Top-k Query Evaluation with Probabilistic Guarantees

Martin Theobald, Gerhard Weikum, Ralf Schenkel
Max-Planck Institute of Computer Science
D-66123 Saarbruecken, Germany
{mtb, weikum, schenkel}@mpi-sb.mpg.de

Abstract

Top-k queries based on ranking elements of multidimensional datasets are a fundamental building block for many kinds of information discovery. The best known general-purpose algorithm for evaluating top-k queries is Fagin's threshold algorithm (TA). Since the user's goal behind top-k queries is to identify one or a few relevant and novel data items, it is intriguing to use approximate variants of TA to reduce run-time costs. This paper introduces a family of approximate top-k algorithms based on probabilistic arguments. When scanning index lists of the underlying multidimensional data space in descending order of local scores, various forms of convolution and derived bounds are employed to predict when it is safe, with high probability, to drop candidate items and to prune the index scans. The precision and the efficiency of the developed methods are experimentally evaluated based on a large Web corpus and a structured data collection.

1. Introduction

1.1 Motivation

Top-k queries on multidimensional datasets compute the k most relevant or interesting results to a partial-match query, based on similarity scores of attribute values with regard to elementary query conditions and a score aggregation function such as weighted summation. This fundamental building block for information discovery arises in many important application classes such as 1) Web and intranet search engines with scores based on word-occurrence statistics and possibly combining criteria like text-based relevance, link-based authority, and recency, 2) multimedia similarity search on feature vectors of images, music, or video, or 3) preference queries over structured and semistructured data such as product catalogs or customer support data (the latter having a major text component as well).

The best known general-purpose method for top-k queries is *Fagin's threshold algorithm*, also known as *TA* [19], which has been independently proposed also by Nepal et al. [33] and Güntzer et al. [22]. This method assumes that each attribute of the multidimensional data space has an index list by which one can access the data items in descending order of the "local" score for the given attribute with regard to an elementary query condition (e.g., tf*idf-based score for a text keyword condition "Trumpet", ontological similarity for categorical attribute conditions such as Genre=Jazz, or absolute distance for numerical attribute conditions such as Year=1970).

The TA method is conservative in that it stops scanning index lists only when it is certain that no more top-k results can be found. We believe that this is overly conservative given that the concept of a top-k query is heuristic anyway. Hardly any end-user would be interested in looking at exactly the k best matches to a similarity query. Rather the rationale of top-k ranking is that users typically find one or a few relevant and novel data items among the top 10 or 20 results. So there is an inherent and unavoidable risk of missing the truly best results (in the subjective judgment of the user) anyway. This in turn justifies relaxing the concept of a top-k query into an *approximate* notion such that the query processor can occasionally tolerate errors: false positives or false negatives with regard to the top k.

The idea of approximate top-k queries has been around in the literature (see, e.g., [3,5,13,17,19]). However, in terms of analyzing how much is lost by the relaxation the prior work either introduced control parameters that are difficult to tune or are based on homogeneity assumptions for multidimensional data distributions. The main focus of the earlier work was image similarity search over color, texture, and contour feature spaces, where the assumptions may indeed be justified. Furthermore, the relaxation control parameters (e.g., a distance slack factor) of these models were difficult to translate into user-perceived guarantees. In contrast, this paper presents a principled approach to approximate top-k queries with *probabilistic guarantees* about the error relative to "exactly top-k" queries, translatable into guarantees about query-result precision and recall. Our approach can cope with heterogeneous distributions where the score variability may radically differ among different text terms or attributes of a semistructured dataset.

We concentrate on algorithms that process index lists by *sorted access only*, as we are aiming at high-dimensional data spaces such as Web or XML documents where queries need to access a potentially large number of very long index lists and random accesses would be very expensive. Our approach allows much more aggressive index list pruning, compared to the original TA method

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

with sorted access. The paper presents comprehensive experimental results that demonstrate the performance gains.

1.2 Related Work

The state of the art on top-k queries has been defined by the seminal work on the threshold algorithm (TA) [18,19, 22,23,33]. TA scans all query-relevant index lists in an interleaved manner, and aims to compute "global" scores for the encountered data items by means of a monotonic score aggregation function such as (weighted) sum, max, etc. The algorithm maintains the worst score among the current top-k results and the best possible score for all other candidates and items not yet encountered. The latter serves as a threshold for stopping the index scans when no candidate can exceed the score of the k^{th} ranked result. The algorithm comes in two variants. *TA-random* eagerly looks up all local scores of each encountered item and thus knows the full score immediately when it first encounters the item. Since random accesses may be expensive and, depending on the application setting, sometimes infeasible, the alternative *TA-sorted* method (coined NRA in [19] and Stream-Combine in [23]) maintains worst-score and best-score bounds for data items based on partially computed global scores, and its stopping test compares the worst-score of the k^{th} ranked result with the best-score of all other candidates. Obviously, TA-random is more effective in pruning the index scans, but TA-sorted avoids expensive random accesses.

Variants of TA have been studied for multimedia similarity search [12,31], ranking query results from structured databases [1], and distributed preference queries over heterogeneous Internet sources such as digital libraries, restaurant reviews, street finders, etc. [11,29,38]. Marian et al. [29] have particularly investigated how to deal with restrictive sources that do not allow sorted access to their index lists and with widely varying access costs. To this end, heuristic scheduling approaches have been developed, but the threshold condition for stopping the algorithm is a conservative TA-style test. Other top-k query algorithms in the literature include nearest-neighbor search methods based on an R-tree-like multidimensional index [3,6,14,24,25] and mapping techniques onto multi-dimensional range queries [8] evaluated on traditional database indexes. In this context, probabilistic estimators for selection cutoff values have been developed by [13,17, 36] and applied to multidimensional nearest-neighbor queries.

Most of the above TA-centric work has studied the TA-random variant. TA-sorted, on the other hand, has been regarded as the less attractive variant to which one would resort only under specific circumstances. However, with a large number of potentially very long index lists TA-sorted should actually be the method of choice. A small number of papers have considered how to minimize random-access costs in TA-random when data sources vary in speed and selectivity [11,29,38]. To this end simple, histogram-based, probabilistic estimators have been developed for making scheduling decisions (i.e., deciding on which source (i.e., dimension) the next random access should be made). None of this prior work has attempted a principled approach to probabilistic score prediction and result guarantees.

Efficient processing of index lists for ranked retrieval is an old topic in information retrieval (IR) research [30]. In this context sorted-access only is the rule of the game. The pruning techniques considered here (see also [4,10, 21]) are heuristic in nature in that they trade off some loss in result quality (effectiveness in IR jargon) for speed without being able to predict and control the resulting effects (other than by experimentation). For the special but important case where the global score is a weighted sum of tf*idf-based text relevance and link-based, and thus query-independent, authority, additional pruning heuristics have been developed in [7,28].

1.3 Contribution

Our approach is based on predicting the total score of a candidate item for which we know a partial score (e.g., the sum of local scores for one or more elementary conditions, but not the total score for all conditions). In doing this we avoid the overly conservative best-score/worst-score bounds of the original TA-sorted method by calculating the probability that the total score exceeds a threshold that would make the item interesting for the top-k result. If this probability is sufficiently low, we drop the data item from the candidate list. The probabilistic prediction involves computing the convolution of the score distributions of different index lists. To this end, we explore a variety of techniques including histograms, efficiently evaluable Poisson estimations, and convolutions based on moment-generating functions with generalized Chernoff-Hoeffding bounds [32,35] for the resulting tail probabilities. As the overhead of these techniques is crucial, the details of our bookkeeping and candidate testing strategies are all but straightforward; we explore a range of strategies based on different setups of priority queues.

To the best of our knowledge, this is the first paper that presents a method for probabilistic top-k queries. Note that our probabilistic guarantees are not about query run-times but about query result quality; run-time bounds that hold with high probability have been derived in [18]. Also, our approach should not be confused with probabilistic methods for deriving local and global scores, e.g., probabilistic IR techniques [16]; we can handle a wide variety of scoring functions as building blocks but our notion of probabilistic guarantees refers to the approximation of the top k ranks in a completely scored and exactly ranked result set.

2. Computational Model

Consider a *Cartesian product space* $D_1 \times \ldots \times D_m$ over domains $D_1, \ldots, D_m$, and a dataset $D \subseteq D_1 \times \ldots \times D_m$ of m-dimensional data points. The data points could be *records* over structured domains (e.g., product catalog entries) or

text *documents* when the domains capture weights of text terms in a high-dimensional IR feature space. In the latter case $D \subseteq \Re^m$ or $D \subseteq [0,1]^m$.

Queries on such a dataset are *partial-match queries* in the form of m-tuples $(q_1, \ldots, q_m)$ where $q_i \in D_i$ or q_i=* meaning that we do not care about the i^{th} dimension value. In the text-document IR case, usually a few q_i values are set to 1 (the query keywords) and all others set to 0 (but there are other approaches as well). The results $d \in D$ to a query $q = (q_1, \ldots, q_m)$ do not necessarily have to match all non-zero q_i values or have non-zero components d_i; rather we would like to retrieve *approximate matches* to this condition according to some similarity measures. We assume that for each domain D_i there is a *similarity function* s_i: $D_i \times D_i \rightarrow [0,1]$. For numerical domains such as year or price, the similarity function could simply be the absolute difference; for categorical domains such as model (of cars) or genre (of movies or books) the similarity function needs to be explicitly defined for all value pairs.

The domain-specific similarity functions are *aggregated* into a *global similarity* function s: $(D_1 \times \ldots \times D_m) \times (D_1 \times \ldots \times D_m) \rightarrow [0,1]$, with $s(x,y) = aggr\{s_i(x_i,y_i) \mid i=1..m\}$ where *aggr* is an aggregation function from $[0,1]^m$ to [0,1]. A widely used aggregation function for this purpose would be summation, yielding $s(x,y) = \sum_{i=1}^{m} s_i(x_i,y_i)$; popular alternatives include weighted summation, product (with a probabilistic interpretation), or maximum. The result of a top-k query q then is given by the k data points d for which sim(q,d) is among the k highest values of all similarities between q and any data points.

Our framework for processing top-k queries is based on *sorted access* to the data in descending order of similarity scores for each dimension. Especially for processing IR index lists, where the lists for frequent text terms may be very long, random lookups into these lists would incur extra disk IOs which are orders of magnitudes more expensive than a sorted-access step in memory (with occasional asynchronous and sequential disk reads). TA-sorted operates on *lists* L_i, i=1..m, which, for a given query value q_i, return data values y_i in descending order of $s_i(q_i,y_i)$. The implementation uses B-tree indexes and scans the inverted index lists in sorted order of scores for individual keys, or looks up keys that exactly match a condition and then merges the results of a forward and a backward scan for neighboring keys (e.g., numerical attribute values with small absolute distance to the query value).

In this computational model, the TA-sorted algorithm, which will be our baseline, can be written in the compact pseudocode form shown in Figure 1. Note that, unlike TA-random, the algorithm requires remembering all non-discarded candidates in memory. We will come back to these implementation details in Section 4; neither the original work [19,23] nor the follow-up research [1,9,29] discuss concrete bookkeeping data structures despite their strong performance impact.

When given a query q with specified values $q_1, \ldots, q_m$, we assume, without loss of generality, that all dimensions are specified or, equivalently, consider only the subspace of dimensions for which the query specifies values. We maintain for each index list L_i the following: a current scan position pos_i and a current score $high_i := s_i(q_i, d)$ for the document d at the current scan position in L_i. We maintain for each record or document d that was already encountered in at least one of the L_i: a set $E(d)$ of dimensions for which we already computed a score s_i, and a partial score $worstscore(q,d) := aggr\{s_i(q_i,d) \mid i \in E(d)\}$ (e.g., $\sum_{i \in E(d)} s_i(q_i,d)$ with summation as *aggr*).

```
TA-sorted:
top-k :={dummy_1, ..., dummy_k}; // with s (dummy_v)=0
min-k := 0;
candidates := ∅;
scan all lists L_i (i = 1..m) in parallel:
// e.g., round-robin or merged in descending order of s_i values
   consider item d at position pos_i in L_i;
   if d ∉ candidates then
      candidates := candidates ∪ {d};
      E(d) := {i};
   high_i := s_i(q_i,d); // current score in L_i
   E(d) := E(d) ∪ {i};
   bestscore(d):=aggr{aggr{s_v(q_v,d)|v∈E(d)},
                      aggr{high_v|v∉E(d)}};
   worstscore(d) := aggr{s_v(q_v,d)|v∈E(d)};
   if worstscore(d) > min-k then
      if d ∉ top-k then
         remove argmin_d'{worstscore(d')|d'∈top-k} from top-k;
         candidates := candidates ∪ {d'};
         add d to top-k
      min-k := min{worstscore(d') | d' ∈ top-k};
    if bestscore(d) ≤ min-k then candidates := candidates – {d};
    threshold := max {bestscore(d') | d'∈ candidates};
    if threshold ≤ min-k then exit;
```

Figure 1: Pseudocode for TA-sorted algorithm

All algorithms are based on the invariant $worstscore(q,d) \leq s(q,d) \leq bestscore(q,d)$ with $bestscore(q,d) := aggr\{worstscore(q,d), aggr(high_i \mid i \notin E(d))\}$ For the case of sum as the aggregation function this becomes $\sum_{i \in E(d)} s_i(q_i,d) \leq s(q,d) \leq \sum_{i \in E(d)} s_i(q_i,d) + \sum_{i \notin E(d)} high_i$.

Suppose we already have k items T that are currently the top k results of a given query q, and let $min_k = min\{s(q,d) \mid d \in T\}$ or, in the case that the items in T have not been fully evaluated, $min_k = min\{worstscore(q,d) \mid d \in T\}$. Then we can prune documents and remainders of index lists for documents whose upper bound cannot exceed the value min_k, i.e., a document d can be dismissed from the candidate set if

$$aggr\{worstscore(q,d), aggr\{high_i \mid i \notin E(d)\}\} < \min_k$$

In this case we say that the *threshold test* fails. This consideration is often unnecessarily conservative, because the expected remainder score of a document is much lower than the sum of the $high_i$ bounds. Of course, using expectations for pruning would not give us any guarantees for not missing any of the true top-k results. But we would expect that the sum of the s_i scores in the remainder set

{1...m} – E(d) is lower than the sum of the $high_i$ bounds with very high probability. So we are interested in estimating the probability that a document *d* that we encounter at position pos_i in the index list L_i and for which *E(d)*=∅ holds qualifies for the top k results:

$$P\left[aggr\{worstscore(q,d), aggr\{s_i(q_i,d) \mid i \notin E(d)\}\} > \min_k\right]$$

If this probability is below some threshold ε (e.g., between 1 and 10 percent) then we may decide to disregard d, without computing its full score. When we choose summation as aggregation function, the above expression becomes

$$p(d) := P\left[\sum\{s_i \mid i \in E(d)\} + \sum\{s_i \mid i \notin E(d)\} > \min_k\right]$$

or equivalently, with $\delta(d) := \min_k - \sum\{s_i \mid i \in E(d)\}$:

$p(d) = P\left[\sum\{s_i \mid i \notin E(d)\} > \delta(d)\right]$. We refer to this condition as the *probabilistic threshold test*. When we compute p(d) during query execution we know upper bounds $high_i$ for the unknown scores. So more precisely, the equation for p(d) should read as

$$p(d) = P\left[\sum\{s_i \mid i \notin E(d)\} > \delta(d) \mid s_i \le high_i \ \ for\ i \notin E(d)\right]$$

3. Probabilistic Score Prediction

In this section we develop the details for estimating the probability p(d) that a document d with non-empty remainder set *{1..m}–E(d)* may qualify for the top-k results. How we estimate p(d) depends on the assumptions that we make about the distribution of the unknown scores s_i that d would obtain. The following subsections discuss various cases that are of interest from both a fundamental-insight and application viewpoint. We will concentrate on the most important case of using summation for score aggregation, and will discuss generalizations at the end of this section. Note that summation is the standard choice in IR keyword query processing, with tf*idf-style weights being precomputed and stored in the index lists.

3.1 Guarantees with Uniform Distributions

In the absence of any other information, Occam's razor suggests that the simplest assumption about the distribution of unknown partial scores is a *uniform* distribution. More specifically, we assume that for document d and dimension *i*∈*{1..m}–E(d)* that has not yet been evaluated, the score $s_i(d)$ is uniformly distributed between $high_i$, the currently known upper bound for the true score, and 0, the assumed lower bound. Instead of 0 we may also use the lowest value that occurs in L_i, provided we have stored this information in the index metadata (i.e., without having to scan L_i to its end). We use continuous distributions rather than discrete ones, as this simplifies the subsequent calculations. We assume that all random variables S_i are independent; this will be reconsidered later.

Treating each unknown s_i value as a random variable S_i we thus have to predict $P\left[\sum_i S_i > \delta\right]$. For two random variables S_1 and S_2 with densities $f_1(x)=1/high_1$ and $f_2(x)=1/high_2$ this requires computing the convolution $f(x) = \int_0^x f_1(z) f_2(x-z)dz$. Taking into account the fact that each factor is non-zero only within certain intervals, namely, $0 \le z \le high_1$ and $0 \le x\text{-}z \le high_2$, or equivalently $\max(0, x-high_2) \le z \le \min(x, high_1)$, solving the integral leads to the following three cases, assuming $high_1 \le high_2$ (without loss of generality):

$$f(x) = \begin{cases} x/(high_1 \cdot high_2) & for\ 0 \le x \le high_1 \\ 1/high_2 & for\ high_1 < x \le high_2 \\ 1/high_1 + 1/high_2 - x/(high_1 \cdot high_2) & \\ & for\ high_2 < x \le high_1 + high_2 \end{cases}$$

and a corresponding cumulative distribution in efficiently evaluable closed form.

Unfortunately, for three and more heterogeneous uniform distributions, this kind of computation, albeit still simple in principle, leads to a rapidly increasing number of cases regarding integration boundaries that are fairly awkward to handle. Therefore, we rather treat the convolution in terms of moment-generating functions $M_i(s) = \int_0^s e^{sx} f_i(x)dx = E\left[e^{sS_i}\right]$ for random variables S_i with densities $f_i(x)$. With independent variables, the convolution has the moment-generating function $M(s) = \prod_i M_i(s)$ [32]. With uniform distributions $f_i(x)$ plugged in, this yields a function from which we cannot easily infer the density of the convolution. Instead, we apply Chernoff-Hoeffding bounds to the tail probability of the convolution [32,35]: $P\left[\sum_i S_i > \delta\right] \le \inf_{s \ge 0}\{e^{-s\delta} M(s)\}$ where the infimum on the right-hand side is either the minimum of the Chernoff bound function, computed by finding the roots of the first derivative, or a limit (e.g., for s approaching 0). This computation can be automated using computer algebra tools like Maple and its programming interface OpenMaple.

A great advantage of this approach is that it can be generalized to incorporate distributions other than uniform ones. Moreover, it can easily handle heterogeneous distributions with say some scores S_i being uniformly distributed and others following, for example, hyperexponential or Zipf distributions. Finally, using results from [35] we can even handle dependent random variables, although the corresponding generalized Chernoff-Hoeffding bounds may not be as strong as in the standard case. Assume that $S_1, \ldots, S_m$ are our random variables of interest. We construct a set of independent random variables $T_1, \ldots, T_m$ such that T_i has the same distribution as the marginal distribution S_i. For a partitioning $\delta = \delta_1 + \ldots + = \delta_m$ for the tail quantile of interest consider the Chernoff bounds ε_i with $P[T_i > \delta_i] \le \varepsilon_i$. [35] have shown that $P\left[\sum_i S_i > \delta\right] \le \inf_{\delta_1+\ldots+\delta_m=\delta}\{\max\{\varepsilon_i \mid i=1..m\}\}$. While it is difficult to determine the best choice of the partitioning values δ_i, a good heuristic choice (that is guaranteed to yield correct bounds) is to set $\delta_i := \delta \cdot high_i / \sum_i high_i$ (i.e., choose the δ_i values in proportion to the $high_i$ values of

the index lists). Computing these generalized bounds can be programmed as OpenMaple procedures.

3.2 Guarantees with Poisson Distributions

As the computation of convolutions and their bounds using OpenMaple incurs non-negligible overhead, we are also interested in approximations that are computationally cheaper. A form of distribution that has nice theoretical properties, can be efficiently evaluated, and is a reasonable fit for realistic score distributions (e.g., the tf*idf-based score distributions for terms in large Web corpora) is the Poisson distribution. For the fitting to the real distribution we assume that S_i is a discrete random variable with n_i equidistant values $v_j = 1 - j \cdot high_i / n_i$ (j=0..n_i-1) where n_i is the number of object ids in the index list L_i. Then the probability for an object having local score v_k is $P[S_i = v_k] = e^{-\alpha_i} \frac{\alpha_i^k}{k!}$. Here α_i is the parameter that we fit to the actual distribution.

The particularity of the Poisson distribution is that the convolution of m such distributions with parameters α_1, …, α_m is again a Poisson distribution with parameter $\alpha = \alpha_1+\ldots+\alpha_m$, and it is the only distribution with this convenient property [2].

As the $high_i$ values change during the index scans we actually need to predict $P[\sum_{i \notin E(d)} S_i > v \mid S_i \leq high_i \text{ for } i \notin E(d)]$ where v is the largest value smaller than the relevant δ in the virtual value discretization. We lower-bound this probability as follows:

$$
\begin{aligned}
& P\left[\sum_{i \notin E(d)} S_i > v \mid S_i \leq high_i \text{ for } i \notin E(d)\right] \\
= \; & 1 - P\left[\sum_{i \notin E(d)} S_i \leq v \mid S_i \leq high_i \text{ for } i \notin E(d)\right] \\
= \; & 1 - \frac{P\left[\sum_{i \notin E(d)} S_i \leq v \wedge (S_i \leq high_i \text{ for } i \notin (d))\right]}{P\left[S_i \leq high_i \text{ for } i \notin E(d)\right]} \\
\geq \; & 1 - \frac{P\left[\sum_{i \notin E(d)} S_i \leq v\right]}{\prod_{i \notin E(d)} P\left[S_i \leq high_i\right]}
\end{aligned}
$$

For computing the values of the cumulative distribution, we use the efficient numerical method given in [34] based on the Incomplete Gamma Function.

3.3 Guarantees with Histograms

Real score distributions may sometimes be impossible to capture with basic distribution functions and parameter fitting. In such cases the only remaining method is to explicitly track the distribution in the form of a compact histogram. Since histogram construction is not exactly inexpensive, we precompute a histogram for the score distribution of each index list. At query time we first compute the convolution of the query-relevant histograms (and possibly of subsets of them). For simplicity we consider only equi-width histograms, but our approach could be easily generalized to more sophisticated histogram variants [26] (at higher run-time costs, however). For conservative probabilistic predictions we further assume that all values within one histogram cell coincide with the upper bound of the cell.

We choose the same number *n* of cells (e.g., between 10 and 100) for each basic histogram covering the score range (0.0, 1.0], and we use *t·n* cells for the convolution histogram over *t* basic histograms, with the same width *1/n* as the basic histograms covering the range *(0.0, t]*. Cell *i* (for i=0..n-1 or i=0..tn-1) covers the interval *(lb[i], ub[i]]* with *lb[i]=i/n* and *ub[i]=(i+1)/n*. Each cell stores the frequency *freq[i]* and the cumulative frequency *cum[i]* of scores that fall into its interval. The convolution H of basic histograms H_1, ..., H_t is computed by

$$H.freq[i] = \sum_{(i_1,\ldots,i_t) \text{ with } i_1+\ldots+i_t=i} H_1.freq[i_1] \cdot \ldots \cdot H_t.freq[i_t]$$

$$H.cum[i] = \sum_{j=0..i} H.freq[i]$$

The convolution histograms that we compute for a given query capture the complete distribution of possible global scores (*t=m*) and partial scores over unevaluated dimensions (here, we compute the convolution of the basic histograms for the *t=|{1..m}-E(d)|* unevaluated dimensions).

As the index scans proceed, we are actually interested in the conditional probabilities of the form $P\left[\sum_{i \notin E(d)} S_i > \delta \mid S_i \leq high_i \text{ for } i \notin E(d)\right]$ where the $high_i$ values reflect the current positions in the index scans. Obviously dynamically rebuilding the histograms after every sorted access is out of the question. We have three ways of addressing this point. The first option is to conservatively bound the conditional probability, analogously to the Poisson approximation model:

$$P\left[\sum_{i \notin E(d)} S_i > \delta \mid S_i \leq high_i \text{ for } i \notin E(d)\right] \geq 1 - \frac{P\left[\sum_{i \notin E(d)} S_i \leq \delta\right]}{\prod_{i \notin E(d)} P\left[S_i \leq high_i\right]}$$

which can be directly looked up in the precomputed histograms.

The second option is to start with the full convolution histograms and dynamically "undo" the terms that contribute to *H.freq[i]* as the $high_i$ values change during query execution. Suppose that $high_j$ changes from some value *ub[k]* to *ub[k-1]*. Then we modify all *H.freq[i]* values with $v_i \leq \sum_i high_i$ as follows: $H.freq[i] =$

$$H.freq[i] - \sum_{\substack{(i_1,\ldots,i_t) \\ \text{with } i_1+\ldots+i_t=i \\ \text{and } i_j=k}} H_1.freq[i_1] \cdot \ldots \cdot H_t.freq[i_t].$$

The subtrahend is also precomputed and additionally stored in cell k of the histogram for index list L_j. The computational overhead for the dynamic maintenance is O(tn) whenever one of the index scans crosses a histogram cell boundary, but the, less critical, precomputation cost and the space for each histogram increase considerably ($O(t^2n^2)$ space instead of O(tn)).

Finally, the third way is to periodically recompute the histograms, after every *r* sorted accesses with *r* being in the order of a few hundred. Each time a convolution histogram is rebuilt from the precomputed basic histograms,

the current $high_i$ values are taken into account; so the recomputation becomes cheaper and the histogram smaller as the index scans proceed towards lower local scores.

3.4 Extensions and Generalizations

Our framework for probabilistic predictions could be extended in three ways: 1) further classes of score distributions, 2) correlated local scores, and 3) more general score aggregation functions other than summations.

As for score distributions, we can accommodate a wide variety of distributions into the Chernoff-Hoeffding bound approach discussed in Subsection 3.1. For example, it would be straightforward to incorporate Zipf distributed scores where $P[S_i = v_k]$ for equidistant values v_k is proportional to 1/k and the cumulative distribution corresponds to the harmonic series, and we can also easily handle heterogeneous mixes of different distributions, say uniform for some index lists but Zipf for some highly skewed ones. For the histogram approach of Subsection 3.3, more general distributions are a non-issue, because histograms are approximations of arbitrary distributions.

As for correlations between the local scores from different index lists, the generalized Chernoff-Hoeffding bounds already provide an approach. The histogram approach, on the other hand, would have to use multidimensional histograms to capture joint distributions. We are not convinced that this is practically viable except for specialized settings. Multidimensional histograms over all index lists may be very space-consuming or sparse or inaccurate, and the subspace that is relevant for a given query is known only at query time when histogram building would already be part of the user-perceived response time. Fitting a parameterized multidimensional distribution, e.g., a multivariate Normal distribution, to the data seems more promising, but the decision for a particular type of distribution function would have to be carefully justified.

Finally, using monotonous score aggregation functions beyond simple sums is already supported, to a large extent, within our framework. A large class of aggregation options can simply be cast into the precomputation of local scores, so that the actual aggregation step again becomes a simple summation. For example, with weighted summation the weights for each dimension can be factored into the local scores; IR-style tf*idf-based scores are of this type for idf values can be viewed as dimension weights. Also note that cosine similarity in IR is usually reduced to summation (i.e., scalar products between document and query vectors) by pre-normalizing (L_2 norm) vector lengths to 1. Using maximum for score aggregation is even simpler than summation; instead of computing the convolution of several S_i distributions, we merely compute $P[\max_i S_i > \delta] = 1 - \prod_i P[S_i \le \delta]$.

4. Query Evaluation Algorithms

Our query processing algorithms use the probabilistic models as predictors for the global scores of data objects that have not been fully evaluated or not seen at all in the index scans so far. Based on this central building block we have developed several algorithms that differ a) in their selection of candidates to which they apply the probabilistic predictions and b) in their actions that they take when a threshold test for a candidate fails (i.e., the candidate is unlikely to be able to qualify for the top-k result). All algorithms maintain the set of current top-k objects and the set of candidates organized as a hash table based on object ids.

4.1 Conservative Algorithm

A naive algorithm would simply predict the scores of all candidate objects in every step of the index scans and drop all candidates whose probabilities of qualifying for the top-k result are sufficiently low. This would incur very high overhead for probabilistic threshold tests; moreover, the score prediction for an object d would have to be recomputed whenever one of the $high_i$ values in the set $\{1..m\} - E(d)$ changes. A better way is to group the candidates by their $E(d)$ values, placing all objects with the same set of evaluated dimensions into one partition using their *bestscore* as priority. Then, it suffices to test the best object per group, i.e., the one with the highest predicted score. This object dominates all other candidates in the same group in terms of the probability of qualifying for the top-k result. Across groups, however, the top objects are not directly comparable.

Based on this observation, the *conservative algorithm* maintains a priority queue for each $E(d)$ group, up to $2^m - 1$ queues for a query with m specified conditions. Note that m is often much smaller than the dimensionality of the data space (e.g., for keyword queries over a text document space). The priority queues merely contain pointers to the hash-table entries of the candidate objects.

As an item d is evaluated at the current scan position pos_i in index list L_i, the conservative algorithm deletes d from its obsolete E(d) queue and, if its *worstscore* still fails the threshold *min-k*, inserts it into the $E(d) \cup \{i\}$ queue using its updated *bestscore* as priority; the insertion is possible with cost O(log n) using a binomial heap or amortized cost O(1) using a Fibonacci heap [15]. If $E(d) \cup \{i\} = \{1..m\}$, i.e., d is completely evaluated, d is dropped from the candidate list.

For periodic index pruning, e.g., after every r = 200 index-scanning steps, the top elements of all queues are probabilistically tested against the current threshold min-k; when a top element fails the test, then all elements of that queue are *dropped* from the candidate list. The algorithm proceeds with fewer candidates and eventually stops when all queues have become empty. Note that there is also one queue for $E(d) = \emptyset$ with a single *virtual element*, capturing the predicted score for an object that has not been seen at all so far.

Advancing the scan pointer in one index list may affect the priorities of other candidates, too, namely by possibly reducing the $high_i$ value of one dimension. But

within one queue, this change affects all elements in the same way; so we can simply track $\sum\{high_i \mid i \notin E(d)\}$ per queue in space and time O(1).

4.2 Aggressive Algorithm

The *aggressive algorithm* is the extreme opposite of the conservative algorithm. It considers one candidate object for probabilistic testing, namely, a *virtual element* d with E(d)=∅ at the current scan positions. If the score prediction for this object falls below the threshold *min-k*, the algorithm *stops* immediately. The prediction for this unknown object is based on the high scores at the current scan positions only. This item's *bestscore* yields an upper bound for all yet unseen documents, i.e., even without probabilistic pruning this algorithm typically stops before the truly best candidate would fail the *min-k* threshold and, thus, yields an approximate result only. The strength of the aggressive algorithm is its minimal overhead, but we do not expect it to perform well in terms of result precision.

4.3 Progressive Algorithm

In between the overly eager behavior of the aggressive algorithm and the substantial overhead of the conservative algorithm is the *progressive algorithm* that maintains a single priority queue for all candidate objects. Again, the queue elements are ordered by their *bestscore* values. In each step the priority of the current candidate, i.e., the one fetched from the index list, is updated and the queue is accordingly maintained.

The algorithm is conservative for it does not immediately track the *bestscore* changes that result from reduced $high_i$ values in each step, but leaves the *bestscore* values higher than they actually are. Otherwise all queue elements would have to be updated, which would amount to rebuilding the entire queue. An additional implementation trick that we employ is that the queue is periodically traversed, e.g., again after every *r=200* index-scanning steps, and we tentatively compute the up-to-date bestscore values of each queue element based on the current $high_i$ values. All elements that do no longer pass the threshold test are dropped from the queue. The priorities of the "surviving" elements are not updated to avoid a massive batch of queue operations. So this periodic removal of unneeded queue elements can be seen as a kind of *garbage collection* without having to rebuild the entire queue.

In conjunction with the periodic garbage collection, the progressive algorithm invokes the probabilistic predictor for each element of the queue using its up-to-date *bestscore*. All objects that fail this probabilistic threshold test are dropped from the queue. The algorithm stops when its queue becomes empty or the top element's *bestscore* falls below the threshold *min-k*.

4.4 Smart Algorithm

The progressive algorithm could stop earlier if it reconsidered all elements in the priority queue with the changing $high_i$ values reflected in each step. In the *smart algorithm* we periodically rebuild the entire priority queue of current candidates with the currently known $high_i$ values taken into consideration. By default the queue is rebuilt every *r=200* steps. The rebuilding has amortized cost O(n) for n queue elements, using a Fibonacci heap, or O(n log n) with a binomial heap [15]. For an online algorithm operating on very large index lists this cost may still be out of the question. Therefore, the smart algorithm maintains only a *bounded priority queue*. Whenever it is rebuilt only the best *b* elements are kept, with *b* being in the order of a few hundred. Newly encountered data objects are admitted to enlarge the queue until the next rebuild; so the maximum size is actually $b + r$ but every rebuild truncates the size back to *b*.

As the priority queue is fully up-to-date after every rebuild, the smart algorithm can take more aggressive actions than the progressive method with regard to candidate pruning. If the top element of the rebuilt queue does not pass the probabilistic threshold test, the smart algorithm immediately *stops* all index scans and terminates.

4.5 Common Framework

All four algorithms share the same algorithmic skeleton illustrated by the pseudocode of Figure 2. We refer to this code as the *Prob-sorted family of algorithms*. Note that, in addition to the probabilistic predictions and corresponding probabilistic threshold tests, all algorithms also include the original Fagin test to compare the maximum *bestscore* of all candidates against the *worstscore* of the current top-k objects. If this test fails all index scans can be stopped immediately, and this extra test is so light-weight that we can always include it.

```
Prob-sorted (RebuildPeriod r, QueueBound b):
...
scan all lists L_i (i=1..m) in parallel:
   ...same code as TA-sorted...
   // queue management
   for all priority queues q for which d is relevant do
       insert d into q with priority bestscore(d);
   // periodic clean-up
   if step-number mod r = 0 then
       // dropping of queues; multiple unbounded queues
       if strategy = Conservative then
           for all priority queues q do
               if prob[top(q) can qualify for top-k] < ε then
                   drop all elements of q;
       // garbage collection; single unbounded queue
       if strategy = Progressive then
           for all queue elements e in q do
               best(e) := bestscore of e with current high_i values;
               if best(e) < min-k then drop e from q;
               if prob[e can qualify for top-k]<ε then drop e from q;
       // rebuild; single bounded queue
       if strategy = Smart then
           for all queue elements e in q do
               update bestscore(e) with current high_i values;
           rebuild bounded queue with best b elements;
           if prob[top(q) can qualify for top-k] < ε then exit;
       // no queues; greedy threshold approximation
       if strategy = Aggressive then
           if prob[virtual-element qualifies for top-k] < ε then exit;
   if all queues are empty then exit;
```

Fig. 2: Pseudocode for Prob-sorted family of algorithms

Let us finally comment on our implementation of the *TA-sorted* baseline algorithm. The original papers on TA do not specify any concrete data structures for the candidate set and how to determine the best candidate in each step. We decided to implement these aspects analogously to the progressive Prob-sorted algorithm, by maintaining a single priority queue with *bestscore* values as priorities. Like before, we do not update all queue elements when one of the $high_i$ values changes, but only update the element currently encountered in the index scan and perform periodic garbage collection with tentative updates.

5. Guarantees for Top-k Results

The probabilistic predictions in our query processing strategies immediately lead to probabilistic *guarantees* from a user viewpoint if we restrict the action upon a failed threshold test to *dropping candidates* but stop the entire algorithm only if we run out of candidates. This is the situation given in the conservative and progressive algorithms. In this case the probability of missing an object that should be in the true top-k result is the same as erroneously dropping a candidate, and this error, call it p_{miss}, is bounded by the probability ε that we use in the probabilistic predictor when assessing a candidate. For the recall of the top-k result, i.e., the fraction of truly top-k objects that the approximate method returns, this means that $P[recall = r/k] = P[precision = r/k] =$

$$\binom{k}{r}(1-p_{miss})^r \; p_{miss}{}^{(k-r)} \le \binom{k}{r}(1-\varepsilon)^r \; \varepsilon^{(k-r)}$$

with r denoting the number of correct results in the approximate top k. We can then efficiently compute Chernoff-Hoeffding bounds for this binomial distribution. Note that the very same probabilistic guarantee holds for the precision of the returned top-k result, simply because recall and precision use the same denominator k. The predicted expected precision then is

$E[precision] = \sum_{r=0..k} P[precision = r/k] \cdot r/k = (1-\varepsilon)$.

For the strategies that test top elements of priority queues and, upon a failed probabilistic threshold test, *stop* the entire algorithm, carrying over the candidate-error probability ε to an argument about recall and precision guarantees would be more sophisticated.

6. Experiments

6.1 Setup

All strategies of our framework are implemented in a comprehensive testbed, using Java and Oracle9i. We performed experiments on a 3 GHz dual Pentium PC with 2 GB of memory. Index lists were stored in the Oracle database, but were fetched in large blocks and cached in the Java program. We used two different datasets with three different workloads in the experiments.

The *Gov setting* uses the data of the TREC-12 Web Track which uses the .Gov Data Collection. It consists of about 1.25 million documents (mostly HTML and PDF) from a large crawl of the .gov Internet domain. We used the original 50 queries from the Web Track's topic distillation task, which are keyword queries with up to 5 keywords. Examples are "legalization marijuana", "Lewis Clark expedition", "airbag injuries death". For systematic experimentation we studied three variations of the local scores in the index lists: a) the *original* scores computed by *tf * log idf*, with *tf* and *idf* normalized by the maximum *tf* value of each document and the maximum *idf* value in the corpus, respectively, b) randomly assigned scores with a (0,1] *uniform* distribution, c) randomly assigned scores with a *Zipf* distribution starting from low scores, so that low scores are much more frequent

In the *expanded Gov (XGov) setting* we wanted to study the impact of the number of query-relevant index lists and modified the queries by adding synonyms and other strongly related terms to the keywords of a query. These additional terms were taken from the synonym entries and descriptions of the WordNet thesaurus [20], where we manually identified for each original keyword the relevant word sense. This query expansion typically doubled the number of keywords per query; the longest query contained 20 keywords ("legalization marijuana cannabis euphoric drug abuse pot smoke …").

The *Imdb setting* used the data of the Internet Movie Database (http://www.imdb.com) to study our methods' performance on a combination of text and structured attributes. The data contains about 375,000 movies and more than 1,200,000 persons (actors, etc.), and we prepared it into a four-attribute object-relational table with the schema Movies (Title, Genre, Actors, Description) where Title and Description are text attributes and Genre and Actors are set-valued categorical attributes. Genre typically contains 2 or 3 genres, and actors were limited to those that appeared in at least 5 different movies. For similarity scores among Genre values and among actors we precomputed the Dice coefficient for each pair of Genre values and for each pair of actors that appeared together in at least 5 movies. So the similarity for genres or actors x and y is set to $\frac{2(\#movies\ containing\ x\ and\ y)}{\#movies\ with\ x\ +\ \#movies\ with\ y}$, and the index list for x contains entries for similar values y, too. A typical query is $Title \supseteq \{Space\} \wedge Genre \supseteq \{SciFi\} \wedge Actors \supseteq \{Harrison\ Ford\} \wedge Description \supseteq \{Robot, War\}$. We compiled 20 queries of this kind by asking colleagues. Note that our similarity scoring does not require a match to satisfy all conditions.

The algorithms compared in the experiments are the four Prob-sorted methods presented in Section 4:

- *Prob-con:* the conservative algorithm,
- *Prob-agg:* the aggressive algorithm,
- *Prob-pro:* the progressive algorithm, and
- *Prob-smart:* the smart algorithm.

For each of them we considered different options for probabilistic prediction. The baseline against which we compare our methods is:

- *TA-sorted:* the threshold algorithm with sorted access only, in the implementation discussed in Section 4.5.

All algorithms access index lists in round-robin manner and cache large index blocks in memory.

6.2 Evaluation Metrics

For efficiency comparison we collected the following measures:

- *accesses*: # sorted access to all index lists altogether,
- *time*: the wall-clock elapsed time,
- *memory*: the peak level of working memory for priority queues.

For assessing the quality of the approximate top-k query results we collected the following measures:

- *precision:* the fraction of top-k results in an approximate result that belongs to the true top-k result,
- *recall:* the fraction of top-k results in the true result that were returned by the approximate top-k query,
- *rank distance:* the footrule distance [27] between the ranks of the approximate top-k results and their true ranks in the exact top-k result, i.e., $\frac{1}{k}\sum_{i=1..k} |i - truerank(i)|$. We did not use Spearman's rank correlation as we wanted to assess only the top k ranks of the approximate result rather than all ranks, but the Spearman measure requires comparing two permutations of the same sets of possible ranks.
- *score error:* the absolute error for approximate vs. exact top-k scores: $\frac{1}{k}\sum_{i=1..k} \left| score_i^{(approx)} - score_i^{(exact)} \right|$.

Note that precision and recall have identical values in our setup, because they have the same denominator k. The baseline for precision and recall is the top-k result of the exact TA-sorted algorithm.

6.3 Results

6.3.1 Baseline Experiment

In the baseline experiment all probabilistic predictors use histograms with cell width 0.01 (i.e., n=100 bins for each basic histogram). Convolution histograms were precomputed at query initiation time, and the impact of changing $high_i$ values was taken into consideration by periodically (i.e., every 200 sorted-access steps) rebuilding the remaining parts of the convolution histograms.

We set the probabilistic prediction confidence level to 90 percent, that is, ε is set to 0.1. For the smart strategy with a bounded priority queue the queue size was set to b=200 entries. We measured top-k queries with k=20.

Figures 3 and 4 show the performance results for the five algorithms under comparison for the Gov and the Imdb settings, respectively. For Gov the chart is based on the original, tf*idf-derived, scores. We present the three efficiency metrics in terms of benchmark totals over all queries, and the three result-quality metrics as macro-averages over all queries. For the Gov setting micro-averaged values are heavily biased by a few long-running queries; for these queries the performance gains of Prob-sorted over TA-sorted are even significantly higher than the macro-averaged values indicate. Figure 5 shows the corresponding results for the XGov setting, which posed a stress test to the queue management.

	# sorted accesses	elapsed time [s]	max queue size	precision	rank distance	score error
TA-sorted	2263652	148.7	10849	1	0	0
Prob-con	993414	25.6	29207	0.87	16.9	0.007
Prob-agg	20435	0.6	0	0.42	75.1	0.089
Prob-pro	1659706	44.2	6551	0.87	16.8	0.006
Prob-smart	527980	15.9	400	0.69	39.5	0.031

Fig.3: Performance of Prob-sorted vs. TA-sorted for Gov

	# sorted accesses	elapsed time [s]	max queue size	precision	rank distance	score error
TA-sorted	1003650	201.9	12628	1	0	0
Prob-con	463562	17.8	14990	0.71	119.9	0.18
Prob-agg	41821	0.7	0	0.18	171.5	0.39
Prob-pro	490041	69.0	9173	0.75	122.5	0.14
Prob-smart	403981	12.7	400	0.54	126.7	0.25

Fig.4: Performance of Prob-sorted vs. TA-sorted for Imdb

	# sorted accesses	elapsed time [s]	max queue size	precision	rank distance	score error
TA-sorted	22403490	7908	70896	1	0	0
Prob-con	10165677	6448	51893	0.90	10.9	0.038
Prob-agg	133745	2	0	0.35	80.7	0.182
Prob-pro	20006283	1791	12435	0.95	9.3	0.031
Prob-smart	18287636	1066	400	0.88	14.5	0.035

Fig.5: Prob-sorted vs. TA-sorted for XGov

Efficiency. The results demonstrate the significant cost savings that the Prob-sorted family of algorithms can achieve compared to TA-sorted. In terms of the number of sorted accesses the conservative algorithm Prob-con gains more than a factor of two, and the smart algorithm Prob-smart achieves even a factor of four for Gov. In terms of run-times, the two probabilistic algorithms even reduce the cost by an order of magnitude. Note that the run-time is not simply a linear function of the sorted accesses but reflects also cache and queue management costs. For the Gov and Imdb settings Prob-con temporarily even created a larger queue than the TA-sorted baseline using periodic garbage collection, but Prob-con dropped this large queue quickly after initialization and then distributed the remaining candidate items to multiple small queues. Interestingly, the progressive algorithm Prob-pro did not do as well as expected; its capabilities for early pruning are limited and its queue management, which required many insert and delete operations, is a significant cost factor. The aggressive method Prob-agg outperformed all competitors, but as expected, its result quality was rather poor; so we would not really consider it a winner.

For individual queries, especially those that involve very long index lists, the savings are even more impressive. For example, for the Gov query "weather hazard extremes" Prob-con and Prob-smart needed 25802 and 19401 sorted accesses with run-times 0.59 and 0.68 seconds, whereas TA-sorted required 160002 accesses and ran in 55.60 seconds (for k=20, at precision 0.9 and 0.75, resp.). In the Imdb setting, the reductions of sorted accesses were not as high as for Gov but the run-time reductions reached a factor of about 10 at a high macro-averaged precision of 0.71 to 0.75. A typical query like "Genre ⊇ {Western} ∧ Actor ⊇ {John Wayne, Katherine Hepburn} ∧ Description ⊇ {sheriff, marshall}" required 10802 sorted accesses for both Prob-con and Prob-smart with run-times 0.41 and 0.51 seconds, whereas TA-sorted performed 26402 accesses in time 4.92 seconds (for k=20, both at precision 0.7).

Finally, for XGov queries with more keywords per query the overhead for queue management and probabilistic predictions became a truly decisive issue. Among the methods with acceptable to very good precision, Prob-con performed best in terms of sorted-access savings, but the run-time gains were only modest because of the overhead of maintaining up to 2^m-1 queues. Prob-pro and Prob-smart were the clear winners in terms of run-time, with acceleration factors up to 8 compared to TA-sorted. Interestingly, these methods did not save that many sorted accesses but benefited greatly from their efficient queue management.

Result Quality. The measurements of result quality show very good results for Prob-con and Prob-pro and still acceptable results for Prob-smart. Prob-con and Prob-pro achieved nearly 90 percent precision (and the same recall) for the Gov setting. For the Imdb setting, the precision figures were worse, one reason being that Genre scores had a major influence on the overall ranking and the small number of different values led to a fairly discontinuous score distribution with big gaps and many ties, causing some inaccuracy of probabilistic predictions. We will discuss the influence of the various predictors in Subsection 6.3.2.

The other two result-quality metrics show that the user-perceived "loss" of an approximate result actually seems well tolerable. The average rank distance for Prob-con and Prob-pro was only around 16. For *k=20* or higher this seems acceptable, in particular, when we consider that the average rank distance is dominated by a few outliers with very high rank distance. In terms of score error, the loss even seems negligible. So by and large, the objects that are returned by the Prob-sorted algorithms are nearly as good as the exact top-k results.

Again, the results for the Imdb setting were not quite as good as for Gov. We manually inspected a fair number of the results and found that in most cases the results would be considered as good matches by a human user. For example, the query "Genre ⊇ {Thriller} ∧ Actor ⊇ {Arnold Schwarzenegger} ∧ Description ⊇ {robot}" returned top results Terminator3, The 6th Day, Total Recall, Die Hard 2, Star Wars IV, etc. (recall that top-k results do not necessarily have to satisfy all query conditions).

For XGov, Prob-con, Pro-pro, and Prob-smart showed very good precision, rank distance, and score error values.

6.3.2 Sensitivity Studies

We studied the influence of several parameters on the performance of our four *Prob-sorted* algorithms: the probabilistic prediction confidence level 1–ε, the result size k for top-k queries, the number n of bins per basic histogram, and the maximum size b of a bounded priority queue for the smart algorithm.

Figure 6 shows the results for varying the ε parameter (the vertical dashed line is the baseline setting). The curves show that for ε below 5 percent the progressive and smart algorithms achieve only marginal savings. For ε between 5 and 20 percent, on the other hand, these two methods offer excellent benefit/cost ratios. The conservative method performs best according to the theory of probabilistic guarantees. Already small ε values like 1 percent lead to significant cost savings, and even ε values as large as 50 percent, which results in sorted-access savings of more than a factor of 4, still yield 70 percent precision. The aggressive method always exhibits great cost savings, but this is at the expense of precision values of 40 to 50 percent only. Still, this may possibly be the preferred method in applications with tight response time demands.

We also compared the measured precision with the expected precision that we predict as a function of ε according to the formulas of Section 5. For Prob-con and Prob-pro the prediction model is fairly accurate. The absolute difference between predicted and measured precision is only one or two percent for ε values of 0.2 or less; it increases for larger ε but the prediction is conservative in that it lower-bounds the measured precision.

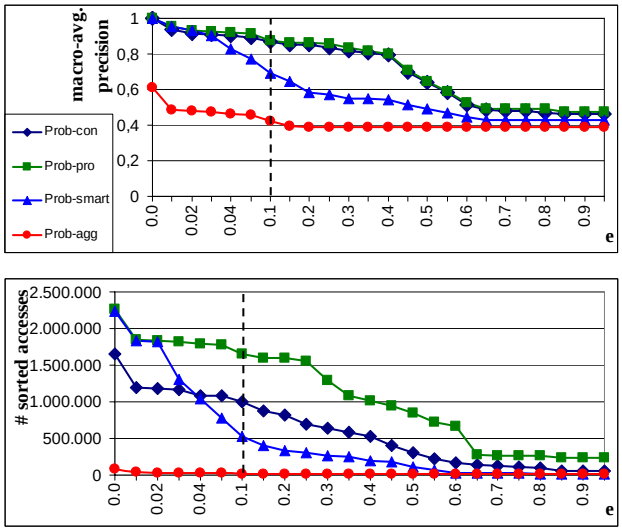


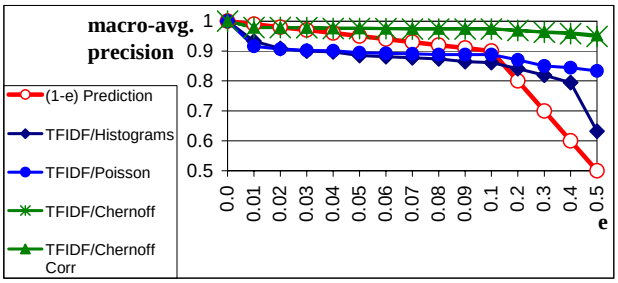
Figure 6: Performance as a function of ε

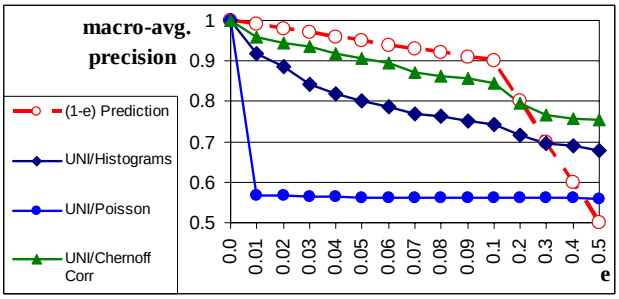
For space limitation we do not include the charts for the sensitivity studies regarding the k, n, and b parameters and briefly discuss only the main insights. With increasing k which was varied between 1 and 200, all methods exhibit linearly increasing sorted-access costs but with different gradients. For large k, the gains of Prob-con, Prob-pro, and Prob-smart compared to TA-sorted are even higher than in the baseline setting; at the same time the precision of the approximate top-k results becomes even better for high k. The performance for different numbers n of histogram bins is fairly stable over a wide range of settings. Between 50 and 1000 bins the relative performance of the different algorithms does not change much; below 50 bins the Prob-smart method does not work that well anymore but Prob-con and Prob-prog remain more robust and show consistently good performance even with down to 25 bins per histogram. Similarly, we found that the maximum queue size parameter b for the Prob-smart algorithm is largely uncritical. Queue size limits as low as b=100 still worked very well for k=20.

6.3.3 Impact of Probabilistic Predictions

Finally, we compared our different approaches for probabilistic prediction: histograms vs. Poisson approximations vs. Chernoff bounds based on the assumption of uniform distributions vs. Chernoff bounds considering term correlations. We limit the presentation to results for the Prob-con algorithm with k=20, ε=0.1, n=100, and the Gov setting. Due to the high overhead of computing Chernoff bounds with OpenMaple, we removed the three most expensive queries from the Web Tracks's topic distillation task which consumed about 40 percent of the overall run time with 50 queries.

Figure 7 shows the performance comparisons for the original tf*idf scores. The dashed line is the predicted precision (for Prob-con this is simply 1–ε). Similar experiments have been run for the artificially generated Uniform- and Zipf-distributed scores on the Gov index lists.





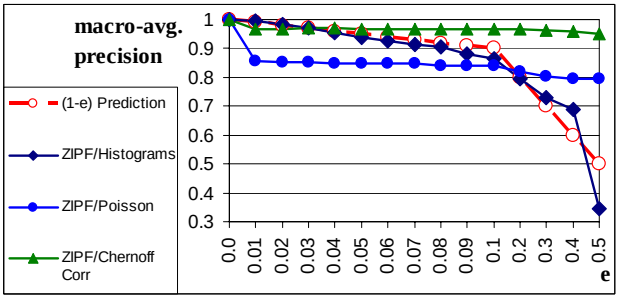


Figure 7: Precision of probabilistic predictors for tf*idf, Uniform-, and Zipf-distributed scores

The charts show that histograms provide the most accurate score predictions. A comparison with the Uniform- and Zipf-distributed scores shows that they are a flexible solution to capture different score distributions already for n as low as 50 to 100 buckets in (0,1]. Both the Poisson estimator and, particularly, the Chernoff-bound method (the latter assuming Uniform-distributed scores) are overly conservative and overestimate score probabilities for the tf*idf and the Zipf case. The difference between the Chernoff-bound methods with and without independence assumption is not really significant for the Gov data, but this could be different in other settings. For Uniform-distributed scores the Chernoff-bounds are fairly accurate over a wide range of ε, whereas, as expected, Poisson estimators do not work for the Uniform case, because they underestimate the tail probability. The Zipf distribution is closer to the original tf*idf score distribution, but has a longer tail of low scores, for which the Poisson estimator works better, but, again, the Chernoff-bounds behave overly conservative.

The advantage of the Poisson approximation method is its very little overhead, whereas the overhead of the histogram method increases with dimensionality. But note that even with the higher-dimensional XGov workload the algorithms still achieved major run-time gains using histograms with dynamic convolutions. The Chernoff-bound predictors are largely independent of the dimensionality, but they suffer from huge startup costs for invoking OpenMaple. For a practically viable solution one would have to hand-code the Maple computations (which involve differentiation and finding roots numerically) in C or C++.

Finally, our model for precision guarantees developed in Section 5 works very well for the Prob-con algorithm. The Prob-pro and Prob-smart algorithms deviate from this basic statistical model, because they merge all candidates into a single queue and Prob-smart even bounds this queue and heuristically stops after testing the top item, only. Here the predictions were reasonably accurate for small ε but degraded and became overly conservative for ε higher than 5 percent. Improving this is subject of future work.

6.4 Discussion

Our comprehensive experiments have shown that Prob-sorted algorithms can achieve major performance gains, in terms of both sorted accesses and actual run-time, and at the same time provide probabilistic guarantees for result precision and recall. Among the four competing algorithms, Prob-con and Prob-smart turned out to be the most interesting ones. Prob-con is closest to the theory of probabilistic guarantees and does best in terms of result quality; Prob-smart offers the best benefit/cost ratio. Both methods achieve run-time gains by an order of magnitude compared to TA-sorted. All four Prob-sorted methods are fairly robust with regard to parameter settings; there is no need for sophisticated tuning. For score predictions, we believe that histograms are the best choice from an engineering viewpoint, but the other two methods showed good results and certainly deserve further studies, too.

7. Concluding Remarks

The novel Prob-sorted family of algorithms that we introduced in this paper are based on two major cornerstones: 1) probabilistic score predictions for trading off a small amount of top-k result quality for a drastic reduction of sorted accesses, and 2) intelligent management of priority queues for efficient implementation. We believe that our experiments have convincingly demonstrated the significant benefits of our approach. We plan to continue the studies of efficient memory management for top-k algorithms, and our future work also includes applying these techniques to ranked retrieval of XML data and integrating them into our XXL search engine [37].

References

[1] S. Agrawal, S. Chaudhuri, G. Das, A. Gionis: Automated Ranking of Database Query Results. CIDR 2003

[2] A.O. Allen: Probability, Statistics, and Queueing Theory with Computer Science Applications. Academic Press, 1990

[3] G. Amato et al.: Region proximity in metric spaces and its use for approximate similarity search. TOIS 21(2), 2003

[4] V.N. Anh, O. de Kretser, A. Moffat: Vector-Space Ranking with Effective Early Termination. SIGIR 2001

[5] K.S. Beyer et al.: When Is "Nearest Neighbor" Meaningful? ICDT 1999

[6] C. Böhm et al.: Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Comput. Surv. 33(3), 2001

[7] S. Brin, L. Page: The Anatomy of a Large-Scale Hypertextual Web Search Engine. WWW Conf. 1998

[8] N. Bruno, S. Chaudhuri, L. Gravano: Top-k selection queries over relational databases: Mapping strategies and performance evaluation. TODS 27(2), 2002

[9] N. Bruno, L. Gravano, A. Marian: Evaluating Top-k Queries over Web-Accessible Databases. ICDE 2002

[10] D. Carmel et al.: Static Index Pruning for Information Retrieval Systems. SIGIR 2001

[11] K.C.-C. Chang, S.-W. Hwang: Minimal probing: supporting expensive predicates for top-k queries. SIGMOD 2002

[12] S. Chaudhuri, L. Gravano, A. Marian: Optimizing Top-K Selection Queries over Multimedia Repositories, to appear in TKDE 2004.

[13] P. Ciaccia, M. Patella: PAC Nearest Neighbor Queries: Approximate and Controlled Search in High-Dimensional and Metric Spaces. ICDE 2000

[14] P. Ciaccia, M. Patella: Searching in metric spaces with user-defined and approximate distances. TODS 27 (4), 2002

[15] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: Introduction to Algorithms. MIT Press, 2001

[16] W.B. Croft, J. Lafferty: Language Modeling for Information Retrieval. Kluwer, 2003

[17] D. Donjerkovic, R. Ramakrishnan: Probabilistic Optimization of Top N Queries, VLDB 1999

[18] R. Fagin: Combining Fuzzy Information from Multiple Systems, J. Comput. Syst. Sci. 58(1), 1999

[19] R. Fagin et al.: Optimal aggregation algorithms for middleware. J. Comput. Syst. Sci. 66(4), 2003.

[20] C. Fellbaum (Editor): WordNet: An Electronic Lexical Database, MIT Press, 1998

[21] N. Fuhr, N. Gövert, M. Abolhassani: Retrieval Quality vs. Effectiveness of Relevance-oriented Search in XML Documents. TR, Univ. Duisburg, 2003

[22] U. Güntzer et al.: Optimizing Multi-Feature Queries for Image Databases. VLDB 2000

[23] U. Güntzer et al.: Towards Efficient Multi-Feature Queries in Heterogeneous Environments. ITCC 2001.

[24] G. R. Hjaltason, H. Samet: Distance Browsing in Spatial Databases. TODS 24 (2), 1999

[25] G.R. Hjaltason, H. Samet: Index-driven similarity search in metric spaces. TODS 28(4), 2003.

[26] Y.E. Ioannidis: The History of Histograms (Abridged). VLDB 2003

[27] M. Kendall, J.D. Gibbons: Rank Correlation Methods. Oxford University Press, 1990

[28] X. Long, T. Suel: Optimized Query Execution in Large Search Engines with Global Page Ordering. VLDB 2003

[29] A. Marian et al.: Evaluating Top-k Queries over Web-Accessible Databases. TODS 29(2), 2004

[30] A. Moffat, J. Zobel: Self-Indexing Inverted Files for Fast Text Retrieval. TOIS 14(4), 1996

[31] A. Natsev et al.: Supporting Incremental Join Queries on Ranked Inputs. VLDB 2001

[32] R. Nelson: Probability, Stochastic Processes, and Queueing Theory, Springer, 1995

[33] S. Nepal, M. V. Ramakrishna: Query Processing Issues in Image (Multimedia) Databases. ICDE 1999

[34] W.H. Press et al.: Numerical Recipes in C. Cambridge Univ.Press, 1992

[35] A. Siegel: Towards a Usable Theory of Chernoff Bounds for Heterogeneous and Partially Dependent Random Variables. TR1995-685, Courant Inst., New York Univ., 1995

[36] Y. Tao, C. Faloutsos, D. Papadias: The Power-Method: A Comprehensive Estimation Technique for Multi-Dimensional Queries. CIKM 2003

[37] A. Theobald, G. Weikum: The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. EDBT 2002

[38] C.T. Yu et al.: Database selection for processing k nearest neighbors queries in distributed environments. JCDL 2001