we develop subscription placement algorithms that run as part of the Subscription Placement operator. These algorithms use the word partitioning information kept within the Router and Placer stage (as it was used for publication routing as well), in addition to the list of currently subscribed words for each one of the Matcher and Dispatcher stages. This latter information is updated as a result of each subscription placement made, and the changes are sent to all other Router and Placer stages. This is not a performance bottleneck, as the subscription rate is expected to be much lower compared to the publication rate.

2.2 Matcher and Dispatcher

This stage contains two operators within. These are the Matcher and the Dispatcher operators. The Matcher operator is responsible for matching streaming publications against the subscriptions placed at the stage. For this purpose, we use a trie-based subscription organization, which takes advantage of similar subscriptions assigned to the same stage to reduce

the overall matching load. Finally, the dispatcher stage is responsible for sending the matching publications to the subscribers.

In a typical deployment, each stage corresponds to a process that can be distributed over machines. Multiple stages can be placed on a single machine as well, such as having one stage per processor core. In what remains, we introduce the techniques and algorithms used in publication routing, subscription placement, and matching in more detail.

3 Publication routing

In this section, we formalize the problem of publication routing and present our solutions. The goal is to come up with routing strategies that reduce spread and improve load balance. Reducing spread results in less load on the matchers, whereas improving load balance results in better utilizing the available resources. Both factors directly impact the throughput.

3.1 Formalization

Let $P \in \mathcal{P}$ be a publication, which is a set of words. Here, $\mathcal{P}$ denotes the set of all publications. Each word $w \in P$ comes from a domain of words W, where $W = \bigcup_{P \in \mathcal{P}} P$. We do not make assumptions about the subscriptions until later in Sect. 4, but we denote the set of subscribed words as W^s. We denote the number of matcher stage instances in the system as N. Our goal is to learn a mapping $M : W \mapsto [1 \dots N]$ that maximizes the throughput. This mapping maps each word to one of the matchers. The throughput, denoted by $T(M)$ for a given mapping, depends on the spread and the load imbalance. We formalize these first and define throughput as a function of them later.

3.1.1 Spread

Let $R(M)$ denote the spread for a given mapping M. The spread can be informally defined as the average number of times a publication will be routed, that is the average size of a publication multicast. Recall that a publication is routed to a matcher iff the mapping M maps a subscribed word $w \in W^s$ contained in the publication P to matcher i; i.e., the publication P is routed to matcher i iff $\exists\, w \in (P \cap W^s)$ s.t. $M(w) = i$. We denote the set of matchers a publication P is routed to as $K(P, M)$. Formally:

$$K(P, M) = \bigcup_{w \in P \cap W^s} \{M(w)\} \tag{1}$$

Given this definition, we can formally define spread, $R(M)$, as follows:

$$R(M) = \sum_{P \in \mathcal{P}} |K(P, M)|/|P| \tag{2}$$

3.1.2 Imbalance

We denote load imbalance as $B(M)$ for a mapping M and define it as the ratio of the maximum load on a matcher to the average load. In a perfectly load-balanced system, the imbalance will be 1. The worst case is when all the load is on a single matcher, in which case the imbalance will be N, that is the number of matcher stage instances. Let us denote the load imposed on a matcher i as $L(i, M)$. Formally, we have:

$$L(i, M) = \sum_{P \in \mathcal{P}} \sum_{w \in P} [w \in W^s \wedge M(w) = i] \tag{3}$$

Here, [...] is the Iverson bracket that evaluates to 1 when the Boolean condition it encloses is true, to 0 otherwise. It is important to note that here we make a simplifying assumption; that is, all publications impose an equivalent load of cost 1 unit on a matcher. We will revise this assumption when we introduce subscriptions into the picture in Sect. 4.

With the definition of load imposed on a matcher at hand, load imbalance, $B(M)$, is easily formalized as:

$$B(M) = \frac{\max_{i \in [1 \dots N]}(L(i, M))}{\sum_{i \in [1 \dots N]} L(i, M)/N} \tag{4}$$

3.1.3 Throughput

We can define throughput T simply as being proportional to the inverse of the maximum load:

$$T(M) \propto (\max_{i \in [1 \dots N]} L(i, M))^{-1} \tag{5}$$

This is because in a data parallel streaming system with a split, the throughput is bounded by the slowest branch due to backpressure [23]. Let p_i be the fraction of the publications sent to matcher i, and let C be the capacity of each matcher. Assuming a unit cost of 1 for publication processing, the throughput is bounded by $C/(p_i \cdot 1)$. We have $p_i = L(i, M)/|\mathcal{P}|$, and thus, we have:

$$T(M) = \min_{i \in [1 \dots N]}(C \cdot |\mathcal{P}|/L(i, M)) \tag{6}$$

Equation 5 follows directly from Eq. 6 after removing the constant terms.

While Eq. 6 is useful to estimate the throughput of a matching M, during the learning of a mapping, as we will see later in this section, a more flexible throughput estimation method is required to avoid getting stuck at local maximas. Intuitively, throughput can also be expressed in terms of spread and imbalance. In particular, throughput is inversely proportional

to spread, since the load on the system increases linearly with the spread. If we consider load imbalance, we see that maximum load appears as the nominator, so the throughput is also inversely proportional to the load imbalance. Thus, we can formulate an estimate throughput, denoted by $\hat{T}(M)$, as follows:

$$\hat{T}(M) \propto (R(M) \cdot B(M))^{-1} \tag{7}$$

The final problem can be formalized as finding the best mapping M^* that maximizes the throughput; that is $M^* = \text{argmin}_M T(M)$ or, alternatively, as $\text{argmin}_M \hat{T}(M)$.

In the remaining of this section, we develop techniques to learn an effective mapping M. First, we introduce several alternatives based on partitioning the word co-occurrence graph. Then we introduce the greedy SALB algorithm that makes use of the word-to-publication bipartite graph. In all approaches, we assume that the system starts with the simple hash-based routing. After an initial training period, the publications data collected so far is analyzed to generate the new mapping M, and the routing is updated to use it.

While not updating the mapping on-the-fly might seem like a drawback, our evaluation in Sect. 6.4 shows that frequent mapping updates are not required to keep the throughput high. Adding more servers would require recomputation of the mapping M as well. Thus, changes in the number of servers can be coincided with the periodic mapping updates.

It is worth mentioning that the mapping M may not contain mappings for every possible word we may see in the future. Even though we have $W = \bigcup_{P \in \mathcal{P}} P$, a new publication that arrives to the system after M has been learned may contain a new word. For such words, we fall back to the default policy of hash-based multicast.

Also note that the same mapping M is used by all the Router and Placer instances. Recall that any Router and Placer can handle any publication or subscription. Furthermore, publications and subscriptions are assigned to Receivers uniformly at random. Thus, one can consider each Router and Placer to instance be observing a sampled subset of the publications and subscriptions. This motivates using the same mapping for all Router and Placer instances. This requires mapping M to be replicated to all instances. Since the size of the mapping is limited by the number of words, it is compact enough to fit into the main memory (typically less that 200 K words, where only the word id is kept, taking less than 2 MBs).

3.2 Word network partitioning

The word network partitioning algorithms construct a mapping M by partitioning the set of words W over the N matchers. The main intuition is to place words that frequently appear together in publications into the same partition, while at the same time balancing the load incurred on each partition. We map this problem to a traditional graph partitioning one, where the words are the vertices and the edges are the co-occurring words. Let us represent this undirected graph as $G(W, E)$ and refer to it as the word network. We define the edge set as $E = \{(w_1, w_2) \mid w_1, w_2 \in W \wedge f(w_1, w_2) > 0\}$. Here, $f(w_1, w_2)$ is the co-occurrence frequency of the words w_1 and w_2. Thus, any two words that appear together in at least one publication is represented as an edge in the word network. We have:

$$f(w_1, w_2) = |\{P \mid \{w_1, w_2\} \subseteq P \wedge P \in \mathcal{P}\}|/|\mathcal{P}|.$$

The co-occurrence frequencies serve as the edge weights. We also define the frequency of a word as $f(w) = |\{P \mid w \in P \in \mathcal{P}\}|/|\mathcal{P}|$. The word frequencies serve as the vertex weights.

Graph partitioning algorithms are well studied in the literature [22] with well-established implementations, such as Metis [12]. These algorithms aim at minimizing the edge cut, defined as the total weight of the edges that go across partitions. This matches our goal of co-locating commonly co-occurring words within the same partition. It is easy to see that such a partitioning will reduce the spread, as several words within a publication will be mapped to the same matcher, reducing the size of the multicast. However, we also need to maintain the load balance. Graph partitioning is able take into account load balance as well. Yet, the load is expressed as vertex or edge weight sums. Unfortunately, it is not possible to express the processing load, as defined in Eq. 3, using such a sum. Thus, we investigate several alternative graph partitioning approaches that differ in how load balancing is formulated, all of them being heuristics. We also look at simple partitionings that serve as baselines. Figure 2 gives an overview of these alternatives, which are further detailed below:

3.2.1 Cut minimization (gC), Fig. 2a

This is a baseline partitioning that does not consider load balancing. It aims at minimizing the cut, using an unweighted word network. Thus, any pair of words that appear at least once together would contribute the same amount toward the total cut.

3.2.2 Co-frequency cut minimization (gFC), Fig. 2b

This is another baseline approach that does not perform load balancing. However, it considers the co-occurrence frequencies when minimizing the cut. Thus, words that appear commonly together are expected to be placed within the same partitions as much as possible. Since this baseline does not

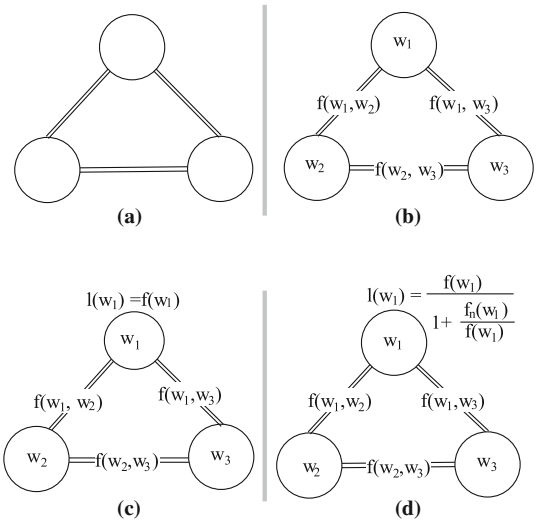


Fig. 2 Word network partitioning algorithms: **a** cut minimizing (gC), **b** co-frequency cut minimizing (gFC), **c** co-frequency cut minimizing, frequency load balancing (gFCL), **d** co-frequency cut minimizing, normalized frequency, and co-frequency load balancing (gNFCL)

consider load balance, and since load balance and spread are at odds, we expect gFC to provide a very low (good) spread and a high imbalance.

3.2.3 Co-frequency cut minimization, frequency load balancing (gFCL), Fig. 2c

This is one of the two graph partitioning-based algorithms that are contenders. Similar to gFC, it minimizes the co-occurrence frequency-based cut. Differently, it tries to maintain load balance as well. Load for a partition is defined as the sum of the vertex loads, where the vertex load is defined as the word frequency. The downside of this approach is that it overestimates the partition load. As a simple scenario, consider a small partition that contains three words that always appear together in publications. In this case, the overall partition load will be three times the correct value. The real load depends on the number of publications routed to the partition, which is lower than the sum of the word frequencies for that partition, due to co-occurrences.

3.2.4 Co-frequency cut minimization, normalized frequency, and co-frequency load balancing (gNFCL), Fig. 2d

This partitioning approach improves upon gFCL by trying to compensate for the overestimation of the partition load. Since using the word frequency as the vertex load results in overestimation, it uses a normalized vertex load for computing the overall partition load. Specifically, it uses the vertex load formulation $l(w) = \frac{f(w)}{1+f_n(w)/f(w)}$, where $f_n(w)$ is the sum of co-occurrence frequencies for the word w. That is, $f_n(w) = \sum_{(w,w')\in E} f(w, w')$. To understand the logic behind this normalization, let us consider two extreme cases. In one extreme case, a word may always appear by itself in publications. In this case, we have $f_n(w) = 0$, and thus, $l(w) = f(w)$. This is the correct load contribution to the partition for word w. As another extreme, we can consider a similar example from the gFCL discussion, that is k words that always appear together in all publications. In this case, we have $l(w) = f(w)/k$, since we have $f_n(w) = (k-1)\cdot f(w)$. The total load of the k words would be $f(w)$, which is again correct. Despite these nice features, there are many scenarios for which the partition load is not exact. As a result, this is just a heuristic too, albeit one that is more accurate than gFCL.

Once the word network partitioning is performed, the results are easily converted into a global mapping M by mapping each word in a partition to the matcher associated with that partition.

3.3 SALB: spread-aware load balancing

The SALB algorithm aims at explicitly modeling the notion of load, rather than relying on some approximation of it as done by the word network partitioning-based approaches. With a more accurate model of load, it better balances it across matchers. However, a good load balance does not necessarily imply a low overall load, since words are not independent and to achieve low average load one needs to co-locate commonly co-occurring words. This latter can be achieved by trying to minimize spread. Accordingly, SALB tries to minimize both imbalance and spread. Note that this also matches with our intuition of approximate throughput as expressed in Eq. 7.

The SALB algorithm is given in Algorithm 1. It is a greedy algorithm that assigns words to matchers one-by-one. It considers words in decreasing order of appearance frequency ($f(w)$ for $w \in W$). Frequent words are assigned a mapping first, as this provides additional flexibility to balance the load later. For each word, each matcher is considered as a candidate mapping and the one with the highest utility is picked as the one to be added to the mapping. The process continues until all words are assigned a mapping. The utility used for picking the best among all matchers is defined as spread times load imbalance times -1 (making higher values better), where spread and imbalance are computed as if the candidate mapping is already applied.

To compute the spread and load imbalance incrementally as words are assigned to matchers, we first build a bipartite graph $G(W, \mathcal{P}, E)$, where W is the set of words and $\mathcal{P}$ is the set of publications. There is an edge (w, P) in E if and only if the word w is contained in the publication P, that is $w \in P$. We use $nbr_G(w)$ to denote the set of neighbors of the

Alg. 1: SALB, Spread-Aware Load Balancing

Data: $\mathcal{P}$, set of publications
Data: N, number of matchers
Result: M, word to matcher mapping
$M \leftarrow \{\}$ ▷ Initialize the mapping
$R \leftarrow 0$ ▷ Initialize the spread
$\forall_{i \in [1 \ldots N]}, L_i \leftarrow 0$ ▷ Initialize loads
$W \leftarrow \bigcup_{P \in \mathcal{P}} P$ ▷ Collect words
▷ Form the word-to-publication bipartite graph
$G(W, \mathcal{P}, E)$ s.t. $E = \{(w, P) \mid w \in W \wedge P \in \mathcal{P} \wedge w \in P\}$
for $w \in W$ in desc. order of $f(w)$ **do** ▷ For each word
  $u^* \leftarrow -\infty$ ▷ Initialize utility for the best mapping
  $l^* \leftarrow 0$ ▷ Initialize delta load for the best mapping
  $k \leftarrow 0$ ▷ Initialize the best mapping index
  for $i \in [1 \ldots N]$ **do** ▷ For each matcher
    ▷ Compute the extra load w brings to matcher i
    $l \leftarrow \sum_{P \in nbr_G(w)} [\nexists w' \in P \text{ s.t. } M(w') = i]$
    $r \leftarrow R + l/|\mathcal{P}|$ ▷ Compute spread
    $\mathcal{L} \leftarrow \bigcup_{j \in [1 \ldots N] \setminus \{i\}} \{L_j\} \cup \{L_i + l\}$ ▷ Union loads
    $b \leftarrow \sqrt{\text{var}(\mathcal{L})}/\text{avg}(\mathcal{L})$ ▷ Compute imbalance
    $u \leftarrow -r \cdot b$ ▷ Compute utility
    if $u > u^*$ **then** ▷ If a better mapping
      $u^* \leftarrow u$ ▷ Update the best utility
      $l^* \leftarrow l$ ▷ Update the delta load
      $k \leftarrow i$ ▷ Update the best mapping
  $L_k \leftarrow L_k + l^*$ ▷ Update the load of the matcher
  $R \leftarrow R + l^*/|\mathcal{P}|$ ▷ Update the spread
  $M(w) \leftarrow k$ ▷ Add the new mapping
return M ▷ Return the constructed mapping

word w in graph G, i.e., the set of publications that contain the word w.

Consider a candidate mapping of word w to matcher i. In order to compute the new spread and imbalance incrementally, a key quantity we need to compute is the additional load this mapping will introduce on the matcher. This amount is denoted via l in the algorithm. We have $l = \sum_{P \in nbr_G(w)} [\nexists w' \in P \text{ s.t. } M(w') = i]$. That is, we find all publications that contain the word w (i.e., $P \in nbr_G(w)$), and for each such publication P, we add 1 to the load if the publication does not contain any other word that is already mapped to matcher i (i.e., $\nexists w' \in P$ s.t. $M(w') = i$). Given this quantity, we can incrementally compute the new spread by adding $l/|\mathcal{P}|$ to the existing spread, as l gives the increase in the number of publications that are routed as a result of adding a new mapping.

Recall that we define utility in terms of spread times imbalance. We already discussed how spread is incrementally updated. Similarly, we update the load imbalance incrementally. For imbalance, we use a slightly different formulation than the ratio of maximum load to average load. Using the maximum term in the formulation results in a highly insensitive metric during the initial iterations of the algorithm, as mappings to matchers other than the one that changes the maximum load make a very small impact. Thus, as an imbalance metric, we use *coefficient of variance* of the matcher loads. Since we have computed the extra load brought by the new mapping, that is l, we can easily come up with the new set of loads on the matchers. This is denoted as the set $\mathcal{L}$ in the algorithm. Then the imbalance is given by $\sqrt{\text{var}(\mathcal{L})}/\text{avg}(\mathcal{L})$, which is the standard deviation of the loads divided by the average load (aka. coefficient of variance). The normalization via the average load is included in the formulation (the denominator), since different candidate mappings may result in different total loads.

Complexity. The outer loop of the algorithm iterates $|W|$ times, and the inner loop iterates $|N|$ times. Assuming there are k words per publication on average and there are d publications containing a word on average, the inner loop performs $\mathcal{O}(d \cdot k + N)$ operations. The N part comes from the computation of the imbalance. In practice, both variance and average can be computed incrementally, yet for brevity we have not shown that in the algorithm. So the inner loop's body can complete in $\mathcal{O}(d \cdot k)$ time. This results in an overall complexity of $\mathcal{O}(d \cdot k \cdot N \cdot |W|)$. We know that k is a small constant irrespective of dataset size, so we can represent the complexity simply as $\mathcal{O}(d \cdot N \cdot |W|)$. The average number of publications a word appears in is bounded by $|\mathcal{P}|$, so an even simpler time complexity formula can be given by $\mathcal{O}(N \cdot |\mathcal{P}| \cdot |W|)$, even though this bound will be rather loose. Also note that we can add the $\log |W| \cdot |W|$ term that comes from the sorting, but this is not necessary as the other multiplicative terms in front of $|W|$ are larger than $\log |W|$ in practice.

Our experimental results show that SALB algorithm performs favorably in terms of the running time compared to graph partitioners on large datasets.

4 Subscription matching and placement

The default policy used for placing subscriptions on matchers is to anycast them to one of the eligible matchers. Let S be a subscription, which is a set of words. We denote the set of eligible matchers as $B(P, M)$ under a given mapping M and define it as $B(P, M) = \{i \mid \exists w \in S \text{ s.t. } M(w) = i\}$. This policy is suboptimal as it does not attempt to group together similar subscriptions and doing so can significantly reduce the load. However, in order to do such a grouping, we need a better understanding of the matching process.

4.1 Matching

We perform the matching using a trie data structure. We sort each subscription before it is inserted into the trie, so that its words are in lexicographic order. The trie takes advantage of common prefixes within the subscriptions. Each trie node has zero or more child nodes, each associated with a word,

Alg. 2: LASP, Load-Aware Subscription Placement

Data: S, subscription to be placed
Data: N, number of matchers
Data: M, word to matcher mapping
Data: H, subscription word map
Result: k, the matcher where the subscription is placed
$u^* \leftarrow -\infty$ ▷ Initialize utility for the best placement
$k \leftarrow 0$ ▷ Initialize the matcher for the best placement
$B(P, M) = \{i \mid \exists\, w \in S \text{ s.t. } M(w) = i\}$ ▷ Eligible ones
for $i \in B(P, M)$ **do** ▷ For each eligible matcher
  $l \leftarrow f(|S \backslash H(i)|)$ ▷ Compute subs. delta load
  ▷ Union all load lists
  $\mathcal{L} \leftarrow \bigcup_{j \in \{1 \ldots N\} \backslash \{i\}} \{f(|H(j)|)\} \cup \{f(|S \cup H(i)|)\}$
  $b \leftarrow \sqrt{\text{var}(\mathcal{L})}/\text{avg}(\mathcal{L})$ ▷ Compute imbalance
  $u \leftarrow -l \cdot b$ ▷ Compute utility
  if $u > u^*$ **then** ▷ If a better mapping
    $u^* \leftarrow u$ ▷ Update the best utility
    $k \leftarrow i$ ▷ Update the best placement
  $H(i) \leftarrow H(i) \cup S$ ▷ Update subscription word map
  return k ▷ The matcher for the best placement

and a potentially empty list of subscriptions. For trie nodes that have large number of children, the child nodes are kept in a hash table. We make use of these hash tables for fast search. For instance, the root node has as many children as there are unique start words in sorted subscriptions.

When a publication is to be matched against the set of subscriptions stored in a trie, we do a scoped traversal of the trie. During the traversal, a child node is visited if and only if its associated word is in the publication. To check this, we probe the child hash table using the set of words in the publication. Since our publications are short, this is quite efficient. Note that, during the traversal, for any visited trie node we are guaranteed that all the words up to the root are in the publication. Thus, whenever a trie node is visited, any subscriptions associated with it are added to the result.

4.2 Load-Aware Subscription Placement

For placing subscriptions, we introduce an algorithm called Load-Aware Subscription Placement, LASP for short. The LASP algorithm is executed within the Subscription Placement operator as part of the Router and Placer stage instances. Any stage instance can place any subscription. To facilitate this, we keep a replicated data structure called the *subscription word map*, denoted as H. For each matcher i, the subscription word map contains the set of unique words that appear in subscriptions assigned to that matcher, denoted as $H(i)$. This structure is potentially updated every time a new subscription is placed. Since the subscription rate is much lower than the publication rate, propagating updates regarding the changes on this structure is cheap. Alternatively, this structure can be kept centralized.

The LASP algorithm, given in Algorithm 2, is structured similar to the SALB algorithm's inner loop. It iterates over all possible placements, each corresponding to placing the subscription on one of the eligible matchers. For each eligible matcher ($i \in B(P, M)$), it computes a utility metric and picks the one with the highest utility as the matcher to place the publication on. The utility is defined as the increase in the subscription load of the matcher times the load imbalance times -1 (making higher values better).

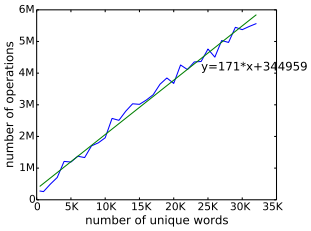


Fig. 3 Number of lookup ops

Subscription load is proportional to the cost of matching a publication against the set of subscribers placed on the matcher. We make a simplifying assumption here: We assume that the matching cost (represented via the f function in the algorithm) is linear in the number of unique words in the trie. This assumption is motivated by the observation that the higher amount of overlap in the subscriptions reduces the size of the trie, which is equal to the number of unique words in it ($|H(i)|$ for matcher i).

Figure 3 plots the number of operations performed in our trie implementation as a function of the number of unique words, using the workload setup described in Sect. 6. We fit a linear line on this graph and employ it as the f function used by the LASP algorithm.

5 Extensions

In this section, we present extensions to the base S^3-TM system to solve two problems commonly encountered in practice, namely skew in publication workload and unexpected spikes in load.

Skew in word frequencies causes high load imbalance and results in limited scalability. The skew becomes more pronounced when the number of machines increases, as the load brought by a single word on a matcher may exceed the average load per machine. By handling skew via the help of a word splitting mechanism that is adaptive to the number of machines used, we reach near-linear scalability.

A micro-blogging service may experience unexpected load spikes, often due to a sudden mass reaction from the user base. Without any special mechanism, these spikes may result in randomly dropping incoming publications, significantly reducing the match quality. We develop load shedding techniques that aim at minimizing the impact of load spikes on match quality.

5.1 Skew handling

When scaling up to large number of nodes, load imposed by some of the words might exceed the average load of a node. Such *hot* words cause skew, since proposed algorithms have the limitation that any given word can be assigned to only a single matcher. Our initial experiments showed that SALB scales linearly up to 64 nodes with many of the real-world tweet datasets. After 64 nodes, linear scalability is lost, and after 128 nodes, no additional speedup is achieved.

To handle skew, we first find the average aggregate word frequency a node should handle. Since SALB tries to balance the load across nodes while keeping the spread low, having a word with frequency higher than the average frequency causes increased load imbalance. Therefore, we limit each word to have at most frequency equal to the half of the aggregate frequency a node should handle. As a result, a single word can only account for half of a node's even share of load. If a word does not satisfy this condition due to high frequency, we split the word into *versions*, until the condition is satisfied. If a word is split into k versions, then that word is replaced with a random version in range $[0 \ldots k)$ when it is encountered within a publication. This effectively reduces the load a single word can incur on a matcher.

This leaves us one last problem; that is, how to place subscriptions that contain one of the hot words. There are three types of subscriptions with respect to the hot words: (i) those that do not contain any of the hot words, (ii) those that contain both hot and regular words, and (iii) those that contain only hot words. For the first category (no hot words), no change is required during subscription placement. For the second category (both hot and regular words), since the LASP algorithm already selects the least frequent words during placement, hot words are eliminated already and subscription is anycast to one of the nodes that are responsible for a regular word from the subscription. Finally, for the last category (all hot words), the hot word with the least frequency is selected and the subscription is multicast to all nodes that are responsible for a version of the selected hot word. The multicast is needed, because multiple nodes may be handling the different versions of the selected hot word. Luckily, the third category of subscriptions is small in size.

Splitting words into versions is performed during the learning phase. It impacts both the creation of the mapping used for routing and the assignment of subscriptions to machines. When the learning phase is repeated, the mapping is updated and the subscriptions are replaced.

5.2 Load shedding

Micro-blogging services may experience unexpected spikes in load due to mass reaction from the users to rare and noteworthy world events. In such scenarios, the input publication rate may exceed the maximum throughput that can be handled by the system. This requires shedding some load to avoid lengthy delays and eventual random dropping of the publications.

There are two aspects to load shedding in streaming systems [25]: How much load to shed and what load to shed. The former typically changes as the workload and resource availability varies, and as such, requires an adaptive solution. In what follows, we first describe how we resolve the 'what' question, and then we describe the adaptive load shedding technique we use to handle the spikes in load (the "how" question).

5.2.1 What load to shed

The most straightforward way to shed load is to randomly drop publications. An alternative and more effective way is to limit the number of matchers they are multicast to. This reduces the spread, and thus load. We perform load shedding by limiting the maximum number of matchers a publication is routed to, say m. If the publication at hand has more than m target matchers it ideally should be routed to, then we only route it to the m matchers that have the highest utility metric. We use two such metrics:

- *Consensus shedding*: Forward to the matchers with the highest number of publication words mapping to them. The main idea is to reduce the number of publication words for which forwarding is not performed, as this may improve the overall match quality.
- *Subscription shedding*: Forward to the matchers that contain the highest number of subscriptions for the words in the publication. The main idea is to minimize the impact of load shedding on the match quality, as the publication is routed to matchers that are more likely to produce matches.

5.2.2 How much load to shed

The Publication Routing operator keeps a buffer of publications. When a new publication is received, it is enqueued into this buffer. A separate thread pulls publications from this buffer and routes them to the matchers. The overload is detected when the buffer is full. The size of the buffer, say b, can be adjusted based on the latency requirements of the system.

We perform dynamic load shedding by making use of this buffer. In particular, we extend it with two additional segments, resulting in a total of three segments. The front of the buffer is called the *ideal* segment, which represents the ideal mode of operation in terms of the buffer fullness. The next segment is called the *stable* segment, and the one following it is called the *overload* segment. The idea is that the sys-

tem will increase the level of load shedding when the buffer fullness is in the overload segment, and reduce the level of load shedding when the buffer is in the ideal segment. No changes will be made when in the stable segment. The goal of the stable segment is to avoid oscillation.

We define lowest shedding level as $l = 0$, which corresponds to $m = \infty$. Level $l = 1$ corresponds to $m = k$, where k is 7 based on our experimentation (see Sect. 6). Each successive level has m decreased by Δ, such as $m = k - \Delta$ and $m = k - 2 \cdot \Delta$ for $l = 2$ and $l = 3$, respectively. Δ could be less than 1, which corresponds to probabilistic forwarding for the last word selected for forwarding.

Let b^i and b^s be the sizes of the ideal and the stable segments. We have $b = b^s + b^i$, and we ensure that the system operates such that the overload segment is avoided via increasing the shedding level. One important point is that we need to avoid oscillation in the system. In particular, the system should not jump from the ideal segment into the overload segment as a result of a single level reduction in the shedding level. We achieve this by adjusting the ratio $r = b^i/b^s$. Let us represent the load in the system for shedding level l as $L(l)$. Modeling the system as a queueing one and applying Little's Law, we say that the queue length is proportional to the input rate times the processing time (roughly inverse of the load level). This gives the following inequality:

$$(b^s + b^i)/b^s > L(l - \Delta)/L(l), \quad \forall l \tag{8}$$

This ensures that reducing the load shedding level never takes the buffer fullness from the ideal segment to the overload segment. We have:

$$r = 1 - \max_l L(l)/L(l - \Delta) \tag{9}$$

We also need to ensure that the system does not move from the overload segment to the ideal segment when the shedding level is increased. That condition is already satisfied by Eq. 9. Finally, the L function is easily computed experimentally, as we will show in Sect. 6.5.

It is important to note that we may increase (decrease) the load shedding level due to being in the overload (ideal) segment, yet when the next adaptation time comes, we might still be in the same segment. In this situation, we continue to decrease (increase) the load shedding level if and only if the buffer fullness level has not went down (up) since the last adaptation time. Given this, we can set the adaptation period low, conservatively. In our system, we set the adaptation period to 1 second.

6 Experimental evaluation

In this section, we evaluate the scalability and performance of the S^3-TM system, with a particular focus on the effectiveness of our publication routing and subscription placement algorithms. The evaluation includes five sets of experiments. The first set of experiments studies scalability, presenting performance as a function of the number of nodes. The second set studies subscription awareness, presenting performance as a function of the number of subscriptions. The third set studies concept drift, that is how the performance of the system is impacted by the temporal changes in the contents of the publications. The fourth set studies the efficacy of the load shedding algorithms. Finally, the last set of experiments studies the learning time of alternative algorithms used for learning the word to matcher mapping. In most of our experiments, we make use of the spread, load imbalance, and throughput metrics. All experiments are performed using tenfold cross-validation, and error bars showing the standard deviation are included in the plots.

The word network partitioning-based algorithms make use of Metis 5.1.0 [12] for graph partitioning. In contrast, the SALB algorithm does not make use of graph partitioning. Mallet [15] implementation of Latent Dirichlet Allocation (LDA) [3] is used for creating topic-based subscriptions, as we will detail later.

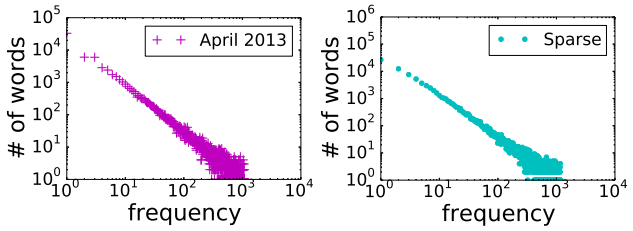
The S^3-TM system is implemented in Python. We use CPython 2.7 series for learning the word to matcher mapping and PyPy 2.7 series (which includes a JIT) for runtime subscription matching. All experiments are executed on Linux machines with 2 Intel Xeon E5520 2.27 GHz CPUs and 48GB of RAM per machine. In the rest of this section, we use the term *node* to refer to a core on a machine. Since we go up to 256 nodes and since each machine has 12 cores in total, we use 1 to 24 machines, depending on the experimental setup.

6.1 Experimental workload

Experiments are performed using two different datasets, details of which are shown in Table 1. Both datasets contain public tweets in the English language, collected using the Twitter Streaming API [27]. These tweets are used for learning the word to matcher mapping. Before learning, we perform preprocessing, including cleaning, stemming, and stop word removal. Cleaning involves removing any non-word tokens (numbers are kept), links starting with the word "http," words starting with @ (screen names), and punctua-

Table 1 Properties of the attributes in the learning corpus

Datasets ⇒	April 2013	Sparse
# Of tweets (sampled)	979,442	979,442
# Of words (sampled)	100,310	198,887
Total word freq. (sampled)	3,874,826	10,190,479
# Of word pairs (sampled)	5,507,437	7,559,671
# Of tweets (unsampled)	9,791,543	10,467,110



(a) April 2013 **(b)** Sparse

Fig. 4 Word frequencies

tions. Each word is stemmed using Porter's algorithm [18]. Stop word removal is performed based on a common stop words list taken from the Mallet library.

The first dataset consists of tweets we collected in April 2013. We used random sampling to create a learning corpus of around 1 million tweets. The learning corpus contains about 100 thousand unique words after preprocessing. Counting multiple occurrences of those words, there are around 3.8 million word occurrences and those words create 5.5 million pairs.

The second dataset is a publicly available tweet dataset called Sparse [6]. We also sampled this dataset to create a learning corpus of around 1 million tweets. The learning corpus contains about 200 thousand unique words, 10.2 million total word occurrences, and 7.6 million word pairs. Figure 4 shows the word frequency distributions of the learning corpus we extracted from the two datasets.

The motivation behind using a learning corpus of size 1 million tweets is the following. The effectiveness of the word to matcher mapping is impacted by the frequent words, and it is sufficient to capture those words with a sample of size 1 million. However, increasing the learning corpus size unnecessarily significantly increases the learning time (see Sect. 6.6). To verify the sampling claim, we used the Sparse dataset to measure the impact of increasing the learning corpus sample size on the throughput. The results are depicted in Fig. 6. We observe that increasing the learning corpus size beyond 0.5 million brings diminishing returns in terms of throughout. The main intuition behind this is that only words with a significantly high frequency are important enough to impact the spread and load balance. As such, a sample of 1 million tweets is as good as 10 million for the purpose of learning.

We generated the subscriptions using two alternative methods. The first one is called *tweet-based subscriptions* and the second one is called *topic-based subscriptions*. To create tweet-based subscriptions, we pick random tweets from the dataset and register them as subscriptions to the system. To create topic-based subscriptions, we model the interests of the users. Specifically, we created a topic extractor using LDA [3] implementation of the Mallet library. We extracted 100 topics from each dataset. For each topic, we selected 5 words related to it. Alpha and beta parameters of LDA are set to 0.1, which is the Mallet default. Since the length of the subscriptions may show variability, we used a Zipf distribution to decide how many predicates a subscription contains. Each subscription selects one topic, decides its length using a Zipf distribution with a skew parameter of 0.5, and gets that number of words from the topic at random. Shortest publication contains a single word and the longest contains 5 words. Overall, the tweet-based model represents the scenario where we have relatively long subscriptions, with low popularity, whereas the topic-based model represents the scenario where we have relatively short subscriptions, with high popularity.

In the rest of this section, we present our experimental results. For brevity, we use the April 2013 dataset for the throughput, spread, and load imbalance experiments, as the results from the other dataset are very similar. For the relative throughput experiments, we use the average values computed using both datasets.

6.2 Scalability

We look at the spread, load imbalance, and throughput as a function of the number of nodes in the system. Here, the number of nodes corresponds to the number of matcher instances, which is the number of cores in our system. We also plot the relative throughput, where we take the throughput achieved using the matching learned via the SALB algorithm as 1 and report the throughput of the alternative approaches relative to that. The geometric mean of the relative throughputs from both datasets is used. The number of subscriptions used for this set of experiments is 100 thousand.

Figure 5a, e plots the relative throughput tweet- and topic-based subscriptions, respectively. We observe that SALB performs up to 130 and 150% better than the baseline hash-based routing, for tweet- and topic-based subscriptions, respectively. Overall, for topic-based subscriptions the improvement relative to hashing is more lasting as the number of nodes increases.

Our main concern is the throughput of the system, and Fig. 5b plots it as a function of the number of nodes, which ranges from 2 to 256. We observe that gC and gFC perform more than an order of magnitude worse than the best approach, so they are not contenders. For the remaining algorithms, we see close to linear scalability up to 128 nodes. After 128 nodes the throughput starts to decrease, except for SALB. We observe that SALB provides the best throughput and scalability, where gFCL and gNFCL are second, with the former being slightly better than the latter, and hash-based routing is the last. The results for the topic-based subscriptions, shown in Fig. 5f, are even more pronounced. In particular, for a 256 node system, SALB provides 2.56 times better throughput than the baseline hash-based routing

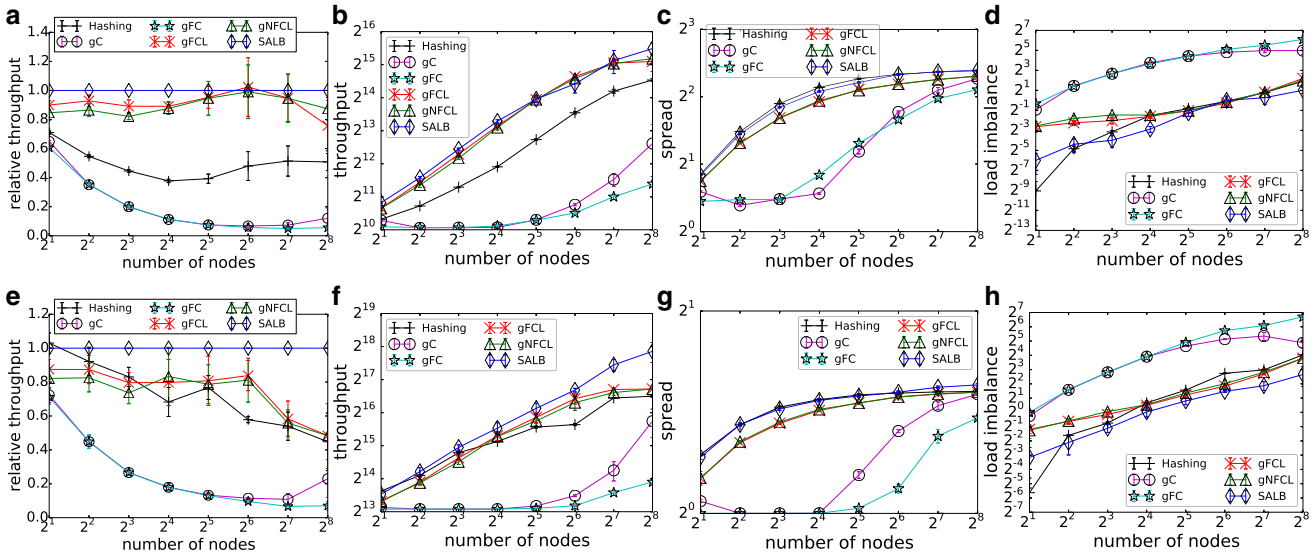


Fig. 5 Relative throughput (**a**), throughput (**b**), spread (**c**), load imbalance (**d**), tweet-based subscriptions. Relative throughput (**e**), throughput (**f**), spread (**g**), load imbalance (**h**), topic-based subscriptions

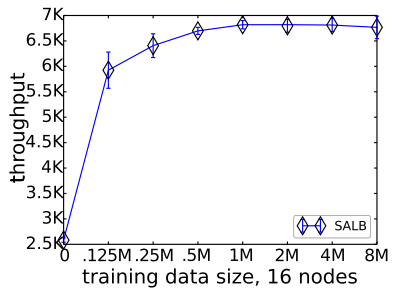


Fig. 6 Throughout versus the learning corpus size

and 2.2 times better throughput than the gFCL and gNFCL approaches.

Figure 5c plots the spread of the routed publications using the tweet-based subscription model. Note that the minimal spread value that can be achieved is 1. We observe that as the number of nodes increases, the spread increases as well, but the rate of increase decreases and eventually the line flattens. This is expected, as we know that the spread is bounded by the maximum number of words in a publication. We also observe that the cut-based graph partitioning algorithms that do not care about load balance (gC and gFC) provide the lowest spread. This is because these algorithms place frequently co-occurring words to the same matchers. But as we will see soon, the load imbalance of these algorithms will result in poor throughput, which is the ultimate metric we care about. The graph partitioning approaches that consider load (gFCL and gNFCL) provide lower spread than the hashing-based routing and SALB algorithms. SALB provides slightly lower spread than hashing, but higher than that of gFCL and gNFCL. Interestingly, as the number of nodes in the system increases, the spread converges to the same number for gFCL and gNFCL, yet hashing and SALB converge to a slightly higher spread.

Figure 5g plots the spread for the topic-based subscription model. The results are similar, with a few notable differences. First, the spread is much lower in general, not crossing 2. Second, the spread difference between the hashing-based routing and SALB is much smaller. Since non-subscribed words are not forwarded to matcher nodes, we observe that spread of the topic-based subscriptions are much lower than the tweet-based ones. As we will see shortly, the story is quite different for load imbalance.

Figure 5d plots the load imbalance using the tweet-based subscription model. We observe that gC and gFC approaches suffer a very high load imbalance, and as we will later observe in throughput experiments, this imbalance causes their throughput to be non-competitive. The SALB algorithm provides the best load imbalance among all. The hash-based routing has imbalance values that are mostly between those of gFCL/gNFCL and SALB. As the number of nodes in the system increases, the imbalance of hashing gets closer to that of gFCL/gNFCL and eventually passes it. This is because for a skewed workload, load balancing becomes increasingly difficult with more nodes. We also observe that gNFCL has slightly higher imbalance than gFCL. Despite considering load balance explicitly, both of these algorithms still fall short in balancing the load and SALB has six and four times better lower imbalance in an eight-node configuration compared to gNFCL and gFCL, respectively. As the number of nodes reach higher values, like 256, the difference between load imbalance values gets smaller, but SALB still performs the best.

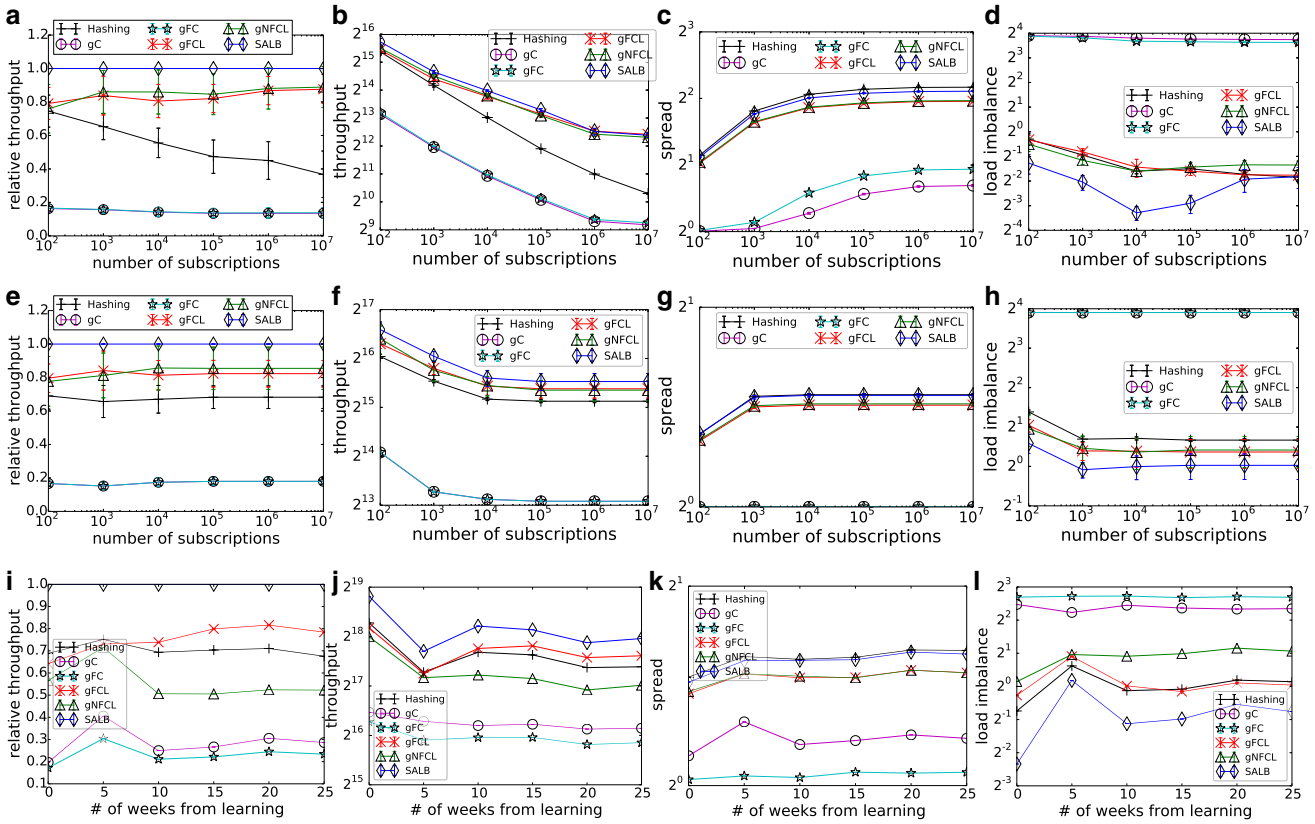


Fig. 7 Relative throughput (**a**), throughput (**b**), spread (**c**), load imbalance (**d**), tweet-based subscriptions. Relative throughput (**e, i**), throughput (**f, j**), spread (**g, k**), load imbalance (**h, l**), topic-based subscriptions

Figure 5h plots the load imbalance for the topic-based subscription model. The results are very similar. The load imbalance is higher in general for topic-based subscriptions, but its rate of increase with increasing number of nodes is lower. Also, for topic-based subscriptions, gFCL has slightly higher imbalance than gNFCL (reversed from tweet-based subscriptions).

6.3 Subscription awareness

We look at the spread, load imbalance, and throughput as a function of the number of subscriptions in the system. We experiment with number of subscriptions that range from 100 to 10 million. The number of nodes is fixed to 16 for this set of experiments. We perform experiments with both tweet-based and topic-based subscriptions. It is important to note that for tweet-based subscriptions, registering 10^7 random tweets gets close to an all-words-subscribed system, which is the worst case scenario for the S^3-TM architecture. This is a highly unlikely scenario in a real-world system, and we use it as a stretch test.

Figure 7a, e plots the relative throughput, for tweet- and topic-based subscriptions, respectively. For the tweet-based subscriptions, SALB provides 15 % better throughput compared to gFCL and gNFCL, and 10 % better throughput compared to gFCL, until 10 thousand and 100 thousand tweet-based subscriptions, respectively. Scaling to 10 million tweet-based subscriptions, SALB still outperforms other approaches. As we mentioned earlier, at this point the system converges to an all-words-subscribed system and minimizing spread becomes critically important. As we have seen from most of the experiments so far, SALB is better at minimizing load imbalance than minimizing spread. That being said, the all-words-subscribed scenario is highly unlikely to be encountered in practice. We also observe that with increasing number of tweet-based subscriptions, the performance of the hash-based routing degrades. For topic-based subscriptions, SALB provides 22 and 18 % better throughput compared to gNFCL and gFCL, and 42 % better throughput compared to hashing, respectively.

Figure 7b, f plots the throughput for tweet- and topic-based subscriptions, respectively. For tweet-based subscriptions, there is an almost linear decrease in the throughput until 1 million subscriptions, whereas for topic-based subscriptions, the rate of throughput reduction quickly diminishes after 10 thousand subscriptions. The latter can be easily explained by the high amount of overlap across the subscriptions for the topic-based model. The former can be explained

by the reverse, that is low overlap among the tweet-based subscriptions. This effect shows the importance of the LASP algorithm for grouping together similar subscriptions.

For both subscription models, SALB algorithm outperforms the alternatives. The gap between the SALB algorithm and hashing initially increases as the number of subscriptions increases. Interestingly, for tweet-based subscriptions the gap continues to widen, whereas for topic-based ones it stabilizes. SALB outperforms hashing by more than 4.2 times and 1.42 times for tweet- and topic-based subscriptions, respectively. For tweet-based subscriptions, SALB is only marginally better than gFCL and gNFCL, whereas for topic-based subscriptions the difference is more pronounced.

Figure 7c, g plots the spread for the tweet- and topic-based subscriptions, respectively. Likewise, Fig. 7d, h plots the load imbalance for the tweet- and topic-based subscriptions, respectively. In general, we observe relationships between the different alternatives as before. SALB has markedly better load imbalance than other alternatives, whereas gFCL and gNFCL have better spread than SALB. SALB's spread is slightly better than hashing for tweet-based subscriptions, but for topic-based subscriptions their spread is the same (lines overlap in the figure). The spread increases with increasing number of subscriptions, but with a decreasing rate that diminishes quickly. The load imbalance increases with increasing number of subscriptions, but again with a decreasing rate that diminishes eventually. SALB keeps its load imbalance advantage across the range, having up to 3.3× lower imbalance than the hashing approach for the tweet-based subscriptions and 1.7× lower for the topic-based ones.

6.4 Concept drift

Figure 7i–l plots relative throughput, throughput, load imbalance, and spread as a function of time. Time corresponds to the number of weeks passed since the learning was performed using the word to matcher mapping. We use the tweets from week 0 to build the word to matcher mapping and use it for evaluating the performance for the following weeks. We report average metrics for 5-week intervals to reduce noise. For this set of experiments, 100 thousand topic-based subscriptions were used. To be able to track the concept drift of subscriptions as well, for each week we extracted new topics and created a new subscription set.

We observe that the throughput is markedly higher for week 0. This is expected, as the model is specifically built for the that week, and certain amount of overfitting exists. The throughput decreases by a factor of 2 after the first week, and Fig. 7k, l shows that this is due to both the increase in the spread and the load imbalance. However, the increase in load imbalance is sharper for all contender approaches. Even though the increase in imbalance is most steep for SALB, it still has better imbalance compared to all others,

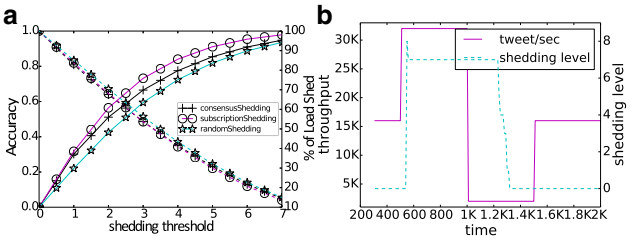


Fig. 8 **a** Accuracy and amount of load shed. **b** Input rate and shedding level

and we observe from Fig. 7i, j that it maintains the throughput advantage over other approaches across the entire time range. Importantly, while there is an initial decrease in throughput after week 0, there is no decreasing trend afterwards. This can be explained by the nature of the spoken languages. Irrespective of the current topics of interest, there is a common structure of the spoken language that makes certain words appear together and learning that structure is sufficient to achieve better scalability and throughput.

6.5 Load shedding

Figure 8a plots the accuracy of matching (on the left y-axis using solid lines) as well as the percentage of load shed (on the right y-axis using dashed lines), as a function of the load shedding threshold (the maximum number of matcher instances a publication is forwarded to). Accuracy is defined as the fraction of the correct matches produced by the system. Note that performing load shedding cannot result in superfluous matches, but only missing matches. As we decrease the shedding threshold, the accuracy initially decreases by a small amount. But as the shedding threshold gets smaller, the rate of decrease in the quality increases. In general, the shape of the quality curve is friendly to load shedding. However, the curve for the percent of load shed is not as friendly. This is because the amount of load shed is low for large thresholds and the rate of increase is initially slow when the threshold is high and increases later as the threshold gets smaller. Still, the load shedding is effective. For instance, it is possible to shed close to 25 % of the load, while still maintaining 90 % accuracy. Among the two load shedding approaches we have proposed, that is subscription shedding and consensus shedding, the former is more effective, as it can provide higher accuracy for the same amount of load shed.

Figure 8b plots the input throughput (tweets/sec, on the left y-axis using solid lines) as well as the load shedding levels (on the right y-axis using dashed lines), as a function of time. Increased load shedding level implies a lower shedding threshold. Note that, this experiment does not start from time 0, since we wait for the buffer that holds the publications to stabilize. Also, in this experiment, we display the throughput and load shedding values for a single Router and

Placer machine. Starting with 16 thousand publications per second input rate, at time 500 we increase the input throughput to 32 thousand, and at time 1000 we decrease it down to 2 thousand. After time 1500, we again go back to 16 thousand publications per second. Using this setup, we show how the shedding level adapts to the changes in the input throughput.

We observe that the change in the shedding level shows a similar pattern with the changes in the input rate, but it is often shifted toward right. This delayed reaction is due to the buffering effect and is more pronounced when the buffer is full (overloaded scenario). For instance, at time 500, the buffer is not full, and the sudden increase in throughput quickly fills up the buffer and takes us to the overload segment. As a result, the algorithm quickly adapts and increases the shedding level to 7 (one below the maximum of 8). However, when there is a very sharp decrease in input rate at time 1000, it takes a longer time for the shedding level to come down. This is because of the large buffer size we use. It takes time for the already buffered publications to be processed. Eventually we get to the ideal region, and the shedding level is lowered. The buffer size can be adjusted based on the latency that could be tolerated. For small buffer sizes, the time it will take for us to lower the shedding level will be shorter.

6.6 Learning time

Figure 9 plots the time it takes to build the word to matcher mapping from the training dataset, as a function of the number of publications in the dataset. It is important to note that the graph partitioning-based algorithms make use of the Metis library, which is a highly optimized C implementation. The SALB algorithm, on the other hand, is a Python implementation. As a result, here we want to focus more on the trend, rather than the absolute numbers. We observe that the rate of increase in the amount of time it takes for SALB to create the mapping is lower compared to graph partitioning-based approaches and after 1 million tweets, SALB starts to take less time. For 1 million tweets, which is the number we have used in all our experiments, it takes around a minute for graph partitioning approaches to compute the mapping and around two minutes for SALB. For 10 million tweets, the number raises to around 4 hours for SALB and slightly higher for the graph partitioning-based approaches.

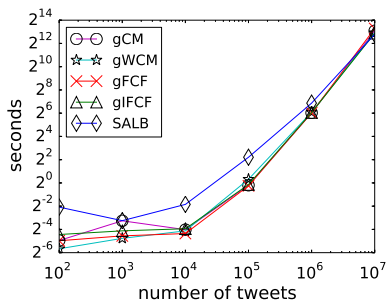


Fig. 9 Learning time

6.7 Discussion

In summary, our experimental evaluations show that:

- The word to matcher mapping created by the SALB algorithm is effective in increasing the throughput of short text matching compared to hashing, by as much as 2.5 times.
- SALB works better than word network partitioning-based solutions, due to its ability to balance the load, in addition to reducing the spread. The word network partitioning approaches fail at the former.
- To achieve scalability for large number of nodes, popular words causing high skew need to be handled via splitting.
- Under extreme load, smart load shedding techniques can be used together with SALB to provide graceful degradation in matching quality.

7 Related work

We discuss prior work related to S^3-TM with an emphasis on pub/sub systems as well as matching and filtering techniques. S^3-TM is relevant to content-based publish/subscribe systems, as it evaluates monitoring queries (subscriptions) against micro-blog posts (publications). Matching and filtering are relevant too, as one of the core components of S^3-TM is the comparison of publications and subscriptions to detect matches.

7.1 Publish/subscribe (pub/sub) systems

Pub/sub systems can be classified into topic based and content based, depending on the matching model. Much early work on pub/sub was topic based, wherein the messages are filtered based on a single topic string (e.g., TIBCO [26], Scribe [5]). Content-based pub/sub systems are more expressive. They use subscriptions in the form of a set of predicates and evaluate them against the entire contents of the publications [7,17]. In this work, we use a variation of the content-based matching model. The key difference from the classical pub/sub work is that our predicates in the subscriptions are just words. We take advantage of this structure by making intelligent content-based routing and placement decisions in order to achieve scalability.

7.2 Wide-area network pub/sub systems

PADRES [8], SIENA [4], CORONA [19], HERMES [17], and GRYPHON[1] are well-known examples of distributed content-based pub/sub middleware that use broker overlays.

For instance, PADRES employs a network of brokers and clients to implement pub/sub functionality. Similarly, SIENA is developed as a distributed general-purpose event notification system that is composed of interconnected servers over a wide-area network. Apart from these systems, there also exist systems performing content-based data dissemination in the context of data streams, such as SemCast [16] and [9]. Compared to these works, we focus on pub/sub within a data center environment. Our system does not use brokers and instead contains multiple router and matcher operators, organized into a pipeline of data parallel stages. However, the fundamental idea behind content-based routing is valid in our approach as well. Different than the classical pub/sub problem, we have knowledge about the characteristics of the publication data and exploit it to optimize the routing.

7.3 Tightly coupled pub/sub systems

StreamHub [2], Cobra [20], and S^3-TM are pub/sub systems that are designed to be run within a data center. We refer to them as tightly coupled pub/sub systems, where scalability and high throughput are the main concerns. StreamHub resembles to our work in terms of its architectural design. However, it treats publications as black boxes during routing, and as a result, are limited to publication broadcast and subscription unicast, vice versa, or a two-level system where the broadcast/unicast roles are switched between the publications and subscriptions at successive levels. Just like the StreamHub, Cobra is also designed to be run within a data center. Cobra resembles our work in terms of the goal of the matching, as they match subscriptions to RSS feeds, enabling users to make content-based filtering and aggregation. While application-level goals of Cobra are similar to our work, the architectural design is different in terms of data parallelism. Cobra has a three-tiered architecture with crawlers, filters, and reflectors. Subscriptions are assigned to filters, and matched data are polled by the users via reflectors. Crawlers collect RSS feeds and send those to filters. However, since Cobra assumes publications as black box like StreamHub, crawlers are limited to broadcasting feeds to all filters. In contrast to StreamHub and Cobra, we take advantage of the short text matching problem domain to avoid the broadcast. Most importantly, our work focuses on optimization of the routing and placement decisions based on the contents of the publications and subscriptions, which is not covered by earlier work.

7.4 Filtering and matching

The processing heavy core of pub/sub systems involve the filtering and matching of publications against subscriptions. State-of-the-art matching algorithms for pub/sub systems fall into one of the two main categories, namely *counting-based* algorithms [4,28] and *tree-based* algorithms [1,10,21]. A counting-based algorithm maintains the number of predicates satisfied for each subscription. A tree-based algorithm organizes subscriptions as a search tree, where each node contains a predicate and each leaf has a set of subscriptions. S^3-TM uses a tree-based subscription matching algorithm as well. In our case, the search tree is a trie structure in which subscriptions can be placed within internal nodes as well.

In a recent work on matching, Shraer et al. proposed an architecture to maintain the top-k tweets relevant to a news story [24]. In their architecture, the matching between a subscription and a publication is achieved by computing a score between the contents of the two with respect to relevance and recency. This architecture limits the subscriptions to a small set of news stories. We have a different model, where subscriptions are set of words queries and matching is based on strict containment, rather than similarity.

Delta [11] is a pub/sub system where subscriptions are reorganized and rewritten to achieve low latency in matching and low resource utilization for scaling up to large numbers of subscriptions. The subscriptions are conjunctives as in our work, but they take the form of more general predicates. The system is designed considering the fact that subscriptions often overlap partially or completely. This is an assumption we also make use of. However, the authors focus mainly on reorganizing the subscriptions for efficient processing via linear programming techniques, and not on optimizing routing or placement. Our work is focused on the latter challenges and relies on a mostly traditional trie-based matching algorithm, which can be easily replaced with more advanced alternatives like delta.

8 Conclusion

In this paper, we presented S^3-TM–a system for scalable streaming short text matching. S^3-TM is designed to be run in a data center environment to evaluate high-throughput, streaming publications in the form of short posts against large number of standing subscriptions in the form of set of query terms. S^3-TM is organized as a data parallel streaming application that contains many instances of routing and matching stages. A core insight of our work is that the matching can be parallelized by using a partitioning of words over matchers. This way, publications can be multicast to a subset of relevant matchers and subscriptions can be anycast to a subset of eligible matchers. We developed several algorithms to learn a mapping that can minimize the size of the multicasts and balance the load across the matchers. Among these, the SALB algorithm that relies on the word-to-post bipartite graph has proven to be the most effective in practice. Our experimental results show that the co-occurrence relationship between words can indeed make the word partitioning-based routing

a scalable and effective solution, resulting in more than 2.5 times higher throughput compared to a baseline approach. S^3-TM also showcases good scalability. As part of this work, we have also developed a load-aware subscription placement algorithm called LASP and experimentally showed its effectiveness in taking advantage of overlap structure among subscriptions. Finally, we have introduced extensions of the base system to handle skew in the publication workload to acheive better scalability, and simple yet effective techniques for load shedding to handle unexpected spikes in load.

Acknowledgments This study was funded in part by The Scientific Technological Research Council of Turkey (TÜBİTAK) under grants EEEAG #111E217 and #112E271.

References

1. Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., Chandra, T.D.: Matching events in a content-based subscription system. In: ACM Symposium on Principles of Distributed Computing (PODC) (1999)
2. Barazzutti, R., Felber, P., Fetzer, C., Onica, E., Pineau, J.F., Pasin, M., Rivière, E., Weigert, S.: Streamhub: a massively parallel architecture for high-performance content-based publish/subscribe. In: ACM International Conference on Distributed Event-based Systems (DEBS), pp. 63–74 (2013)
3. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent dirichlet allocation. J. Mach. Learn. Res. **3**, 993–1022 (2003)
4. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and evaluation of a wide-area event notification service. ACM Trans. Comput. Syst. **19**(3), 332–383 (2001)
5. Castro, M., Druschel, P., Kermarrec, A.M., Rowstron, A.I.: Scribe: A large-scale and decentralized application-level multicast infrastructure. IEEE J. Sel. Areas Commun. **20**(8), 1489–1499 (2006)
6. Choudhury, M.D., Lin, Y.R., Sundaram, H., Candan, K.S., Xie, L., Kelliher, A.: How does the data sampling strategy impact the discovery of information diffusion in social media? In: AAAI Conference on Weblogs and Social Media (ICWSM) (2010)
7. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. **35**(2), 114–131 (2003)
8. Fidler, E., Jacobsen, H.A., Li, G., Mankovski, S.: The padres distributed publish/subscribe system. In: International Conference on Feature Interactions in Telecommunications and Software Systems (FIW) (2005)
9. Gedik, B., Liu, L.: Quality-aware distributed data delivery for continuous query services. In: ACM International Conference on Management of Data (SIGMOD) (2006)
10. Kale, S., Hazan, E., Cao, F., Singh, J.P.: Analysis and algorithms for content-based event matching. In: International Workshop on Distributed Event-Based Systems (DEBS), pp. 363–369 (2005)
11. Karanasos, K., Katsifodimos, A., Manolescu, I.: Delta: Scalable data dissemination under capacity constraints. VLDB Endow. (PVLDB) **7**(4), 217–228 (2013)
12. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
13. Li, M., Ye, F., Kim, M., Chen, H., Lei, H.: Bluedove: A scalable and elastic publish/subscribe service. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 1254–1265 (2011)
14. Liu, L., Pu, C., Tang, W.: Continual queries for internet scale event-driven information delivery. IEEE Trans. Knowl. Data Eng. **11**(4), 610–628 (1999)
15. McCallum, A.K.: MALLET: A machine learning for language toolkit (2002). http://mallet.cs.umass.edu
16. Papaemmanouil, O., Çetintemel, U.: SemCast: Semantic multicast for content-based stream dissemination. In: IEEE International Conference on Data Engineering (ICDE), pp. 37–42 (2004)
17. Pietzuch, P.R., Bacon, J.M.: Hermes: A distributed event-based middleware architecture. In: IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 611–618 (2002)
18. Porter, M.F.: An algorithm for suffix stripping. Program: electronic library and information systems pp. 313–316 (1997)
19. Ramasubramanian, V., Peterson, R., Sirer, E.G.: Corona: a high performance publish-subscribe system for the world wide web. In: USENIX Conference on Networked Systems Design and Implementation (NSDI) (2006)
20. Rose, I., Murty, R., Pietzuch, P., Ledlie, J., Roussopoulos, M., Welsh, M.: Cobra: Contentbased filtering and aggregation of blogs and rss feeds. In: USENIX Conference on Networked Systems Design and Implementation (NSDI), pp. 3–3 (2007)
21. Sadoghi, M., Jacobsen, H.A.: Be-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space. In: ACM International Conference on Management of Data (SIGMOD), pp. 637–648 (2011)
22. Schaeffer, S.E.: Survey: graph clustering. Comput. Sci. Rev. **1**(1), 27–64 (2007)
23. Schneider, S., Hirzel, M., Gedik, B., Wu, K.L.: Safe data parallelism for general streaming. IEEE Trans. Comput. (2013). doi:10.1109/TC.2013.221
24. Shraer, A., Gurevich, M., Fontoura, M., Josifovski, V.: Top-k publish-subscribe for social annotation of news. VLDB Endow. (PVLDB) **6**(6), 385–396 (2013)
25. Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M., Stonebraker, M.: Load shedding in a data stream manager. In: Very Large Databases Conference (VLDB), pp. 309–320 (2003)
26. TIBCO Inc., Tib/rendezvous. White Paper (1999)
27. Twitter Streaming API. http://dev.twitter.com/docs/streaming-apis. Retrieved Dec (2013)
28. Yan, T., Garcia-Molina, H.: Index structures for selective dissemination of information under the boolean model. ACM Trans. Database Syst. **19**(2), 332–364 (1994)