

# Ad-Hoc Aggregations of Ranked Lists in the Presence of Hierarchies

Nilesh Bansal  
University of Toronto  
nilesh@cs.toronto.edu

Sudipto Guha  
University of Pennsylvania  
sudipto@cis.upenn.edu

Nick Koudas  
University of Toronto  
koudas@cs.toronto.edu

## ABSTRACT

A variety of web sites and web based services produce textual lists at varying time granularities ranked according to several criteria. For example, Google Trends produces lists of popular query keywords which can be visualized according to several criteria. At Flickr, lists of popular tags used to tag the images uploaded can be visualized as a cloud based on their popularity. Identification of the  $k$  most popular terms can be easily conducted by utilizing well known *rank aggregation* algorithms.

In this paper we take a different approach to information discovery from such ranked lists. We maintain the same rank aggregation framework but we *elevate* terms at a higher level by making use of popular *term hierarchies* commonly available. Under such a transformation we show that typical early stopping certificates available for rank aggregation algorithms are no longer applicable. Based on this observation, in this paper, we present a probabilistic framework for early stopping in this setting. We introduce a relaxed version of the rank aggregation problem involving a deterministic stopping condition with user specified precision. We introduce an algorithm  $pH - RA$  for the solution of this problem. In addition we introduce techniques to improve the performance of  $pH - RA$  even further via precomputation utilizing a sparse set system. Through a detailed experimental evaluation using synthetic and real datasets we demonstrate the efficiency of our framework.

## Categories and Subject Descriptors

H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval

## General Terms

Algorithms

## Keywords

List Aggregation, Top-K

## 1. INTRODUCTION

A variety of web sites and web based services produce textual lists at varying time granularities ranked according to several criteria.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.  
Copyright 2008 ACM 978-1-60558-102-6/08/06 ...\$5.00.

For example, Google Trends produces lists of popular (ranked according to frequency) query keywords which can be visualized according to several criteria (e.g., geographic restrictions etc). At Flickr, lists of popular tags used to tag the images uploaded can be visualized as a cloud based on their popularity. Web analytics companies such as Alexa produce ranked list of websites according to their traffic rank. Such ranked lists of text strings, depending on the context can be very informative. For example, in the case of popular query strings, they reveal popular searches and in some sense track the current interest of web searchers. In the case of popular blog tags, they reveal popular topics in the blogosphere (under the assumption that tags track blog post topics, of course). In the case of images, popular tags reveal popular events/places or locations associated with the images.

The blogosphere, being an unregulated collective of posts, constantly evolves by the contributions of bloggers, who blog on almost anything, from politics, to popular events, fashion, products, etc to name a few subjects. This very nature of the blogosphere is highly regarded as a lucrative source of competitive intelligence information for marketers, companies, opinion trackers. Naturally, data mining the contents of the blogosphere is of great research interest and the findings of potentially actionable value. Blog search engines such as BlogScope offer a platform for analyzing the contents of the blogosphere [4, 7]. BlogScope warehouses the contents of the blogosphere; indexing, storing, processing and making it available for further analysis, visualization, and querying. Language processing and data mining techniques are employed to provide features such as: correlation of multiple information sources, automatic identification of events in time, visual summarization of chatter, influence inference, faceted search, exploratory navigation, and demographic segmentation. At the time of writing, the system is tracking 23M blogs, warehousing over 275M blog posts. Around 35K new blog posts are added to the system every hour, and visitors from 20K unique IPs are served daily. A detailed description of the features of BlogScope is outside the scope of this paper. We focus on one particular aspect of blog analysis offered.

BlogScope produces ranked lists (ranked according to various measures, such as popularity, interestingness, or term frequency) of keywords from the entire contents of the blogosphere. Stop words are removed automatically. The terms remaining form lists of size in the order of multiple millions of terms and are produced at select temporal intervals (per minute, hour, etc). As a result an agglomerative stream of ranked lists is produced, adding a new list every time step (say hour), evolving as a function of time. Such lists may be queried specifying temporal restrictions, to identify say the  $k$  most popular blog post terms for the user specified temporal intervals and values of  $k$ . The interval specified may vary arbitrarily, from minutes, to hours, days or months. Identification of

the  $k$  most popular terms can be easily conducted by utilizing well known *rank aggregation* algorithms, such as Fagin’s [14] threshold algorithm (TA) [18, 20, 14] or TA-sorted (Non Random Access, NRA) and its variants [14]. Such algorithms utilize monotone (commonly linear) combining (aggregation) functions to compose element ranks across all relevant lists. Although highly popular terms within a temporal interval reveal interesting information, most often such information is already known or anticipated. For example recent highly popular terms in the blogosphere have been, *pakistan, musharraf, afganistan, war, iraq, opensocial*, and *gphone* to name a few (for the week of November 5 2007).

In this paper we take a different approach to information discovery from such ranked lists that aims to unearth blog-chatter on not so expected or anticipated topics. We maintain the same rank aggregation framework but we *elevate* terms at a higher level by making use of popular *term hierarchies* commonly available. For example, popular car sites, contain hierarchies organizing cars by make, model, type, etc. All online shopping sites have hierarchies on the products they offer, such as kind, type, brand, model, features etc. Various web directories (e.g. Dmoz) provide hierarchical classification of websites. Even an individual analyst can supply its own hierarchy. Utilizing such hierarchies would aid discovery of different types of popular events.

For example, there might be lots of discussion about several digital camera models, such as the Canon S series of cameras. The terms S700, S600, etc, which are Canon digital camera models correspond to chatter about Canon digital cameras. However individually such terms might not have enough popularity to make it to the top ranking set of keywords, for rank aggregation queries. Utilizing product hierarchies, enables us to *elevate* such terms to the term *Canon digital camera*. Now, the total score of that newly introduced term in the presence of the product hierarchy would be higher (equal to the aggregate of the scores of all individual terms mapped to Canon digital camera). We wish to support highly dynamic hierarchies so we chose to impose no restriction on them, their shape or type and we assume that they are supplied on demand (per query). Mapping of terms in the ranked list to terms (nodes) in the hierarchy takes place at query time. This is a basic requirement in order to obtain a general solution. We would like to support rank aggregation queries in the presence of such hierarchies. Under the transformation imposed on terms of a ranked list by the use of a hierarchy, rank aggregation models in the literature [14] no longer apply. A fundamental obstacle in applying such algorithms, is that after the transformation the final score of a term is not known. One would have to, in the worst case, scan the entire list in order to correctly determine the score of a term that might be part of the final answer (highest scoring terms). As a result, such transformations in the presence of hierarchies, render early stopping in rank aggregation algorithms impossible.

An example is presented in Figure 1, where we have two ranked lists over car companies, camera brands and movie names. A hierarchy is also provided to elevate the keywords to higher level terms. This hierarchy is used to generate two transformed rankings. In both the rankings, Cars is at the top, but after aggregation, Camera has the highest total score. Notice that although popular rank aggregation algorithms [14] provide a certificate to test as an early stopping condition, no such certificate exists in this example. One has to always traverse the lists until the end in order to compute the correct answer.

In order to enable meaningful information discovery, the entire lists have to be scanned and transformed using information from

hierarchies<sup>1</sup>. Any thresholding based approach to limit the size of the ranked lists is deemed to fail, as it is not clear how to choose a meaningful value for this threshold. This seriously impacts performance as the ranked lists we operate on are in the order of millions of terms.

Motivated by the problem of information discovery in the blogosphere in this paper, we place the problem of rank aggregation in the presence of hierarchies into perspective and we propose algorithms to efficiently solve it. In particular we make the following contributions:

- We formally define the problem of rank aggregation in the presence of hierarchies ( $H - RA$ ) as an important information discovery primitive in ranked lists of keywords such as those produced by blog search and analysis engines.
- We present a framework to reason about early stopping during the computation of top- $k$   $H - RA$  in a probabilistic sense.
- Identifying the difficulties in solving the  $H - RA$  problem probabilistically, we present algorithms for solving the  $H - RA$  problem in a deterministic way with *high* (user specified) *precision* ( $pH - RA$  problem). This provides an alternative way to facilitate early stopping in the  $H - RA$  computations and subsequently obtain improved performance with controlled loss in accuracy.
- Since ranked lists are continuously produced, we present a method to organize them temporally in a streaming fashion enabling precomputation in order to obtain even further performance benefits in solving our problems, exploring space, performance tradeoffs in a controlled way.

Section 2 briefly reviews work related to the work presented here. Section 3 formally defines the problems of interest in our study. Section 4 formally presents our framework. Section 5 discusses probabilistic early stopping conditions for our problems, while Section 6 presents a framework and an algorithm for deterministic early stopping. Section 7 introduces preprocessing technology to be used in conjunction with our algorithms. Section 8 presents an experimental evaluation of our algorithms, we conclude in Section 9.

## 2. RELATED WORK

Top- $k$  problems are widely studied in the literature. The best known general purpose algorithm for this problem was proposed by Fagin et al. [14], Guntzer et al. [18], and Nepal et al. [20], and is commonly known as the *threshold algorithm* or TA. Many variants and enhancements of TA have been proposed [19, 10]. Theobald et al. [22] proposed a probabilistic variant of TA for approximate query processing. Arai et al. [1] introduced techniques to determine probabilistic confidence bounds on the correctness of results in the top- $k$  buffer during the execution of the TA algorithm. These are the only algorithm in the literature addressing probabilistic stopping conditions for the TA based top- $k$  algorithm [14]. Bast et al. [5] proposed different schemes to schedule list probes in order to improve the efficiency of TA. All the work mentioned above, however, focuses on the problem when the base lists do not contain any duplicates.

Little work has gone in developing algorithms for preprocessing the lists, and preparing them for TA execution. The only work

<sup>1</sup>If the mapping for some element is not present in the hierarchy, we copy it as is in the transformed list.

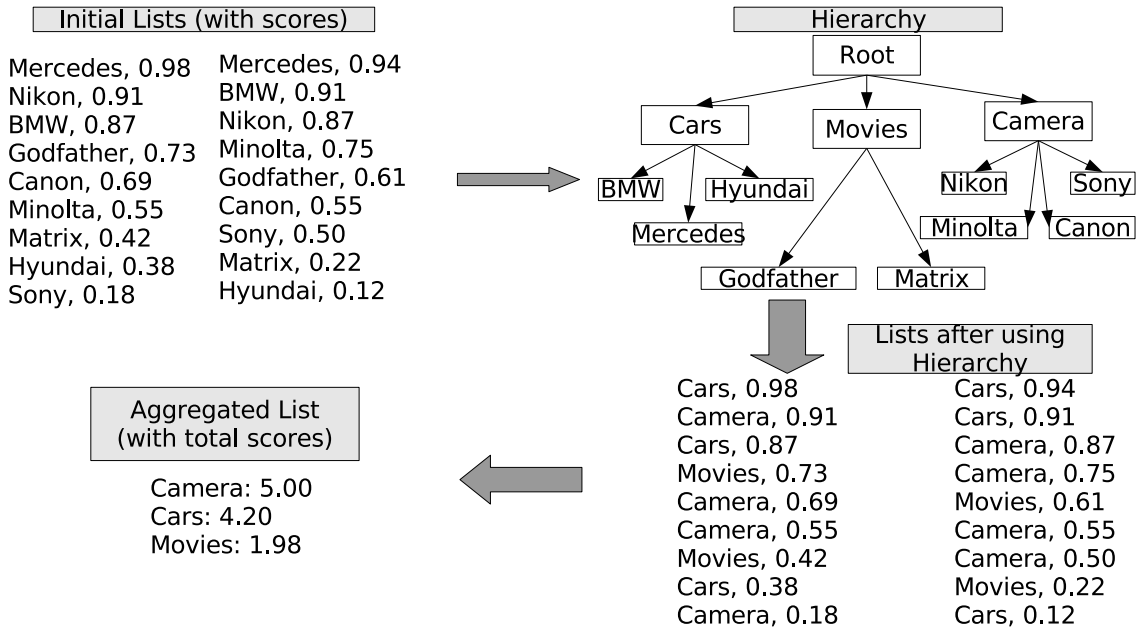


Figure 1: Aggregating two lists in presence of a hierarchy.

in this area that we know of is by Dubinko et al. [11], who maintain a binary tree on the lists to reduce computation performed at query time. This model however can not be tuned exploring trade-offs between offline precomputation and online query processing. Fontoura et al. [16] propose a preprocessing scheme for combining postings lists in an inverted index for efficient query answering. Their approach however is designed for sorted lists consisting only of numeric elements. Guha et al. [17] in the context of OLAP proposed a *sparse interval set* in order to maintain a better histogram approximation of a data distribution in the case of range query workloads. We build upon the sparse interval set in Section 7 and propose a preprocessing scheme that is incrementally maintainable in a streaming fashion. To the best of our knowledge the work presented herein is the first to consider approximate top- $k$  computations in the presence of lists with item multiplicity larger than one.

Researchers have addressed the problem of list merging in the past [12, 9, 23, 2]; most of this work is conducted in the context of meta search. In meta search, when a user submits a query, the meta search engine retrieves ranked lists of results from a few different search engines, and returns a merged list. Chakrabarti et al. [8] consider the problem of ranking objects in a search query, which requires computation of top- $k$  in presence of multiple scores in each list. Our problem and solutions however are highly different than those in [8] as we focus on approximate processing. We demonstrate that by relaxing the precision requirements, we can obtain large performance gains. Additionally, in all the above mentioned cases, the lists to merge are not available until just before merging, and hence precomputation of any intermediate results is not possible. In our case however, one can leverage the fact that our base lists are static and have a natural temporal order. Moreover, queries have temporal restrictions as well and thus maintaining auxiliary data structures is possible. We will capitalize on this property of the problem and design precomputation strategies to improve query performance. However, the hierarchies used to elevate keyword

to terms corresponding to hierarchy nodes are dynamic. They are provided at query time<sup>2</sup>. We do not wish to constrain our solution to specific hierarchies; any hierarchy can be provided along with a query and our algorithms should be able to report top ranking terms given the hierarchies provided on demand.

### 3. PROBLEM DEFINITION

Let  $X = X_1, X_2, X_3 \dots$  be a temporally ordered sequence of ranked lists. Each list  $X_i$  consists of ranked terms, according to some specified ranking function  $R$  (e.g., tfidf [21]). Let  $x_{ij}$  denote the  $j$ -th element for list  $X_i$ ,  $1 \leq j \leq N$ , where  $N$  is the size of the lists. Without loss of generality and to simplify our notation we assume that all lists are of the same size. At a specified temporal granularity (e.g., every hour) a new list is included in our sequence. Thus,  $X$  is an agglomerative stream of ranked lists  $X_i$ . Let  $s_{ij}$  denote the *score* according to  $R$  of the  $j$ -th element of the  $i$ -th list.

We consider hierarchies consisting of multiple levels. Each level consists of terms with similar abstraction (for example a level of a product hierarchy may be ‘type of product’, another level may be ‘product brand’, etc). Without loss of generality we utilize hierarchies to map (or elevate) terms of a list  $X_i$  to the same level of the hierarchy. Thus, a hierarchy  $H$  facilitates a mapping between the set of elements of  $X_i$  and terms at a specified level in the hierar-

<sup>2</sup>No prior knowledge of the hierarchy is assumed, which is required for handling unseen terms (as usually happens in dynamic information sources such as blogs and news). The hierarchies are constructed separately by a third party and not by us (our system just utilizes them). Such hierarchies are ubiquitous. For example, for product related hierarchies, sites like [www.buy.com](http://www.buy.com) and [www.bestbuy.com](http://www.bestbuy.com) contain extremely detailed product and feature categories. Similar sites exist to provide hierarchies for virtually any domain of interest (for cars see [www.car.com](http://www.car.com)). Google Directory ([directory.google.com](http://directory.google.com)), Yahoo Directory ([dir.yahoo.com](http://dir.yahoo.com)) and Dmoz Open Directory ([dmoz.org](http://dmoz.org)), each maintains its own continuously evolving categorization of websites.

chy<sup>3</sup>. Assume that the domain of terms at the specified level in the hierarchy is  $L$ . We treat a hierarchy as a function  $H(x_{ij}) = y$ ,  $1 \leq j \leq N$ ,  $y \in L$ , where  $|L| \leq N$ ; from the domain of terms in list  $X_i$  to the domain of terms at a specific level of a hierarchy. The effect of mapping list  $X_i$  using function  $H()$  is to produce a list  $Y_i$  of size equal to the size of  $X_i$ . However, since  $|L| \leq N$  there exist multiple elements in  $Y_i$  corresponding to the same term. With each term  $y$  from the hierarchy we associate two numbers:  $M_y$ , the *maximum multiplicity* of term  $y$ , is the maximum number of elements from a list  $X_i$  that can be associated to (map to)  $y$ . Notice that  $M_y$  is a property of the hierarchy; given any hierarchy we can easily identify  $M_y$  just by counting the children of the node  $y$ . We also associate  $m_y^i$ , the *actual multiplicity* which is the number of times  $y$  appears in list  $Y_i$  (mapped version of  $X_i$ ). The maximum multiplicity in the hierarchy, across all elements, is denoted by  $M = \max_{y \in L} M_y$ .

Let  $\phi(g)$ ,  $1 \leq g \leq m_y^i$  be a function returning the position  $j$ ,  $1 \leq j \leq N$  in  $X_i$  of the element  $x_{ij}$  that is mapped for the  $g$ -th time to  $y$ . We define the score of term  $y$  in list  $Y_i$  as

$$s_i(y) = \sum_{g=1}^{m_y^i} s_{i\phi(g)}.$$

**PROBLEM 3.1 (( $H - RA$ )).** Let  $Q = [X_i \dots X_j]$  be a query specified as a temporal restriction on  $X$ , along with a value  $k$  and a mapping  $H$ . The problem of rank aggregation in the presence of a hierarchy is equivalent to the problem of rank aggregation on lists  $Y_i \dots Y_j$ . More specifically is the problem of identifying the  $k$  highest ranking terms  $y$  in lists  $Y_i \dots Y_j$ , obtained by applying  $H()$  on the elements of  $X_i, \dots, X_j$ .

It is evident that the  $H - RA$  problem has an obvious solution if one is willing to scan all lists  $X_i \dots X_j$  entirely. We would like to obtain solutions that are able to report the answer faster. Since the score of term  $y$  in a list  $Y_i$  is not known precisely until the last element in the list is encountered in the worst case, it is evident that early stopping for the  $H - RA$  problem is not always possible in a deterministic sense. In Section 5 we demonstrate the difficulties associated with obtaining a probabilistic solution to this problem. Consequently, in order to be able to solve this problem faster and still have deterministic guarantees for the answers we are obtaining, we relax the  $H - RA$  problem as follows:

**PROBLEM 3.2 ( $H - RA$  WITH PRECISION  $p$  ( $pH - RA$ )).** Let  $Q = [X_i \dots X_j]$  be a query specified as a temporal restriction on  $X$ , along with a value  $k$ , a precision threshold  $p$ ,  $0 \leq p \leq 1$  and a mapping  $H$ . The rank aggregation problem in the presence of hierarchies with precision  $p$  is the problem of identifying the  $k$  highest ranking terms  $y$  in lists  $Y_i \dots Y_j$ , obtained by applying  $H()$  on the elements of  $X_i, \dots, X_j$ , such that at least  $p \cdot k$  answers correctly belong to the answer of  $H - RA$ .

Notice that we impose deterministic requirements for the solution of the  $pH - RA$  problem. We are seeking solutions that are able to report the answers to  $pH - RA$  fast and we describe several performance enhancements to our basic solutions exploring pre-computation.

## 4. AGGREGATION IN PRESENCE OF HIERARCHIES

We will employ the Fagin’s NRA (No Random Access) version of the threshold algorithm [14] to aggregate  $T$  lists,  $X_1, \dots, X_T$

<sup>3</sup>Without loss of generality assume that all elements of a list  $X_i$  map to terms at a specific hierarchy level. If this is not the case the term(s) not mapped are transferred to the corresponding list  $Y_i$  unmapped.

using a hierarchy  $H$ . In the discussion that follows, for simplicity, we assume the aggregation function to be an unweighted summation, though the techniques are generic enough to accept any linear monotone function, in accordance to previous work.

We probe the lists in a round robin manner. After reading an element  $x$ , we transform it to the appropriate term (on a specified level) in the hierarchy, as  $H(x) = y$ . If  $y$  has never been encountered before, we add it to a buffer that maintains a list of all seen elements along with their *worstcase score*, and information about how many times they have been seen in each list. Worstcase score  $ws(y)$  of  $y$  is the sum of scores of all seen instances of  $y$ . Therefore, after scanning  $d$  elements from each of the  $T$  lists,

$$ws(y) = \sum_{i=1}^T \sum_{j=1}^d I_{\{H(x_{ij})=y\}} \cdot s_{ij}$$

where  $I_{\{H(x_{ij})=y\}}$  is one if  $H(x_{ij}) = y$ , and zero otherwise. Also with each element  $y$  in the buffer, we maintain  $T$  counters,  $c_1(y), \dots, c_T(y)$ , with  $c_i(y)$  representing the number of times  $y$  is seen in lists  $Y_1, \dots, Y_T$ .

$$c_i(y) = \sum_{j=1}^d I_{\{H(x_{ij})=y\}}$$

When we encounter an element that already exists in the buffer, we update its worstcase score and increment the counters appropriately. To consider the elements never seen before, we include a *virtual element* in the buffer with worstcase score zero, and all counts also equal to zero.

During the execution, the buffer has three types of elements: fully seen, partially seen and completely unseen. For the elements fully seen, their worstcase score is the same as their final score. For the rest of the elements, we associate a quantity, *unseen score*, which is the sum of scores of unseen instances of the element. Hence, for  $y$ , unseen score in  $X_i$  will be (after reading  $d$  elements),

$$us_i(y) = \sum_{j=d+1}^N I_{\{H(x_{ij})=y\}} \cdot s_{ij}.$$

Total unseen score of  $y$  therefore will be  $us(y) = \sum_{i=1}^T us_i(y)$ . The problem however is that we don’t know the values of  $us(y)$ . As a result, we can try to estimate it probabilistically or determine an upper bound for it. We explore both possibilities in the sections that follow.

Periodically we order the elements in the buffer according to their worstcase score, and select the  $k$  highest scoring elements as the current top- $k$  (denoted by *currTopK*). The minimum worstcase score among these elements is denoted by *min-k*. The termination condition of the algorithm is determined by comparing the *min-k* score with the scores of elements not in the current top- $k$ . In the next section, we discuss a probabilistic stopping condition utilizing this framework, while in Section 6 we propose a relaxed deterministic condition for termination.

## 5. AGGREGATION WITH PROBABILISTIC GUARANTEES

A probabilistic stopping condition will enable early termination of the algorithm by relaxing the accuracy requirement at termination. Such approximate query processing is desirable in information discovery application scenarios where fast turn-around time is important. Since the probabilistic algorithm requires knowledge of score distributions to estimate the unseen score, we can approximate the scores either by employing a parameterized distribution

(e.g., geometric) or by maintaining precomputed histograms on the scores for each list. We analyze both cases in detail.

## 5.1 Probabilistic TA Assuming Geometric Distribution

The main premise of early stopping in the TA algorithm is that scores decrease as we traverse the list. To capture such decreasing scores, we use the following geometric condition<sup>4</sup>: if  $s_{ij}$  denotes the score of  $j^{\text{th}}$  element in list  $X_i$ ,

$$s_{i,j+b} \leq r_i \cdot s_{ij} \quad \forall j, r_i \leq 1,$$

for some fixed constant  $b$  in every list  $X_i$  (notice that since all lists are available, such a  $r_i$  can always be determined). This means that the score must decrease by a factor of at least  $r_i$  every  $b$  elements. This implies that the total score of all elements in the list  $X_i$  is less than  $b(1 + r_i + r_i^2 + \dots + r_i^{n/b}) = \frac{b \cdot (1 - r_i^{n/b})}{1 - r_i}$ , assuming all scores in the range  $[0, 1]$ . We can precompute the value of  $r_i$  for each of the lists and store it as metadata.

For the analysis, we assume that each element  $y$  has a probability  $p_{yi}$  associated with it for every list  $X_i$ , and each position in the list  $X_i$  is filled by selecting elements (terms) independently. This means that elements for each position in the list are selected independently, and an element  $y$  has a chance  $p_{yi}$  of being selected at any position in  $X_i$ .

$$\Pr[H(x_{ij}) = y] = p_{yi}, \quad \forall j \leq N \text{ (independent of } j)$$

We now apply the TA algorithm as described in the previous section. The unseen score of  $y$  in a list is the total score of all instances of  $y$  that are yet to be seen. The unseen score can be computed by modeling it as a sum of independent random variables (one for each unseen position). Thus, after reading  $d$  elements from the list  $X_i$ , the expected unseen score  $\mu_{yi}$ , for  $y$  in  $X_i$  can be computed as

$$\begin{aligned} \mu_{yi} &= \sum_{j=d+1}^N p_{yi} s_{ij} \\ &\leq \sum_{j=d+1}^N p_{yi} s_{id} r^{(j-d-1)/b} \\ &\approx s_{id} p_{yi} b \cdot \frac{1 - r^{(N-d)/b}}{1 - r} \\ &\approx s_{id} \cdot \frac{p_{yi} b}{1 - r} \end{aligned}$$

The simplification in the last step assumes that the number of elements  $N$  in the list is much larger than  $b$  (hence  $r^{(N-d)/b}$  is negligible). Similarly the standard deviation of the unseen score  $us(y)$  is,

$$\sigma_{yi} = s_{i,d} \cdot \sqrt{\frac{p_{yi}(1 - p_{yi})b}{1 - r^2}}$$

Using this information, we can compute the mean and variance of  $us(y)$  as  $\sum_{i=1}^T \mu_{yi}$  and  $\sum_{i=1}^T \sigma_{yi}^2$ . This can be used in conjunction with the Chebyshev [15] inequality—for a random variable  $Z$ ,  $P(|Z - E[Z]| \geq \epsilon) \leq \frac{\sigma_Z^2}{\epsilon^2}$ —to estimate the probability that  $us(y) + ws(y) > \min-k$  (with some specified confidence bound). If this bound is not tight enough, we can also use Chernoff [15] bounds as presented below.

<sup>4</sup>For the purpose of analysis, we make certain simplifying assumptions. Practical applicability of the approach is discussed later in the section.

The fact that  $y$  appears or not at the  $j^{\text{th}}$  position in  $X_i$  is a Bernoulli trial with success probability  $p_{yi}$ , moment generating function of which is  $g_{ij}^y(t) = (1 - p_{yi}) + e^{t s_{ij}} p_{yi}$ . The moment generating function of the unseen score of  $y$  in the list  $X_i$  therefore will be,

$$\begin{aligned} g_{yi}(t) &= \prod_{j=d+1}^n g_{ij}^y \\ &\leq \prod_{j=d+1}^n \left( (1 - p_{yi}) + p_{yi} e^{t \cdot s_{id} r^{(j-d)/b}} \right) \end{aligned}$$

For large values of  $j$ , the term in the product becomes insignificant as it approaches one, and hence can be neglected. The moment generating function for  $us(y)$  will be  $g_y(t) = \prod_{i=1}^T g_{yi}$ . This information can be used with the Chernoff bound for estimating the probability that  $us(y)$  is greater than  $\min-k$  (with a specified confidence value). Specifically for a random variable  $Z$  with moment generating function  $g_Z(t)$ ,

$$\begin{aligned} \Pr(Z \geq z) &\leq g_Z(t) e^{-tz} \text{ for } t \geq 0, \\ \text{and } \Pr(Z \leq z) &\leq g_Z(t) e^{-tz} \text{ for } t \leq 0. \end{aligned}$$

To complete the TA algorithm described in the previous section, we use the following termination condition. For each element in the buffer (including the virtual element) but not in the current top- $k$ , we evaluate the probability that it does not belong to the top- $k$  (using the Chernoff bound with  $g_y(t)$ , or Chebyshev bound with mean and variance). This probability is,

$$\Pr[y \text{ belongs to top-}k] = \Pr[us(y) > \min-k - ws(y)] \quad (1)$$

If the maximum of such probabilities is bounded by some parameter  $\epsilon$ , we terminate. The probability that the precision of this algorithm is  $k'/k$ , i.e.  $k'$  out of  $k$  elements in the output belong to the actual top- $k$  answer for the problem, is  $\binom{k}{k'} (1 - \epsilon)^{k'} \epsilon^{k-k'}$ .

Estimation of  $p_{yi}$  is a crucial step. We can either assume that  $y$  has the same multiplicity  $M_y$  in each list; in this case  $p_{yi} = M_y/N$ . If we don't assume this, we can estimate  $p_{yi}$  by applying Bayesian Decision Theory [6] since the number of occurrences of  $y$  in  $X_i$  follows a binomial distribution. After traversing  $d$  elements of a list, we have gained some knowledge about this binomial distribution, and hence can guess its characteristic probability. If we assume the prior to be a Beta distribution,  $B(\alpha, \beta)$ , the posterior will also be Beta distributed,  $B(c_i(y) + \alpha, d - c_i(y) + \beta)$ , where  $d$  is the total number of observations and  $c_i(y)$  is the number of successes. With no other information, we can assume the prior to be uniformly distributed (uniform distribution is the same as the Beta distribution with parameters 1 and 1). We can also maintain precomputed information about the distribution of multiplicities for each list (if we have some knowledge about hierarchies) to use as a prior for a more accurate estimation. Therefore, after traversing  $d$  elements in a list, and encountering  $c_i(y)$  occurrences of  $y$ ,

$$p_{yi} \sim B(c_i(y) + 1, d - c_i(y) + 1).$$

For the first case, when we assume we know  $p_{yi}$  as  $M_y/N$ , we can use the Chernoff bound with  $g_y(t)$  to estimate the probability in Equation 1. This will require a minimization on parameter  $t$ , to find a tight bound on  $us(y)$ . If we wish to use the Beta distribution for estimating  $p_{yi}$ , we can first find a bound on  $p_{yi}$  and then use that to compute  $g_y(t)$  in order to apply Chernoff bounds.

While the probabilistic analysis discussed earlier is numerically possible (using tools like Matlab), it involves the solution of complex equations at each estimation interval, which will impose significant computational overheads. The assumption that scores fol-

low a geometric distribution and elements are selected independently is a simplifying assumption required for analysis purposes. Similar analysis can be done assuming different probabilistic distributions. Without these assumptions, the probabilistic approach will become further complicated. Such an approach is interesting from a modeling perspective, however to obtain a practical solution we seek lightweight algorithms. We discuss a solution that utilizes information precomputed on base lists (the  $X_i$ 's) in the form of histograms in the next section.

## 5.2 Probabilistic TA Using Histograms

Probabilistic modeling of the adaptation of the TA algorithm in our setting in order to derive early stopping conditions is an interesting exercise, as discussed in the previous section. A practical realization of such an approach however is not easy, due to the high computational overheads for solving the required equations at realistic scales of the problem. The only work that discusses implementation of a probabilistic top- $k$  approach is by Theobald et al., [22]. They provide an analysis of their framework with parameterized distributions (like Uniform and Poisson) for modeling purposes, but since real life data usually do not follow such distributions, they propose the use of histograms for approximating score distributions. Probabilistic bounds can be derived by convoluting these histograms. We discuss an adaptation of such an approach to our problem.

We maintain precomputed histograms over scores for each of the lists. To simplify our presentation assume that  $y$  has multiplicity  $M_y$  in each list. Assume that we have seen an element  $y$ ,  $c_i(y)$  times, in the list  $X_i$  after  $d$  iterations. Let  $P_i^d$  denote the score distribution of the unseen score part of  $X_i$  ( $P_i$  will change as we progress in the algorithm). As a result the probability distribution of  $us_i(y)$  will be the convolution of  $P_i^d$  done  $M_y - c_i(y)$  times with itself, i.e.,

$$\Pr_{us_i(y)} = CONV_{j=1}^{M_y - c_i(y)} P_i^d \quad (2)$$

The probability distribution of  $us(y)$ , which is a sum of  $us_i(y)$ , will therefore be,

$$\Pr_{us(y)} = CONV_{i=1}^T \left( CONV_{j=1}^{M_y - c_i(y)} P_i^d \right) \quad (3)$$

$CONV(P_1, P_2, \dots)$  here denotes the convolution of  $P_1, P_2, \dots$  distributions. Using this probability distribution we can evaluate the probability that an element not in the current top- $k$  can enter the actual top- $k$ . Then we terminate the algorithm when this probability is bounded by  $\epsilon$  for all elements in the buffer (including the virtual element). Formally, we stop when,

$$\max_{y \notin currTopK} \Pr[ws(y) + us(y) > \text{min-}k] < \epsilon$$

The cost of each convolution required is exponential in the number of lists being aggregated. This is further complicated by the fact that this needs to be done for each element in the buffer, and at each estimation step (since  $P_i^d$  changes as we progress down the lists for each element). Even if we group together the elements with identical counts (number of times seen in each list), there will be  $T^M$  different convolutions, where  $M$  is the maximum multiplicity across all the elements in the hierarchy. It is evident that this technique, being exponential in both  $M$  and  $T$ , is not practically viable. For any realistic setting,  $T$  can be expected to be at least in the order of ten or hundred lists and  $M$  equally sizable.

The solution outlined above extends the technique of Theobald et al. [22] in our setting in which we obtain lists as they are transformed by hierarchies on demand. These hierarchies introduce duplicates in the lists, forcing the probabilistic estimation framework

to consider each element multiple times for each list. This leads to an increase in the number of histograms convoluted for each step by a factor of  $M$ , increasing the time required to convolute by an exponential factor. The number of different possible distributions that the unseen score of an element can follow also increases exponentially from  $2^T$  to  $2^{TM}$ , since each of the  $T$  counters can now have value between 1 and  $M$ . The work by Theobald et al. is focused towards search queries in information retrieval where the number of lists under aggregation is small (2-4), while our application, in order to support ad-hoc restrictions on temporal range, requires techniques that are scalable to hundreds of lists.

To confirm our analytical expectation that such approaches are not viable for our setting, we implemented the probabilistic framework discussed above using histograms. Experiments with this approach show that the computational effort required for convolutions is significantly (orders of magnitude) higher than the savings in I/O. Even for small data sizes (10 lists, with multiplicities in range 5 – 10), such a probabilistic approach requires significantly more time than the naive approach of scanning all lists entirely, aggregating term scores in memory and sorting to identify the top- $k$  elements.

## 6. DETERMINISTIC AGGREGATION WITH PRECISION GUARANTEES

The maximum possible score of every element in the buffer can be computed using the knowledge of the maximum multiplicity of an element in a list as,

$$\text{maxPossibleScore}(y) = ws(y) + \sum_{i=1}^T s_{id}(m_y^i - c_i(y)).$$

Since  $m_y^i$  may not always be known or available, we can instead use  $M_y$  which can be determined by the hierarchy at query time to bound the maximum possible score:

$$\text{maxPossibleScore}(y) \leq ws(y) + \sum_{i=1}^T s_{id}(M_y - c_i(y)). \quad (4)$$

One possible termination condition for the algorithm of Section 4 is to stop when no element (except those in the current top- $k$ ) in the buffer (including the virtual element) has its maximum possible score above the min- $k$  score. This stopping condition provides the solution to the  $H - RA$  problem, and guarantees that the precision of the top- $k$  elements we output will be 1.0. But such a stopping condition is overly conservative and pessimistic; it essentially assumes that all unseen instances of elements not in the current top- $k$  are right after the current scan position. Contrasting this with the original TA algorithm (operating on lists without duplicates), this problem is more serious here since in our case, the estimate of  $\text{maxPossibleScore}(y)$  is amplified by a factor of  $M_y$  (notice that  $M_y=1$  in the case of lists without duplicates). Such an overestimate of  $\text{maxPossibleScore}$  eliminates the possibility of early stopping. We report experiments in Section 8 confirming this expectation.

From an information discovery point of view, obtaining results fast is a primary concern. Normally users do not know what exactly we are looking for, thus they submit multiple queries utilizing diverse hierarchies and seek interesting trends. Fast turnaround time in such a speculative information discovery process is essential. To increase the possibility of early stopping of the algorithm (and thus obtain improved performance) while maintaining a deterministic framework for query answers, we relax our precision requirements supplying a target precision value  $\rho$  for the desired result at query

---

**Algorithm 1** TA: Solving  $pH - RA$ 

---

**INPUT**  $k, \rho, H$  and input lists,  $X_1, \dots, X_T$ ,  
1: Initialize buffer to empty  
2: Add one virtual element to the buffer with its worstcase score and all  $c_1, \dots, c_T$  counters set to zero  
3: set  $falseNegatives = \text{infinite}$   
4: **while**  $falseNegatives > (1 - \rho)k$  **do**  
5:   **for**  $i = 1$  to  $T$  **do**  
6:     read  $x = \text{next element from } X_i$   
7:     read  $s_i = \text{score of } x \text{ from } X_i$   
8:     set  $y = H(x)$   
9:     **if**  $y$  not in buffer **then**  
10:       insert  $y$  in the buffer with its worstcase score and all  $c_1, \dots, c_T$  counters set to zero  
11:     **end if**  
12:     Increase worstcase score of  $y$  in buffer by  $s_i$   
13:     Increment the  $i^{\text{th}}$  counter  $c_i$  for  $y$  in buffer by 1  
14:   **end for**  
15:   **if**  $\text{size}(\text{buffer}) \geq k$  **then**  
16:     sort the buffer in descending order of worstcase scores  
17:     set  $\text{min-}k = \text{score of } k^{\text{th}} \text{ element in buffer}$   
18:     set  $falseNegatives = 0$   
19:     **for**  $j = k + 1$  to  $\text{size}(\text{buffer})$  **do**  
20:       set  $x' = j^{\text{th}}$  element of buffer  
21:       set  $\text{max}(x') = \text{ws}(x') + \sum_{i=1}^T s_i(M_{x'} - c_i(x'))$   
          { $\text{ws}(x')$  is the worstcase score of  $x'$ ,  $c_i(x')$  is the  $i^{\text{th}}$  counter for  $x'$  in the buffer, and  $M_{x'}$  is the multiplicity of  $x'$  in  $H$ }  
22:       **if**  $\text{max}(x') > \text{min-}k$  **then**  
23:         **if**  $x'$  is the virtual tuple **then**  
24:         set  $falseNegatives = \text{infinity}$   
25:         **end if**  
26:         set  $falseNegatives = falseNegatives + 1$  {If max score of virtual tuple is greater than min- $k$  score, number of false negatives can be infinite}  
27:       **end if**  
28:     **end for**  
29:   **end if**  
30: **end while**  
31: **OUTPUT** top- $k$  elements in the buffer

---

time. Our algorithm guarantees that the top- $k$  result set we output will have precision greater than  $\rho$ . We refer to this problem as  $pH - RA$ . Setting  $\rho$  to 1.0,  $pH - RA$  reduces to  $H - RA$ .

Thus we seek to identify a number of elements  $k'$  that have their maximum possible score above the min- $k$  score. If the maximum possible score of the virtual element is less than the min- $k$  score, and  $k' < k$ , then at least  $k - k'$  among the current top- $k$  elements belong to the actual top- $k$ . This means that the precision is guaranteed to be at least  $1 - k'/k$ . Hence, we can terminate when  $1 - k'/k \geq \rho$ .

As we decrease the value of target precision  $\rho$ , our algorithm will be able to terminate earlier. Experimental results demonstrate that the observed precision is usually significantly higher than  $\rho$ . Pseudo code for the complete algorithm  $pH - RA$  is presented as Algorithm 1.

Algorithm 1 can also be used as a filter for two step top- $k$  computation. We can set  $\rho$  to a low value (e.g., 0.2) in the first step and find top  $k/\rho$  elements using the  $pH - RA$  algorithm. The output of the first step is guaranteed to contain all correct top- $k$  elements. In the second step, random access can be used to disambiguate the results to identify the actual top- $k$  result set.

As an example, we present in Table 1 a snapshot of the buffer after scanning 7 elements from each of the two initial lists displayed in Figure 1 using the same hierarchy. In this example,  $\text{max}$  scores

Element	$ws$	$c_1$	$c_2$	$\text{max}$
Camera	4.82	3	4	5.24
Cars	3.70	2	2	4.62
Movies	1.76	2	1	2.26
Virtual	0.0	0	0	1.84

**Table 1: Example snapshot of the buffer after completion of 7 iterations during the algorithms' execution on the lists shown in Figure 1. This table presents the element, its worst score, two counters and the maximum possible score.**

are calculated using Equation 4 using  $s_{1,7} = 0.42$  and  $s_{2,7} = 0.50$  since  $d = 7$ , and taking in account that  $M_{Cars} = 3$ ,  $M_{Camera} = 4$  and  $M_{Movies} = 2$ . If we were computing top-2, the min- $k$  score would be 3.70, which is greater than the best scores of all other elements (including the virtual tuple), and hence  $(Cars, Camera)$  would be the true answer for all values of target precision  $\rho$ .

## 7. PREPROCESSING FOR AGGREGATION

Since all the lists  $X_i$  are available to us in advance, we explore and design a framework for efficient answering of ad hoc rank aggregation queries over subsets of these lists by utilizing a suitable preprocessing strategy. In the absence of user supplied 'preferences' towards individual lists, we assume that each list participates in rank aggregation with the same weight. We remark that the preprocessing strategies presented in this section can be extended to incorporate such information if available. Our goal is to obtain even better performance via precomputation. Motivated by the streaming nature of our problem (a new list  $X_i$  arrives at every timestep) we design solutions that are amenable to incremental maintenance. Our goal is to support ad hoc range queries on the evolving stream of lists  $X_i$ . A range query requesting merging all lists between time  $[t, t + s]$  requires merging of all lists in the range  $[X_t, X_{t+s}]$ . Since preprocessing will increase the space requirements of our solution, we seek solutions that offer a tunable trade-off between increased storage requirements and query answering time.

We construct a *sparse interval set* on our input lists (that is incrementally maintainable in a streaming setting). Maintenance of such a structure guarantees that every range query of size  $s$  can be answered by merging less than  $2\lceil \log_2 s \rceil + 2$  precomputed lists, for some tunable parameter  $l$ . Use of sparse set system does not affect the accuracy of query result.

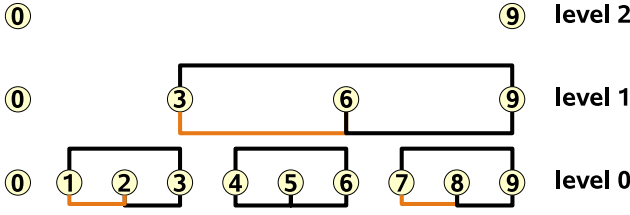
Consider  $l > 1$  a parameter and set  $r = \log_2 n$ , where  $n$  is the number of lists currently in the system. Assume that the items are indexed  $0, 1, \dots, n - 1$ ; they define the level 0 points.

- Consider the numbers  $0, l^j, 2l^j, \dots$ . These define points at level  $j$ .
- Overall we have  $r + 1$  levels with 0 and  $n$  as level  $r + 1$  points.
- The interval  $[0, n - 1]$  is in the sparse system  $S$ . Any pair of level  $j$  points between adjacent level  $j + 1$  points defines an interval in  $S$ .

CLAIM 7.1. *A point is never contained in more than  $O(l^2 \log_2 n)$  intervals.*

PROOF. Follows from [17].  $\square$

The sparse system can be maintained incrementally, since we only need more intervals as lists arrive, never having to delete or remove anything. Each time a new list arrives we increment  $n$  to  $n + 1$ ,



**Figure 2: An example sparse system over 10 initial points with  $l = 3$ . Lines connecting points represent intervals in the system. Three orange intervals show intervals that need to be merged for answering the query  $[1, 8]$ .**

List Arrival	Intervals Added
0	none
1	none
2	$[1, 2]$
3	$[2, 3], [1, 3]$
4	none
5	$[4, 5]$
6	$[5, 6], [4, 6], [3, 6]$
7	none
8	$[7, 8]$
9	$[8, 9], [7, 9], [6, 9], [3, 9]$

**Table 2: Intervals that are added to the sparse set system of Figure 2 as lists arrive in a streaming setting.**

and identify all the intervals that end at the index corresponding to the newly arrived list. Once the intervals are identified, we retrieve and merge base lists corresponding to each of these intervals and save them for later use. To merge the base lists, we read each one in memory (one by one), and compute the total score for each element  $x$  in them, and save back a merged list with scores sorted in descending order. Claim 7.1 ensures that the amount of preprocessing required is bounded.

**CLAIM 7.2.** Any range  $[0, s]$  can be represented as a disjoint union of at most  $\lceil \log_l s \rceil + 1$  intervals from the above collection.

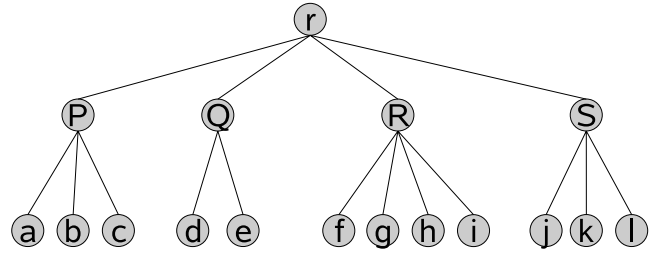
**PROOF.** We will first show using induction that,  $[1, s]$  can be represented by using at most  $j$  intervals where  $s \leq l^j$ . This is true for the base case  $j = 1$ . Now consider  $l^j < s \leq l^{j+1}$ . We can write  $[1, s]$  as  $[1, \alpha l^j]$  and  $[\alpha l^j + 1, s]$  where  $\alpha$  is maximal. The later is similar to  $[1, s - \alpha l^j]$ . Since  $\alpha$  is maximal,  $s - \alpha l^j \leq l^j$ , and therefore  $[1, s - \alpha l^j]$  can be represented by  $j$  intervals. Also  $[1, \alpha l^j]$  is a valid interval in the sparse system. Thus by induction,  $[1, s]$  can be represented by using at most  $j$  disjoint intervals.

Since  $[1, s]$  can be represented by using at most  $\lceil \log_l s \rceil$  intervals,  $[0, s]$  can be represented by  $\lceil \log_l s \rceil + 1$  intervals.  $\square$

**CLAIM 7.3.** Any range  $[t_1, t_2]$  can be represented as a disjoint union of at most  $2\lceil \log_l s \rceil + 2$  intervals from the above collection, where  $s = t_2 - t_1 + 1$ .

**PROOF.** The query  $[t_1, t_2]$  can be divided in two parts  $[t_1, \alpha l^j]$  and  $[\alpha l^j + 1, t_2]$ , but cutting it at a point  $\alpha l^j$ , such that  $j$  is maximum. By symmetry, and using the result of previous claim, we can express each of these parts by at most  $\lceil \log_l s \rceil + 1$  intervals, totaling to  $2\lceil \log_l s \rceil + 2$ .  $\square$

We precompute and maintain lists for each of the intervals in the sparse system. At query time, we first find a minimal set of disjoint



**Figure 3: An example hierarchy.**

$X_0$	$X_1$	$X_2$	$X_3$	$X_4$	Scores
a	k	c	d	a	0.9
f	l	d	a	l	0.8
l	f	e	j	k	0.7
k	b	j	c	c	0.6
c	d	f	l	j	0.5
b	j	k	k	b	0.4
j	c	l	f	f	0.3

**Table 3: Five example input lists  $X_0$  to  $X_4$  with scores. Scores for all the lists are assumed identical.**

intervals that cover the query range exactly. This can be done by recursively identifying the largest interval contained in the query. Claim 7.3 ensures that the number of such intervals will be bounded to be logarithmic in the actual query size. We retrieve the precomputed lists for each of these intervals and apply the  $pH - RA$  algorithm. Note that, by constructing the sparse set, we are not making any approximation, and hence all our correctness guarantees remain valid.

In order to find the minimal set of disjoint intervals, we first identify the largest interval contained in the query and then recurse for the remaining parts. Our experiments show that this step usually takes a few milliseconds for query range size in the order of hundreds, and hence optimizations here are of little significance.

Figure 2 shows an example of the sparse system after 9 lists have been added. Table 2 shows the intervals which are added to the system after each list arrives. We maintain precomputed aggregated base lists ( $X_i$ ) for each of these intervals. When the query  $[1, 8]$  arrives, we first determine a minimal disjoint set of intervals required to exactly cover the query. This can be done by recursively identifying the largest interval contained in the query, which results in  $[1, 2]$ ,  $[3, 6]$ , and  $[7, 8]$ . We fetched the lists corresponding to these three intervals and apply  $pH - RA$ . Utilizing the sparse set system in this case, we had to merge only 3 lists as opposed to 9, hence reducing the query time by a factor of 3.

## 7.1 Offline Preprocessing of Hierarchies

If the hierarchies are available to us for offline computation, we can preprocess them and maintain for each element  $y$  its multiplicity  $m_y^i$  in each list. To save on storage, we may further compress this information by grouping elements with similar multiplicities in the same bucket, and by replacing their multiplicity by its maximum possible value. Such relaxation will not affect the accuracy of the proposed algorithms. More relaxed estimates on multiplicities however may lead to longer running times.

For example consider the hierarchy in Figure 3. If this hierarchy is available for preprocessing along with the five input lists in Table 3, then multiplicities as shown in Table 4 can be maintained precomputed. Observe that while the maximum multiplicity for the group  $R$  can be 4, it is actually much less (1 in all the five lists);

List	$P$	$Q$	$R$	$S$
$X_0$	3	0	1	3
$X_1$	2	1	1	3
$X_2$	1	2	1	3
$X_3$	2	1	1	3
$X_4$	3	0	1	3

**Table 4: Multiplicities in  $X_0$  to  $X_4$  based on the hierarchy of Figure 3**

hence such precomputed structures can aid to better estimate scores at runtime.

If storage is not a constraint, we can also preprocess each list, by grouping all its elements that map to the same higher level term in the hierarchy. This way, the resulting list will not have any duplicates. For each interval of the sparse system, we can maintain such a list. When a query arrives, we can retrieve the already merged lists without any duplicates and aggregate them using the vanilla TA algorithm [14]. Table 5 presents preprocessed lists without duplicates for lists shown in Table 3 using the hierarchy in Figure 3. Note that the storage of completely preprocessed lists requires more space than storing just the multiplicities.

$X_0$	$X_1$	$X_2$	$X_3$	$X_4$
$P$ (1.8)	$S$ (2.1)	$Q$ (1.5)	$S$ (1.6)	$S$ (2.0)
$S$ (1.6)	$P$ (0.9)	$S$ (1.3)	$P$ (1.4)	$P$ (1.9)
$R$ (0.8)	$R$ (0.7)	$P$ (0.9)	$Q$ (0.9)	$R$ (0.3)
$Q$ (0.0)	$Q$ (0.5)	$R$ (0.5)	$R$ (0.3)	$Q$ (0.0)

**Table 5: Complete preprocessing of the five lists using the hierarchy. Aggregated scores are shown in brackets.**

## 7.2 Online Processing of Hierarchies

If we don't have the hierarchy available for preprocessing, we can do a single pass on the hierarchy at query time to find the maximum multiplicity  $M_y$  of each element  $y$ . This information can be used with the TA algorithm presented in Section 6. For the hierarchy in Figure 3, one scan over the hierarchy is sufficient to deduce multiplicity information as presented in Table 6.

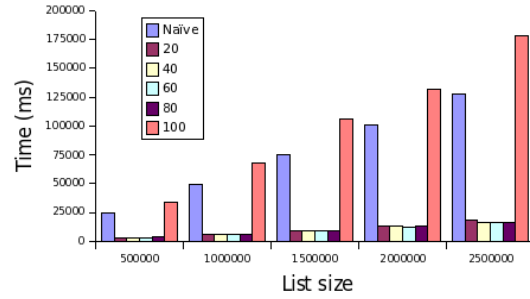
$P$	$Q$	$R$	$S$
3	2	4	3

**Table 6: Multiplicities in the hierarchy of Figure 3**

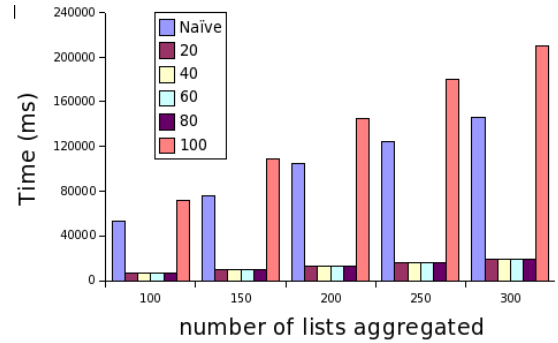
For constructing the sparse set system, we can merge the lists (simple aggregation without hierarchy) for each of the intervals in the sparse system, and save them for later use. When the query arrives, we can retrieve these already merged lists (maximum  $2^{\lceil \log_i s \rceil}$  for a query) and aggregate them using  $pH - RA$  as described in Section 6.

## 8. EXPERIMENTS

We conduct a variety of experiments to evaluate the techniques presented in the previous sections. We compare performance of the algorithms in terms of their running times. We report precision values obtained and Spearman's footrule distance between the observed and the actual top- $k$  result to demonstrate the qualitative effectiveness of the aggregation algorithms proposed. The experiments were conducted on a 3.0GHz Pentium 4 machine with 3GB RAM, running SUSE Linux. All the algorithms were implemented



**Figure 4: Observed running times of the naive algorithm and of  $pH - RA$  (Algorithm 1) for different target precision values (20 to 100) as we vary the list size.**



**Figure 5: Observed running times of the naive algorithm and of  $pH - RA$  (Algorithm 1) for different target precision values (20 to 100) as we vary the number of lists aggregated.**

in Java. For performance comparisons, operating system caches and buffers were cleared before each run (by remounting the underlying filesystem).

In our evaluation we utilize both synthetic and real data. In order to be able to flexibly explore several aspects of our algorithms and vary parameters on demand we utilize synthetic data sets. We demonstrate the practical utility of our techniques with real data sets from BlogScope in Section 8.3.

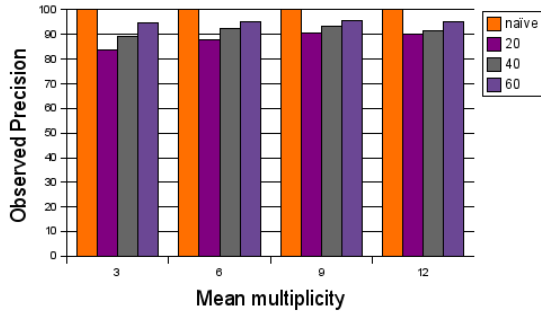
### 8.1 Evaluating $pH - RA$

In the experiments presented in this section, we deploy Algorithm 1 to merge lists in order to produce a top-100 result set. We evaluate the termination condition everytime 10% of the size of the list has been read. Running times and precision observed were recorded for each experiment. Precision in this case is the fraction of elements in the top- $k$  the algorithm outputs that also belong to the actual (precision  $p = 1$ ) top- $k$ .

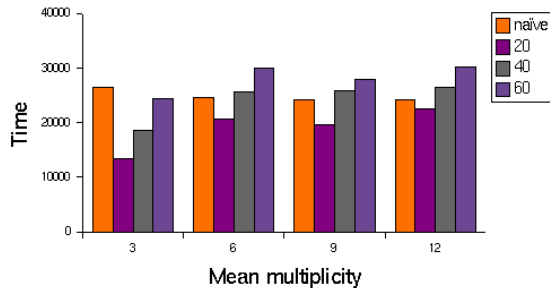
Observing real data sets extracted from BlogScope, we note that several terms in the base lists  $X_i$  appearing in close proximity in  $X_i$  (i.e., their relative popularity is close<sup>5</sup>) map to the same hierarchy node. For example, keywords *saddam*, *iraq* and *america* appear very close together in base lists. Same behavior is observed for other keywords such as *sony*, *nintendo*, *xbox*, *PS3*. Thus, to generate the lists, we started with an initial list, and conducted a number

<sup>5</sup>Real datasets consists of lists of tokens extracted from BlogScope for each day. Each of these lists is sorted according to the popularity of the token in the blogosphere for that day [4].

of random swaps of consecutive elements (equal to five times the length of the initial list) in order to obtain the next list. We generated 300 lists in this way. In order to generate a hierarchy we start with the initial list, make 50 times the list size number of random swaps (of consecutive elements) and group together consecutive elements to the same higher level term in the hierarchy. This way, a slight correlation between the position of terms and the hierarchy is introduced in the data, i.e., elements that map to the same higher level term are close in positions. Multiplicities are derived from a Normal distribution with  $mean = 6$ . The variance of the Normal distribution was set to  $mean/3$  and all multiplicities were restricted between 1 and  $2 \cdot mean$ . The scores were derived from an exponential distribution,  $\Pr[z < Z] = e^{-\lambda z}$ , with the parameter  $\lambda = 0.005$ .



**Figure 6: Observed precision values for the naive algorithm and for different target precision values (20, 40, and 60) in Algorithm 1 for ‘adverse’ data.**



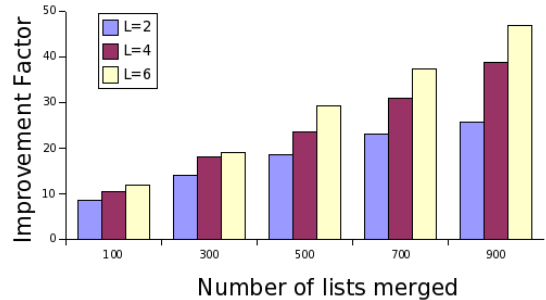
**Figure 7: Observed running times of the naive algorithm and of  $pH - RA$  (Algorithm 1) for different target precision values (20, 40, and 60) for ‘adverse’ data.**

Figure 4 presents a comparison of running times as we merge 100 lists and vary the list size from 0.5 million to 2.5 million. Figure 5 presents the same comparison, but this time we vary the number of lists from 100 to 300, keeping list size fixed at 0.5 million. The observed precision is 100% in all the cases and hence is not reported explicitly in the graph.

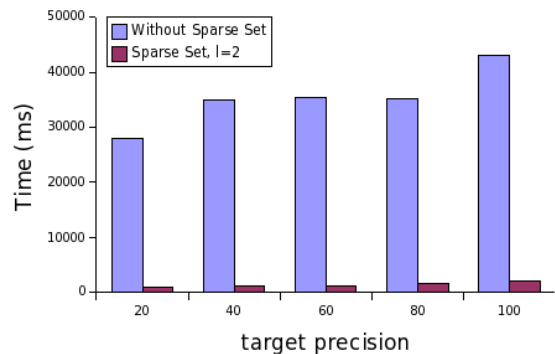
We refer to the algorithm that reads the base lists in their entirety and conducts rank aggregation in memory as *Naive*. It can be observed that as long as the precision parameter (value of  $p$ ) supplied at  $pH - RA$  is smaller than 100% the improvements in running time are profound (a factor of 10 improvement). The observed (actual) precision in all cases for Figures 4, 5 is always 100%. It is only when we set the precision parameter of  $pH - RA$  to 100% that the algorithm observes a performance degradation relative to Naive; as

explained in the Section 6, TA is highly conservative and its over-estimation of  $maxPossibleScore$  by a factor of  $M$  makes early stopping impossible. The overhead in the case of  $pH - RA$  comes from additional book keeping costs. From these experiments we conclude that for the case of  $pH - RA$  even a small decrease in the precision parameter can bring significant performance advantages and an observed zero loss in accuracy.

The basic premise of early stopping in any TA based approach is that correlations are present in the data. This premise however may not always be true. To complete our evaluation of the proposed algorithm, we conduct the experiments with ‘adverse data’ without any correlation between the hierarchy and the list positions. For this set of experiments, we generate the lists by starting with an initial list, and introducing a number of random swaps (same number as the list size) of consecutive elements to generate the next list. The hierarchy was generated by grouping random elements from the base lists to the same higher level term. This grouping was done so that the multiplicities follow a Normal distribution with specified mean value (varied in the graph), and variance equal to one third the mean value.



**Figure 8: Improvement factor (reduction in number of lists that need to be aggregated) by using the sparse set system for different values of the parameter  $l$ .**



**Figure 9: Time required for merging 100 lists with and without the use of sparse set system for preprocessing.**

The observed precision values for different settings of the precision parameter of  $pH - RA$  are shown in Figure 6. Observe that the Naive algorithm has precision 100%, and all others are close to this value. Figure 7 shows the performance of the algorithms for this experiment. For high values of the target precision parameter specified in  $pH - RA$  the observations are similar as before, and the source of  $pH - RA$  overheads the same. The performance of  $pH - RA$  improves gracefully as the mean multiplicity decreases

due to less aggressive overestimation in  $maxPossibleScore$ . For a low precision parameter specified in  $pH - RA$ , the algorithm performs well (20-30% savings in running times). Even in this case, the observed precision was much higher than 80%.

### 8.2 Sparse Interval Set

When we use the sparse set system for preprocessing, the number of lists that we merge at query time is much less than the number otherwise required. We refer to the ratio between the two as the *improvement factor* (since running times are proportional to number of lists aggregated). Figure 8 shows the average value of the improvement factor for different values of parameter  $l$  as we increase the query size. The improvement is in orders of magnitude when the query size is large. Note that, the accuracy is not affected by using the sparse set system, and hence is not reported.

Figure 9 shows running times for merging 100 lists each with half a million elements and using a randomly generated hierarchy having mean multiplicity six, with and without the use of the sparse set system. For the case of the sparse set system, the value of  $l$  was set to 2. Query time is reduced significantly when preprocessing is used.

### 8.3 Real Data

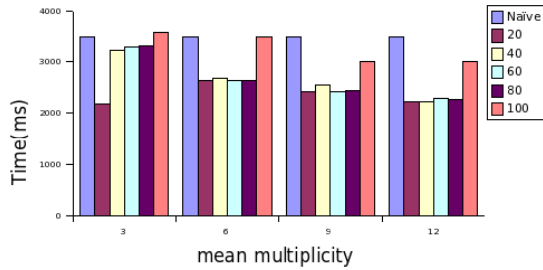


Figure 10: Running times for merging 90 lists obtained from BlogScope with correlated hierarchy.

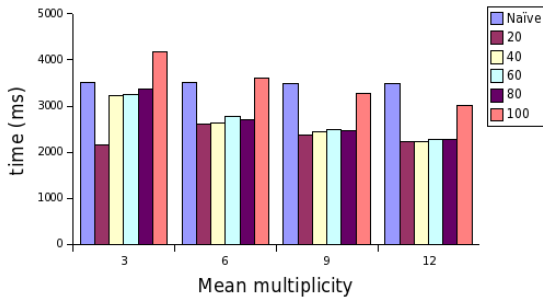


Figure 11: Running times for merging 90 lists obtained from BlogScope with uncorrelated hierarchy.

To evaluate our proposed techniques in a real world setting, we used BlogScope to generate 90 lists, each list corresponding to a single day, for a 90 day period. Each list had 50K keywords ranked according to BlogScope’s keyword popularity ranking methodology on the corresponding day in the blogosphere. We first show experiments using these real lists with synthetically generated hierarchies. Experimental results with ‘real’ hierarchies are reported later in this section. Figure 10 and Figure 11 present performance results for experiments conducted using these data. For Figure 10

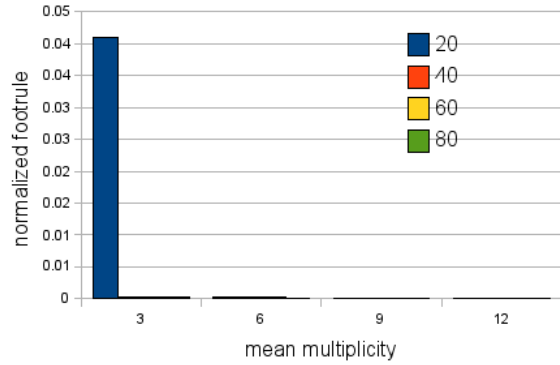


Figure 12: Normalized footrule distance (on y-axis) for merging 90 lists obtained from BlogScope with correlated hierarchy for different mean multiplicity (on x-axis) and target precision values.

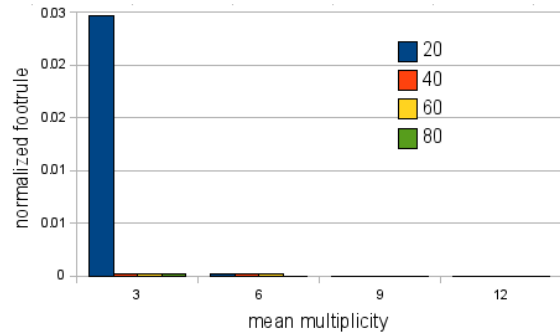
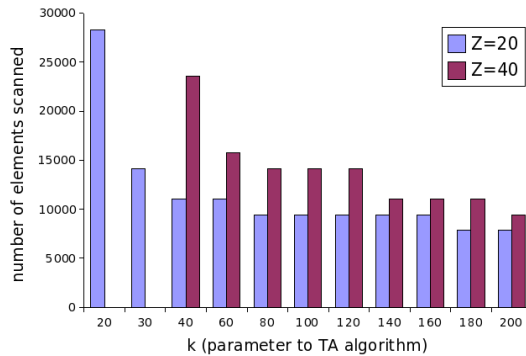


Figure 13: Normalized footrule distance (on y-axis) for merging 90 lists obtained from BlogScope with uncorrelated hierarchy for different mean multiplicity (on x-axis) and target precision values.

we used a ‘correlated’ hierarchy, i.e., elements mapping to the same higher level term appear near each other in the lists. Figure 11 displays results for the case of ‘uncorrelated’ hierarchies (keywords randomly mapping to hierarchy nodes). The observed precision was 100% in all the cases. To study the amount of disarray between the observed and the actual top- $k$  result, we computed the Spearman’s footrule distance between the two lists. Footrule metric measures the distance between two permutations by comparing the position of elements in them [13]. We report footrule distance values between the actual and the observed top- $k$  result in Figures 12 and 13 for correlated and uncorrelated hierarchies respectively. Footrule values reported herein were computed by first calculating the aggregate distance between position of elements in the two lists, and then normalizing to the range  $[0, 1]$  such that two completely different lists have normalized footrule distance equal to 1.0. It can be observed that the footrule distance drops sharply as target precision is increased. In most cases, the distance was zero implying that result obtained by  $pH - RA$  was identical to the correct result. This demonstrates that the proposed algorithm with relaxed precision requirement results in significant runtime savings for little or no loss in accuracy.

In our next experiment, we explore the dependency between  $k$  and the user specified absolute value ( $z$ ) of correct top- $k$  results. This relationship provides an insight on the choice of  $k$ . For a fixed

value of  $z$ , we run the TA algorithm (with different values of  $k$ ) till the correct top- $z$  elements are present in the current top- $k$  (in the buffer of the TA algorithm), and record the number of elements scanned in each list. In other words we stop when top- $k$  elements in the buffer contain all of actual top- $z$  elements. Results are reported in Figure 14 for  $z = 20$  and  $z = 40$ . When  $z = k$ , the problem is the same as  $H - RA$ , as precision requirement is 1.0. It can be observed that relaxing the precision even slightly (i.e., setting  $k = 30$  when looking for top-20) results in significant performance gains. Increasing the value of  $k$  further exhibits a diminishing returns phenomenon. Empirically, a value of  $k \sim 3z$  is enough to obtain the maximum performance gains at high accuracy.

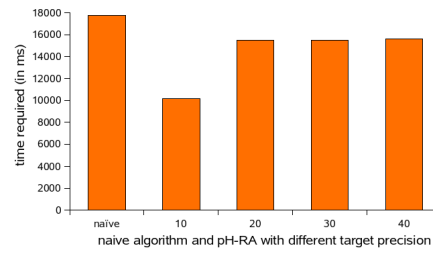


**Figure 14: Number of elements that need to be scanned in each list so that actual top- $z$  results are contained in  $currTopK$  in the buffer.**

Our final experiment presents the performance of our algorithm on lists generated from BlogScope using hierarchies generated from real data. The hierarchy is generated by conducting a specific form of keyword clustering to group together keywords that correspond to the same event as proposed in [3]. Thus our top- $k$  problem seeks to identify highly popular events. Examples of such event clusters include  $\{iraq, bush, war, troops\}$  (corresponds to the war in Iraq),  $\{iphone, ipod, apple, macworld\}$  (Apple and iPhone launch), and  $\{beckham, soccer, david, mls\}$  (David Beckham in Major League Soccer). Each keyword was mapped to exactly one cluster id, thus elevating the keyword to the event corresponding to the cluster. Figure 15 shows the running times of the naive algorithm and compares it with that of  $pH - RA$ . A target precision of 0.1 was able to achieve an actual precision value of 0.9 while running almost twice as fast as the naive algorithm. For all other  $\rho$  values, precision=1.0 was observed. Performance can be improved further by making use of the sparse interval set system.

## 9. CONCLUSIONS

We studied the problem of ranked list aggregation in the presence of hierarchies. We presented a probabilistic analysis of early stopping approaches in this setting. We show that the probabilistic approach, building upon similar techniques proposed for top- $k$  [22], is not practically feasible. The cost of computing the exact top- $k$  solution deterministically is often very high, and hence is not practically acceptable either. We therefore introduced a relaxed version of the rank aggregation problem involving a deterministic stopping condition with user specified precision. We introduced an algorithm  $pH - RA$  for the solution of this problem. In addition we introduced additional techniques to improve the performance of  $pH - RA$  even further via precomputation utilizing a sparse set system. Through a detailed experimental evaluation using synthetic



**Figure 15: Using a hierarchy generated from real data for merging 90 lists obtained from BlogScope. The graph compares the naive algorithm and  $pH - RA$  with different target precision requirements. Precision was observed to be 90% for  $\rho = 10\%$ , and 100% for all other cases.**

and real datasets we demonstrated the efficiency of our framework and established the fact that relaxing precision requirements results in significant performance gains. Future work will explore variants of the basic problem proposed herein, in which dynamic roll-ups and drill-downs between different levels of the hierarchies is supported. This would enable a more interactive analysis setting in the temporal dimension.

## 10. REFERENCES

- [1] B. Arai, G. Das, D. Gunopulos, and N. Koudas. Anytime measures for top- $k$  algorithms. In *VLDB*, pages 914–925. ACM, 2007.
- [2] Aslam, J. A., and Montague, Mark. Models for metasearch. In *SIGIR*, 2001.
- [3] N. Bansal, F. Chiang, N. Koudas, and F. W. Tompa. Seeking Stable Clusters in the Blogosphere. In *VLDB*, 2007.
- [4] N. Bansal and N. Koudas. BlogScope: A System for Online Analysis of High Volume Text Streams. In *VLDB*, 2007.
- [5] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. IO-top- $k$ : Index-access optimized top- $k$  query processing. In *VLDB*, 2006.
- [6] J. O. Berger. *Statistical Decision Theory and Bayesian Analysis*, 2nd Edition. Springer, 1985.
- [7] BlogScope. <http://www.blogscope.net/about/>.
- [8] K. Chakrabarti, V. Ganti, J. Han, and D. Xin. Ranking objects based on relationships. In *SIGMOD*, 2006.
- [9] W. W. Cohen, R. E. Schapire, and Y. Singer. Learning to order things. *JAIR*, 10:243–270, 1999.
- [10] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top- $k$  queries using views. In *VLDB*, 2006.
- [11] M. Dubinko, R. Kumar, J. Magnani, J. Novak, P. Raghavan, and A. Tomkins. Visualizing tags over time. In *WWW*, pages 193–202. ACM, 2006.
- [12] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *WWW*, pages 613–622, 2001.
- [13] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top  $k$  lists. *SIJDM: SIAM Journal on Discrete Mathematics*, 17, 2003.
- [14] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *JCSS: Journal of Computer and System Sciences*, 66, 2003.
- [15] W. Feller. *An Introduction to Probability Theory and its Applications*, volume 1. John Wiley & Sons, 2nd edition, 1957.
- [16] M. F. Fontoura, R. Lempel, R. Qi, and J. Zien. Inverted index support for numeric search. In *Internet Mathematics*, 2006.
- [17] S. Guha, N. Koudas, and D. Srivastava. Fast algorithms for hierarchical range histogram construction. In *ACM PODS*, 2002.
- [18] U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *VLDB*, pages 419–428. Morgan Kaufmann, 2000.
- [19] A. Marian, N. Bruno, and L. Gravano. Evaluating top- $k$  queries over web-accessible databases. *ACM Trans. Database Syst*, 29(2):319–362, 2004.
- [20] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, 1999.
- [21] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513–523, 1988.
- [22] M. Theobald, G. Weikum, and R. Schenkel. Top- $k$  query evaluation with probabilistic guarantees. In *VLDB*. Morgan Kaufmann Publishers, 2004.
- [23] R. R. Yager and V. Kreinovich. On how to merge sorted lists coming from different web search tools. *Soft Comput*, 3(2):83–88, 1999.