Tracking Hot-k Items over Web 2.0 Streams◇

Parisa Haghani * Sebastian Michel ‡ Karl Aberer *

* Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
parisa.haghani@epfl.ch, karl.aberer@epfl.ch

‡ Universität des Saarlandes, Saarbrücken, Germany
smichel@mmci.uni-saarland.de

Abstract: The rise of the Web 2.0 has made content publishing easier than ever. Yesterday's passive consumers are now active users who generate and contribute new data to the web at an immense rate. We consider evaluating *data driven* aggregation queries which arise in Web 2.0 applications. In this context, each user action is interpreted as an event in a corresponding stream e.g., a particular weblog feed, or a photo stream. The presented approach continuously tracks the most popular tags attached to the incoming items and based on this, constructs a dynamic top-k query. By continuous evaluation of this query on the incoming stream, we are able to retrieve the currently *hottest* items. To limit the query processing cost, we propose to pre-aggregate index lists for parts of the query which are later on used to construct the full query result. As it is prohibitively expensive to materialize lists for all possible combinations, we select those tag sets that are most beneficial for the expected performance gain, based on predictions leveraging traditional FM sketches. To demonstrate the suitability of our approach, we perform a performance evaluation using a real-world dataset obtained from a weblog crawl.

1 Introduction

The world has turned into one large-scale interconnected information system with millions of users. End users, with the advent of Web 2.0, are now content generators who actively contribute to the Web. User generated data is usually in form of semi-structured text like personal blog entries with categorization [1] or images and videos annotated with tags [Fli, You]. Each user action, for example uploading a picture, tagging a video or commenting on a blog, could be interpreted as an event in a corresponding stream. Data stream processing has gained a lot of attention in the recent years (see [BBD$^+$02, Mut05] for surveys), since many of today's applications are best captured in this model. Data items in different formats stream in to a processing unit where each item has the chance of being

◇ This work is partially supported by NCCR-MICS (grant number 5005-67322), the FP7 EU Project OKKAM (contract no.ICT-215032), and the German Research Foundation (DFG) Cluster of Excellence "Multimodal Computing and Interaction" (MMCI).

[1] http://google.blogspace.com/, http://www.weblogs.com/, http://www.blogger.com/

seen once before being archived for later uses. While this model has been successfully applied in scenarios such as sensor networks, traffic monitoring and financial data feeds, Web 2.0 generated data has less frequently been treated as streams. Most data mining approaches on this ever growing source of data run their analysis algorithms in an offline fashion [KNRT05, Kle02, HJSS06], hence disregarding the live nature of the web.

Given the immense volume of data being published on the web and the desire of consuming newly published data, there is an increasing need for processing this information in real time in efficient ways. All this gives rise to considering this data in a streaming model.

As an *example* of temporal streams of information in a Web 2.0 application consider published content in form of news articles or posts on personal weblogs (blogs). Explicit temporal annotations (i.e. *written at*, *uploaded at*) of the content of weblogs or news portals makes them natural items of a temporal stream. Mechanisms such as RSS and atom are used to notify users of newly published data on their favored weblogs or news portals. The items in a blog feed stream are generated at distributed sources depending on the subscriptions which are made by the user. The large body of information retrieval techniques can be used in order to extract categories or topics from the published text [APL98, ACD+98].

We consider online processing of aggregation queries over streaming data where each data item carries a particular set of tags with it. We aim at monitoring the hottest items at each time by defining a top-k query of currently popular tags. Hot items are subsequently defined as those items which have high score with regard to the defined query.

1.1 Problem Statement and Contribution

We consider a stream $\mathcal{S}$ of tagged items where each item has the following format:

$$d =< itemId, time, \mathcal{T}_d >$$

$itemId$ is a unique identifier specifying the object this item is describing, i.e. URL of an image or post, and $time$ represents the time when d was produced. Let $\mathcal{T} = \{t_1, ..., t_n\}$ be the global set of tags which are used to annotate items. $\mathcal{T}_d \subset \mathcal{T}$ is the set of tags with which d is annotated. The number of tags an item carries is usually very small (e.g., around 5) compared to standard document retrieval where a text document contains lots of terms. For each tag we assume a given score $score(d, t)$ that reflects the relatedness of the item to the tag.

We further assume *in-order* streams; items arrive in the same order that they are generated. In most streaming scenarios, as well as ours, recent items are of more interest than old ones. This is captured by the sliding window model. A sliding window (W) is assumed over the stream and items are considered *valid* while they belong to this window. Sliding windows can be either count or time based, i.e., bounding the number of items either by count or focusing only on those that occurred in a particular time interval.

At each point in time we can compute statistics over the tags used in items currently in the sliding window W or compute aggregation queries over these items. This view forms the basis of our approach, which builds on statistics on tag usages to determine a set of

popular tags. This tag set is then interpreted as a continuous and dynamic keyword query which is executed against the sliding window as time evolves. We call this query dynamic as it is re-build with evolving time due to changes in tag popularities.

Definition 1 Hot Tags *and* **Hot Items:** *At each timestamp τ, the set of hot tags (H_τ) consists of the c tags with the highest popularity.*

The set of hot tags defines the query we use to rank the valid items, i.e., the query is data-driven and changes with time as the popularity of tags changes. For a valid item, we define its current score as the sum of scores of the hot tags it carries. More formally,

$$s(d, H_\tau) := \sum_{t \in H_\tau \cap T_d} score(d, t)$$

The task is to continuously compute the top-k items as the query changes. In contrast to standard top-k query processing over text (or XML) documents, here, the query is supposed to be rather big to capture not only a few but many hot topics for diversity reasons. In summary, the considered tags (features, in standard IR terminology) is small whereas the query is long, which is in clear contrast to traditional query processing techniques.

In this work we focus on efficiency aspects and the potential of pre-aggregations and how to decide which subqueries to pre-compute. For the actual decision which tags should be considered in as query terms, one can think of other measures than the pure popularity count based methods we use in our work, e.g., methods that aim at identifying trending (hot) topics.

In this paper we make the following contributions. We show how to continuously compute the set of hot items over social (Web 2.0) data streams by defining a dynamic top-k aggregation query and show how pre-aggregations of popular sub queries can be used to efficiently process the query. We evaluate our proposed methods on a real-world dataset of blog posts showing the suitability of our approach.

This paper is organized as follows. Section 2 presents the related work. Section 3 briefly describes the general structure that we consider in this paper together with a baseline algorithm. Section 4 describes the problem of pre-aggregating groups of index lists for efficient query processing and presents next to an offline problem definition an efficient and effective approximation for online processing. Section 5 presents the experimental evaluation. Section 6 concludes the paper.

2 Related Work

Data stream processing has been a hot topic in the past years as many of todays applications require real-time processing of dynamic data. For comprehensible surveys of this topic in general see [BBD$^+$02, Mut05]). Early works mostly consider one-pass algorithms in limited space over the whole stream where all tuples are considered valid at all times. A related problem to ours is reporting on *quantiles* or *heavy hitters* in streams. The goal

is to report on most repeated items in the stream, when the number of items is so high that keeping statistics for each is not possible. Approximate solutions to this problem exist which make use of techniques such as the famous AMS sketches [AGMS02], or more recently group testing, see for example [CCFC04, CM03] and the references within. In our work, the number of tags we consider and desire to know the hottest amongst is small enough such that exact statistics could be kept for each.

Another line of research in stream processing is dedicated to top-k query answering in data streams. Mouratidis et al. [MBP06] maintain a skyline [BKS01] which represents the possible top-k candidates. Their solution is optimized for *fixed* queries and they focus on changes introduced by items timing out or new items arriving in. In a more general setting, [DGKS07] proposes indexing methods for answering *adhoc* top-k queries based on arrangements. While our queries can not be considered as *fixed* (as the set of hot tags changes over time with new items arriving) they are not completely *adhoc* either. We exploit this fact to pre-aggregate parts of the query which can be used several times in future queries. Jin et el. [JY$^+$08] consider top-k queries on uncertain streams where the data items are associated with existential probabilities. In our envisioned applications all items are certain.

Mainly motivated by the wealth of news feeds and other online information streams, another related problem is *Topic Detection and Tracking* (TDT) which has been extensively studied in the past few years [APL98, ACD$^+$98, HCL07]. The goal here is to detect new events appearing in the data stream and tracking those events in order to later identify data which further discuss the same event. Another related topic is mining frequent itemsets in a data stream. In a recent work Calders et al [CDG07] define a new measure as the frequency of an itemset and propose an incremental algorithm that allows for reporting the exact frequencies of frequent itemsets. The problem of itemset mining is orthogonal to our problem and can be used to improve the quality of our choice of pre-aggregation queries. In another line of research related to Web 2.0 applications with temporal considerations, Hotho et al. [HJSS06] consider discovering topic-specific trends in folksonomies which are collections of resources tagged by users (such as Flickr or del.icio.us [2]). Their analysis is based on the famous PageRank algorithm. They perform the algorithm in an offline manner and assume the whole corpus of data to be available. Weblog evolution is considered in [KNRT05], where *time graphs* are introduced and used for community tracking again in an offline mode. In [MK09], the goal is to identify weblogs defined as *starters* and *followers* specified by certain linking relations in an efficient way. In contrast to the above, we continuously evaluated the data as it arrives in an online manner. For a survey of temporal data analysis methods see [Kle06] and the references within.

Keeping the query results updated as data streams in with high rates requires high performance evaluation of top-k queries. One way to improve the performance of expensive queries is to maintain their results as materialized views. In order to avoid reprocessing a top-k query in face of updates in the database, such as insertions or deletions, authors in [YYY$^+$03] suggest maintaining a top-k' view, where $k' > k$ and show how to choose k' dynamically to adapt to the system workload. In [HKP01] authors investigate answering a top-k query based on the materialized results of another top-k query where the preference

[2]http://del.icio.us

function is a linear combination of all attributes of tuples. It is shown how to decide given a preference function and it's top-l results if the top-1 result of another preference function can be found in these materialized l tuples. In [DGKT06], the TA algorithm is adapted to the case where a set of views, not necessarily the single inverted lists, are available. The views are visited in a lock-step manner and in each iteration the maximum score of unseen tuples are calculated by a linear programming optimization, given the preference functions of each of the views. Given a set of views, the best subset for answering a query is chosen based on a process simulating the TA utilizing the data distributions in each view. In the same line, [KPSV09], investigate top-k query processing when intersection of single inverted lists are also available. A combinatorial solution is proposed to solve the specific linear program appearing when the set of lists consist of only single or intersection of two single lists. A very interesting result of the paper is that in order to guarantee instance optimality all available lists should be investigated. In a streaming scenario however, maintaining the intersection of all pairs of single lists is not possible due to memory constraints. In this work we propose to maintain the intersection of several lists instead of just pairs of them and we chose the intersections based on the benefits they potentially have for future data-driven top-k queries.

3 System Model and Structure

In this section we briefly describe the general structure that we consider. As mentioned in Section 1.1 we consider one data stream as the input to our system where the items in this stream contain a list of tags and they are considered valid while belonging to a sliding window.

We assume all valid items are sorted in a first-in-first-out list. This provides an efficient mechanism for evicting expired items. Newly arriving items in the stream are placed at the head of this list and old items are dropped from the tail. In addition to the time sorted list, we maintain a hash index on the valid items that point to the set of their tags. Furthermore, for each tag, we keep a sorted list of items that have been annotated with this tag. Let l_i represent the list maintained for tag t_i. l_i is sorted based on t_i's score for each item in descending order. When an item expires, it is also removed from the sorted list it belongs to. Considering newly arriving items is easily achievable as it causes only insertions to a few lists plus one insertion to the hash index and the time sorted list, as described above. Note that as opposed to standard top-k processing where each document has potentially very many features (terms), here, the average number of tags per item is rather small. As a result updating the structures with new arrivals does not incur high cost.

For basic query execution we employ the threshold algorithm (TA) [Fag02], which works as follows. It reads in parallel from the index lists, which are sorted by score in descending order. For each item observed it looks up its score in all other lists it has not been observed so far, which is done in our case with one lookup to the hash map as described in the previous paragraph. The aggregated scores of the items at the current sequential access scan depth define the stopping condition. The computation can be stopped if there are at least k items with a score better than the aggregated score at the sequential scan lines. We

employ the TA algorithm over the single term index lists as our baseline algorithm.

The top-k query needs to be re-evaluated in two cases: first, when an item which was part of the top-k results expires. The second case happens when the set of top tags changes and causes a change in the query aggregation function. In order to avoid re-computations from scratch when a hot item expires, a k-skyband over the score-time space can be kept [MBP06]. The k-skyband of a query contains only those items which have a chance of becoming a top-k result during their life time. When an item which was part of the top-k results expires, it is enough to evaluate the query on the k-skyband, instead of the entire valid items, to fill in the top-k results. This dramatically decreases the cost of re-evaluations, however, it is only useful when the query remains unchanged. For the rest of this paper we do not consider possible optimizations when the top-k query is not changing, as this is a well addressed problem [MBP06, DGKS07], rather, we will focus on solutions for the changing query issue. In the next section we describe our approach for pre-aggregating stable parts of the top-k query in order to decrease the cost of evaluations when the query changes.

4 Grouping for Pre-Aggregation

Observing the changes in the top-k query itself, which is considered to be quite large (~ 100 tags), shows that although the query itself changes more or less every time, there is a fraction of tags that remain as part of the query for a long duration of time. These consists of those tags which are popular most of the time and represent current long-lived events. Observing stable sub-queries, motivates us to maintain pre-aggregations for those sub-queries which can later be used to evaluate the complete query more efficiently.

In this section we propose to group lists corresponding to "stable" tags together to reuse their aggregated results. More precisely, we pre-aggregate certain lists and try to assemble at query time the final top-k result given the pre-aggregated values.

4.1 Optimal Solution

To better understand the complexity of the problem, in this section we formulate an offline algorithm. The offline algorithm assumes a finite stream and complete knowledge over incoming data. Therefore the set of different top-k queries for a given time period is known to the offline algorithm.

Given a set of queries $\mathcal{Q} = \{Q_1, Q_2, ..., Q_n\}$, the goal is to find an optimal set of subsets of tags $\mathcal{S}$ that can answer all queries in Q efficiently, re-using pre-aggregations in $\mathcal{S}$. Each member of $\mathcal{S}$ is a subset of tags and if its cardinality is larger than one, represents a pre-aggregation of the lists maintained for the tags it contains. For example $\mathcal{S} \ni S_i = \{t_j, t_k\}$ means we are maintaining a sorted list for $t_j \vee t_k$. Let L_i represent the list corresponding to S_i. Items in L_i are sorted based on their score with regard to S_i:$s(d, S_i) := \sum_{t \in S_i \cap T_d} score(d, t)$.

In case of ties, more recent items are preferred. L_i is created utilizing the simple lists we maintain for tags which are members of S_i. Assuming equal length l for all simple lists, the cost of aggregating k such lists is $k * l$.

Now assume a query Q_y. Recall that each query is specified by a set of tags. We say a subset $S'_y \subset \mathcal{S}$ *exactly covers* Q_y if members of S'_y are pairwise disjoint and $\bigcup_{S_i \in S'_y} S_i = Q_y$. If a subset exactly covers a query, a standard TA algorithm can use it to evaluate that query. The effectiveness of a list L_i depends on the co-occurrences of tags in S_i in the stream of items. We assume the percentage of items likely to be read before TA can stop is known for a list L_i and we denote it by c_i. Note that c_i depends on the query and other available lists, but for simplicity we consider it as an independent fixed value. The cost of evaluating a query Q_y using S'_y can be estimated by: $\sum_{S_i \in S'_y} c_i$.

Let $\mathcal{P}$ be the powerset of $\bigcup Q_i$. Given the above cost functions, we can formulate our goal as an optimization problem which aims at minimizing the following cost function with regard to the boolean variables x_{ij}:

$$\sum_{S_i \in \mathcal{P}} y_i * |S_i| * l + \sum_{Q_j \in \mathcal{Q}} x_{ij} * c_i$$

and the following constrains:

$$\begin{cases} y_i = \bigvee_i x_{ij} & (C1) \\ \forall Q_j \forall t \in Q_j \sum_{i:t \in S_i} x_{ij} = 1 & (C2) \end{cases}$$

x_{ij} = 1 shows that S_i is used in evaluating Q_j. $y_i = 1$ if S_i is used in evaluating at least one query. The first constraint (C1) assures this. The first summation in the cost function accounts for the pre-aggregation expenses while the second part shows the evaluation cost. The second constraint (C2) ensures that the set of S_i's used for evaluating each query exactly cover that query.

The above optimization problem is not a standard linear programming problem, as the variables y_i depend on x_{ij}'s. However, even if we ignore the first part of the cost function (the query evaluation cost), we face a 0-1 linear programming problem which is known to be NP-hard (cf., e.g., [MS08]).

4.2 Efficient Grouping

Given the complexity of the problem described above and the fact that the set of future top-k queries is actually not known in advance, we address the problem with an approximate approach.

Clearly it is beneficial to pre-aggregate sets of tags which frequently appear in the future top-k queries: Aggregating the corresponding lists of a set of tags pays off only when the resultant list can be used enough number of times in future queries. For each observed

tag we maintain the number of times it has appeared in the set of hot tags and predict its probability of being part of the aggregation query based on this past information.

Assume the number of single tags with probability of appearing in future queries larger than a specific threshold is r. These tags have to be grouped together to form a pre-aggregated list. However, grouping all of them together may not be beneficial, as to be able to use such a pre-aggregation *all* involved tags should be part of the query. The probability that a pre-aggregation of m single lists is usable in future queries, decreases with increasing m: if p is the probability of the most frequent tag, and assuming tags appear independent of one another, p^m is an upper bound of the probability that this aggregation list is usable. We should therefore, pre-aggregate *subsets* of the r candidate tags.

Grouping those tags which co-occur together in the streaming items is highly beneficial for the overall performance as they have higher chances of appearing *together* in future queries. Given the data-driven nature of the query, the query evaluation using the TA algorithm can be done more efficiently due to the already pre-aggregated partial queries. A pre-aggregation of tags which do not co-occur together and aggregating them creates a list of size of sum of the sizes of single lists with non-aggregated scores. On the other hand, aggregating single lists which have high correlation, i.e., their corresponding tags occur together, results in a list with more score variations (in case of ties in the original list) and higher scores, which is more effective in decreasing the threshold value maintained by the TA algorithm and causing it to stop reading more entries.

As a measure of tag co-occurrence we calculate the resemblance value for two index lists, which is defined as the fraction of the size of their intersection over the size of their union. Based on the given intuitions above, in the next section we describe our proposed algorithm for selecting tag sets to be materialized.

4.3 Tag Set Generation

To actually compute the tag sets to be materialized our algorithm considers all tags that frequently occurring in the queries with a probability above a parameter α. Since the cardinality of the set of tags is not large, we can maintain exact statistics for the number of occurences of each tag in a query. We normalize the number of occurences and use it as the probability of tag's occurance in future queries.

Based on this group of tags we generate the tag sets of interest in the following way, illustrated in Figure 1:

1. each tag is considered to be a node of a graph
2. for each pair of tags the resemblance is calculated
3. each pair of tags with resemblance $\geq \rho$ is treated as an edge in a graph
4. the connected components of the graph are sets of tags to be materialized

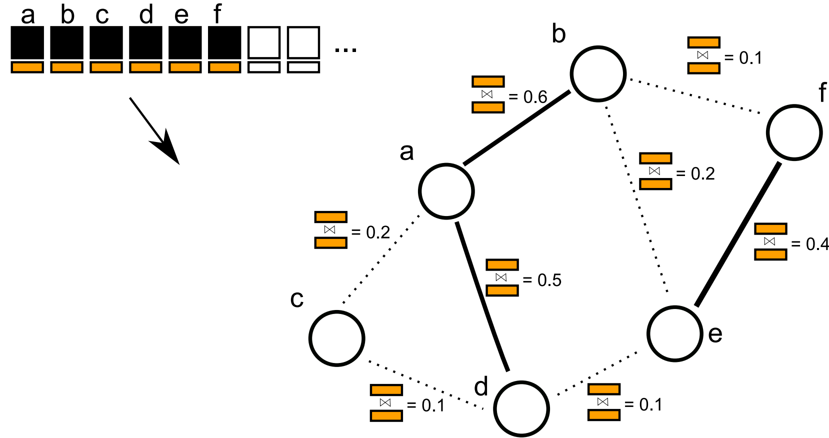

Figure 1: Illustration (showing a subset of all cases to be considered) of the process of determining tag sets of interest for pre-aggregation for $\rho = 0.3$. Given the top re-occurring tags as the nodes in a graph, we connect those nodes whose index lists have a resemblance of at least 0.3. All resulting connected components are then selected and the corresponding index lists pre-aggregated.

This technique favors those frequently reoccurring parts of the query that also frequently appear *together* in the data stream.

4.4 FM Sketches for Resemblance Calculation

As the computation of the exact resemblance is extremely expensive we employ a sketching technique that can efficiently estimate the resemblance value independent of the size of the involved index lists. In addition, as even exact resemblance numbers cannot guarantee the optimal pre-aggregation, the effect of slightly inaccurate resemblance numbers are negligible.

We make use of the well known Flajolet-Martin sketches (FM sketches) [FM85], which are compact and precise estimators of the cardinality of a multi-set. Given two sets S_1 and S_2 and their corresponding synopses in form of FM sketches, once can determine the size of the intersection by combining the sketches in an extremely efficient bit-wise fashion. More precisely, one obtains actually the size of the union given the bit-wise OR operation of the bit-sets of the two sketches. Then, the size of the intersection is given by the inclusion-exclusion principle ($|S_1 \cup S_2| = |S_1| + |S_2| - |S_1 \cap S_2|$), hence we can estimate the resemblance value.

As we keep index lists for the tags we observe, there is only the small overhead of maintaining a sketch for each of these lists. When enumerating the candidate tag sets we estimate their suitability to the query processing based solely on the sketches. There is no need to compute the aggregation and assess its size as the size is directly given by the

sketch combinations, which is very efficient.

Due to the inherently approximate nature of the sketches, the resemblance values are not exact, which leads to decisions of which tag sets to materialize that varies from the algorithm employing the true resemblance numbers.

5 Experiments

We have implemented our algorithm in Java 1.6 and executed on a Windows 2003 server with a quad core 2.33 GHz Intel Xeon CPU, 16GB RAM, and a 800GB RAID-5 disk.

We have obtained the ICWSM 2009 Spinn3r Blog Dataset[3]. It consists of 44 million blog posts between the time period of August 1st and October 1st, 2008. Each blog entry (post) consists of plain text, a timestamp, a set of tags, and other meta information such as the blog's homepage URL etc. The data is formatted in XML and is further arranged into tiers approximating to some degree search engine ranking. We have parsed the blog posts for the highest tier levels resulting in $11,395,571$ (timeStamp, postId, tags)-entries, with $2,444,780$ distinct postIds, hence, an average of ~ 2.2 tags per blog entry. For the score of a document w.r.t. a particular tag we simply consider score 1 if the tag is attached to the document, 0 otherwise. While in principle a measure like *tag frequency* would be more suitable, the way the dataset is generated limits us to the boolean values.

Algorithms

We consider the performance of three algorithms in this experimental evaluation. All are based on the TA algorithm [Fag02]. The difference stems from the index lists they can involve in the query processing. More precisely, we run the following algorithms:

- **plain:** This is the plain algorithm involving only accesses to single-tag index lists.
- **comb:** This algorithm uses pre-aggregation of tag sets that are supposed to help the query execution. The set of tags to be pre-aggregated are chosen using the algorithm described in Section 4.3. True resemblance values are calculated by merging lists and measuring the resultant size.
- **combsketch:** This algorithm also uses pre-aggregation tag sets as described in Section 4.3. However, the resemblance values are estimated using sketches as described in Section 4.4.

Note that the comb algorithm is in fact impractical, as it incurs huge costs just for measuring the resemblance values. However we ignore this cost and use this algorithm to show the best achievable performance using our proposed set aggregation method.

[3] http://www.icwsm.org/2009/data/

Measures of Interest

We will report on several measures as part of our performance study. Note that we do not report on accuracy measure as *all algorithms report the exact top-k results* to the query described above. We consider the number of entry accesses as the main cost to assess the suitability of the methods under comparison. We split this measure up in several ingredients to better understand the strong and weak points of the approaches. In particular for the algorithms that use pre-aggregation, the cost for materializing lists for sets of tags does not occur in each query processing step. We measure:

- **eval_cost:** This measure reports on the average number of entry accesses the threshold algorithm makes to calculate the results.
- **pre-aggregation_cost:** With this measure we provide an insight on how costly the pre-aggregation operation is, that means, how many entries on average need to be accessed when materializing the index lists for sets of tags, determined by the selection algorithm. The plain algorithm does not incur any pre-aggregation cost.
- **total_cost:** In addition to the measures described above we also report on the total cost which consists of the total (non-averaged) cost for all query evaluations plus the overall cost for doing the pre-aggregation. We ignore the cost for calculating the resemblance values.

Results

We run the mentioned three algorithms for different parameter settings averaging over 45 query evaluations for each setting. The query evaluation is fired at every 500 items. The tag set generation algorithm (described in Section 4.3) is run periodically at every 20 evaluations. Unless otherwise stated we use a time-based sliding window of size $W = 10,000,000$ milliseconds. The default number of desired top-k items denoted by $kdocs$ is 100. The number of tags used in defining the query is denoted by $ctags$ and its default value is set to 75.

We first observe the effects that parameters α and ρ have on the costs incurred by our proposed algorithms. Figures 2,3, and 4 shows the different cost values while varying the parameter α and fixing all other parameters. As explained in Section 4.3, α denotes the threshold for considering a tag for the subsequent tag set generations. Figure 2 presents the evaluation cost with changing α. Small α values causes the algorithm to consider tags which actually do not occur later in the query. These tags may have high enough resemblance with other tags to be part of a connected component. The tag set corresponding to such a component is however, useless, since it contains a tag which does not actually appear in the query. As a result, both comb and combsketch have total costs close to plain with small α values. On the contrary, for large enough values of α a large fraction of materialized lists are actually reusable, therefore the evaluation cost of comb and combsketch is much smaller than plain. For too high values of α, less than necessary number of

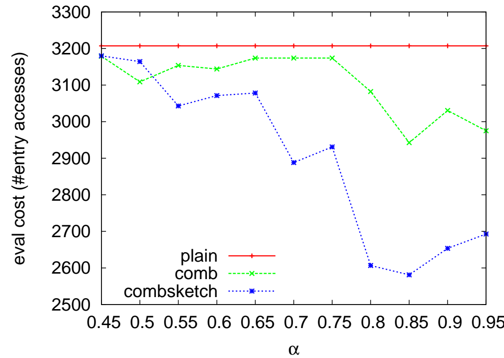

Figure 2: Eval_cost values when varying the α parameter. W=10,000,000ms, kdocs=100, ctags=75, $\rho = 0.6$

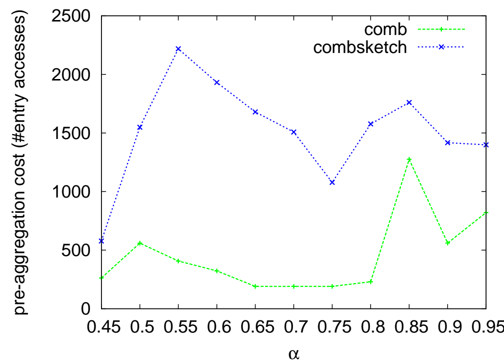

Figure 3: Pre-aggregation_cost values when varying the α parameter. W=10,000,000ms, kdocs=100, ctags=75, $\rho = 0.6$

tags are actually considered, lowering the total benefits of them in evaluating the queries. The pre-aggregation_cost is shown in Figure 3. We see that for $\alpha = 0.85$ both comb and combsketch have high pre-aggregation_cost which actually pays off very well, as the total cost at this value has a minimum for both methods.

Figure 7 shows the total costs when varying the parameter ρ, which specifies whether or not an edge should be considered between two nodes in the tag set generation algorithm. In our experiments ρ is not an absolute value, as the resemblance values estimated by combsketch and sketch are very different in the absolute sense but they usually hold the same ordering: if a list l_1 has higher true resemblance to l_2 than l_3 this likely holds also

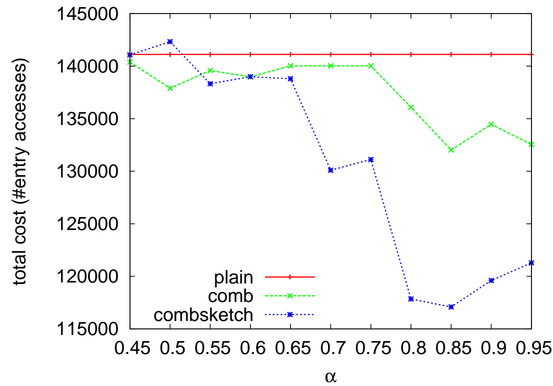

Figure 4: Total_cost values when varying the α parameter.W=10,000,000ms, kdocs=100, ctags=75, $\rho = 0.6$

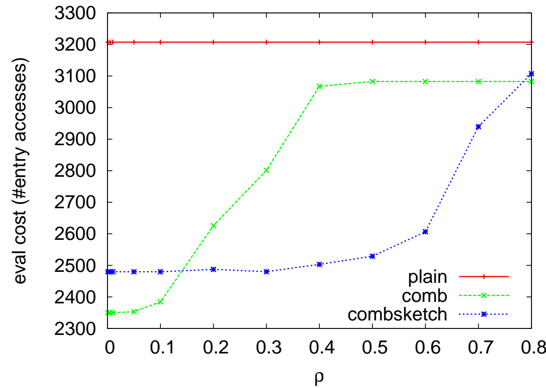

Figure 5: Eval_cost values when varying the ρ parameter. W=10,000,000ms, kdocs=100, ctags=75, $\alpha = 0.8$

in the estimated values by combsketch. So we calculate the highest resemblance value res_{max} and $\rho * res_{max}$ is the threshold considered. We repeat the same procedure for $\rho + step$, each time increasing the resemblance threshold until it reaches 1. This way, we produce smaller tag sets which have high resemblances. So, as observed also in Figure 6 the pre-aggregation cost decreases by increasing ρ. Note that the ρ value were the pre-aggregation cost is actually paid off in evaluations is different for comb and combsketch.

After discovering good parameters for our algorithms, we evaluate our methods by fixing

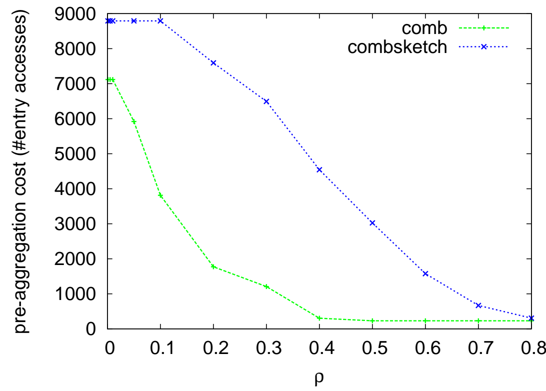

Figure 6: Pre-aggregation_cost when varying the ρ parameter. W=10,000,000ms, kdocs=100, ctags=75, $\alpha = 0.8$

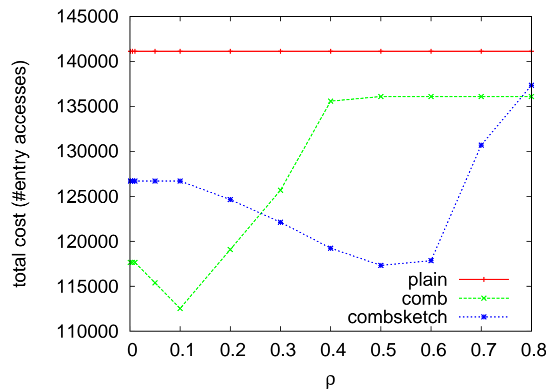

Figure 7: Total_cost when varying the ρ parameter. W=10,000,000ms, kdocs=100, ctags=75, $\alpha = 0.8$

those parameters to the best found, and changing the system variables. Figure 8 shows the total cost incurred by the three algorithms when changing the size of the sliding window. Clearly the cost for all three methods increases, as more items are valid at each instance of time, therefore the lists to be accessed are longer. However our algorithms incur much less cost than the plain algorithm. Figure 9 shows the same measure when changing kdocs. As expected the TA algorithm can stop earlier for smaller values of kdocs. Figure 10 finally, shows the total cost when varying ctags. Since this number defines the number lists we

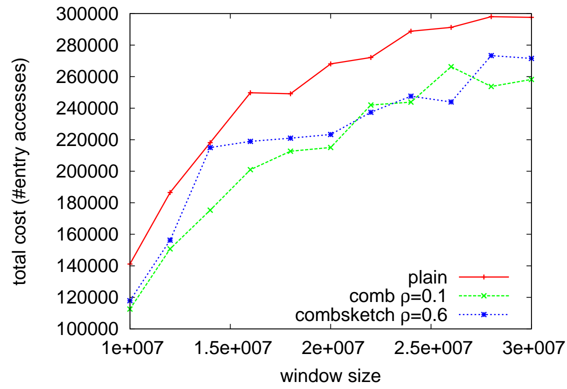

Figure 8: Total_cost when varying the window size, $\alpha = 0.8$, kdocs =100 and ctags =75

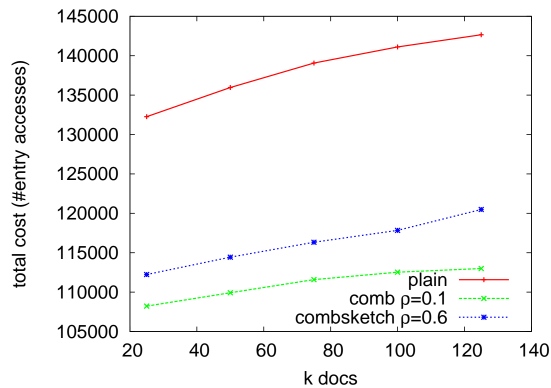

Figure 9: Total_cost when varying kdocs, $\alpha = 0.8$, W=10,000,000ms and ctags=75

should consider in the evaluation, it has a direct effect on total cost. In all three cases, our proposed algorithms incur less cost than the plain method. Although combsketch has only estimates of the true resemblances, its performance gains is very close to comb which has the true resemblance values.

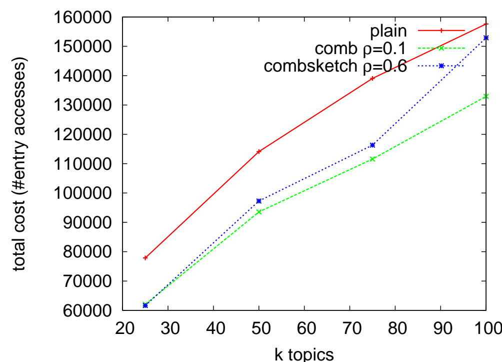

Figure 10: Total_cost when varying ctags, $\alpha = 0.8$, W=10,000,000ms and kdocs=75

6 Conclusion

We addressed the problem of continuous monitoring of top-k hottest items over a stream of tagged items such as blog entries or images. We have defined the property of being hot as a top-k aggregation query where the query itself is characterized by the set of most popular tags in a given time period. This causes the top-k query to change over time, hence requires the system to re-evaluate the top-k query from scratch. Our approach is based on the observation that parts of the top-k query are stable for certain time intervals, therefore, do not have to be re-computed in each evaluation phase. As materializing pre-computations of all possible subsets is impractical, we have presented an approximate algorithm to identify the most promising tag subsets (i.e., top-k query ingredients) leveraging FM sketches to predict the suitability of these tag sets. The presented generation method itself gives an easy to use mean to control the amount of pre-aggregated lists.

References

[ACD+98] James Allan, Jaime Carbonell, George Doddington, Jonathan Yamron, and Yiming Yang. Topic Detection and Tracking Pilot Study Final Report, 1998.

[AGMS02] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking Join and Self-Join Sizes in Limited Storage. *J. Comput. Syst. Sci.*, 64(3):719–747, 2002.

[APL98] James Allan, Ron Papka, and Victor Lavrenko. On-Line New Event Detection and Tracking. In *SIGIR*, pages 37–45, 1998.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *PODS*, pages 1–16, 2002.

[BKS01] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The Skyline Operator. In *ICDE*, pages 421–430, 2001.

[CCFC04] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004.

[CDG07] Toon Calders, Nele Dexters, and Bart Goethals. Mining Frequent Itemsets in a Stream. In *ICDM*, pages 83–92, 2007.

[CM03] Graham Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. In *PODS*, pages 296–306, 2003.

[DGKS07] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Nikos Sarkas. Ad-hoc Top-k Query Answering for Data Streams. In *VLDB*, pages 183–194, 2007.

[DGKT06] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. Answering Top-k Queries Using Views. In *VLDB*, pages 451–462, 2006.

[Fag02] Ronald Fagin. Combining Fuzzy Information: an Overview. *SIGMOD Record*, 31(2):109–118, 2002.

[Fli] Flickr Photo Sharing: http://www.flickr.com.

[FM85] Philippe Flajolet and G. Nigel Martin. Probabilistic Counting Algorithms for Data Base Applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[HCL07] Qi He, Kuiyu Chang, and Ee-Peng Lim. Analyzing feature trajectories for event detection. In *SIGIR*, pages 207–214, 2007.

[HJSS06] Andreas Hotho, Robert Jäschke, Christoph Schmitz, and Gerd Stumme. Trend Detection in Folksonomies. In *SAMT*, pages 56–70, 2006.

[HKP01] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. In *SIGMOD Conference*, pages 259–270, 2001.

[JY$^+$08] Cheqing Jin, Ke Yi, Lei Chen 0002, Jeffrey Xu Yu, and Xuemin Lin. Sliding-window top-k queries on uncertain streams. *PVLDB*, 1(1):301–312, 2008.

[Kle02] Jon M. Kleinberg. Bursty and hierarchical structure in streams. In *KDD*, pages 91–101, 2002.

[Kle06] Jon Kleinberg. Temporal dynamics of on-line information streams. In *In Data Stream Management: Processing High-Speed Data*. Springer, 2006.

[KNRT05] Ravi Kumar, Jasmine Novak, Prabhakar Raghavan, and Andrew Tomkins. On the Bursty Evolution of Blogspace. *World Wide Web*, 8(2):159–178, 2005.

[KPSV09] Ravi Kumar, Kunal Punera, Torsten Suel, and Sergei Vassilvitskii. Top-k aggregation using intersections of ranked inputs. In *WSDM*, pages 222–231, 2009.

[MBP06] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD Conference*, pages 635–646, 2006.

[MK09] Michael Mathioudakis and Nick Koudas. Efficient identification of starters and followers in social media. In *EDBT*, pages 708–719, 2009.

[MS08] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[Mut05] S. Muthukrishnan. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[You] Youtube, Broadcast Yourself: http://www.youtube.com/.

[YYY+03] Ke Yi, Hai Yu, Jun Yang, Gangqiang Xia, and Yuguo Chen. Efficient Maintenance of Materialized Top-k Views. In *ICDE*, pages 189–200, 2003.