Anytime Measures for Top-k Algorithms

Benjamin Arai
Univ. of California, Riverside
barai@cs.ucr.edu

Gautam Das
Univ. of Texas, Arlington
gdas@cse.uta.edu

Dimitrios Gunopulos
Univ. of California, Riverside
dg@cs.ucr.edu

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

ABSTRACT

Top-k queries on large multi-attribute data sets are fundamental operations in information retrieval and ranking applications. In this paper, we initiate research on the anytime behavior of top-k algorithms. In particular, given specific top-k algorithms (TA and TA-Sorted) we are interested in studying their progress toward identification of the correct result at any point during the algorithms' execution. We adopt a probabilistic approach where we seek to report at any point of operation of the algorithm the confidence that the top-k result has been identified. Such a functionality can be a valuable asset when one is interested in reducing the runtime cost of top-k computations. We present a thorough experimental evaluation to validate our techniques using both synthetic and real data sets.

1. INTRODUCTION

Top-k queries on large multi-attribute databases are commonplace. Such queries report the k highest ranking results based on similarity scores of attribute values and specific score aggregation functions. Such queries are very frequent in a multitude of applications including (a) multimedia similarity search (on images, audio, etc.), (b) preference queries expressed on attributes of assorted data types, (c) Internet searches on scores based on word occurrence statistics and diverse combining functions, and (d) sensor network applications over streams of sensor measurements.

Several algorithms have been introduced in literature to efficiently perform top-k computations. Among the most successful is the TA algorithm discovered independently by Nepal et. al., [21], Guntzer et. al., [12] and Fagin et. al., [23]. In this algorithm each value of an attribute can be accessed independently via an index in descending order of its score. Such a score is computed with a specific query condition. Numerous algorithms for performing top-k computations have been proposed [10, 8, 7, 2, 19, 16, 1, 15, 20] depending on the model of data access, stopping conditions, etc. The majority of such computations however can be exhaustive. The algorithms come to a stop only when there is absolute certainty that the correct top-k result has been identified.

An *anytime* algorithm is an algorithm whose quality of results improves gradually as computation time increases [13]. Although several types of such algorithms have been proposed, *interruptible* anytime algorithms are highly popular and useful. An interruptible anytime algorithm is an algorithm whose runtime is not determined in advance but at any time during execution can be interrupted and return a result. Moreover, interruptible algorithms have an associated *performance profile* which returns result quality (for suitably defined notions of quality) as a function of time (relative to execution) for a problem instance. Such algorithms are valuable since at any point during the execution a user can obtain feedback regarding the result quality at that point. If one is satisfied with the current feedback one may bring the algorithm to a halt. Thus, such algorithms provide a graceful trade-off between result quality and response time.

In this paper we initiate a study of *anytime top-k* algorithms. We study the behavior of common top-k algorithms at any point of their execution and we reason about top-k result quality. Notice that this notion of anytime top-k computation is significantly different from the notion of approximate top-k algorithms previously introduced in the literature [6, 3]. Such models aim to relax the control parameters of the computation (e.g., distance) which are difficult to translate into guarantees perceived by a user. The actual behavior of such models remains largely empirical. In contrast we wish to monitor a top-k algorithm at any point in its execution and reason about result quality. For large data collections such an approach can be significantly beneficial as one may decide to terminate the computation early if one is satisfied with the current quality of the results. In particular we make the following contributions:

- We initiate the study of anytime top-k computations. We present a framework, within which at any point in query execution for suitable top-k algorithms, we can compute probabilistic estimates of several measures of top-k result quality. Such measures include *confidence of having the correct top-k result, precision of the results assessed with respect to the correct top-k results, rank distance between the current top-k result and the exact result, as well as the difference between the scores of the current top-k result and the exact result.*
- We investigate the monotonic properties of these anytime measures for various top-k algorithms such as TA and TA-Sorted. We show that such measures are monotone for TA, but for a single instance of a top-k computation of TA-Sorted, these measures can be non-monotonic, though in *expectation* such measures are monotonic.
- We present algorithmic enhancements to TA and TA-Sorted by which they can provide such anytime guarantees with small runtime overheads during the course of their execution over large data collections.

VLDB '07, September 23-28, 2007, Vienna, Austria.

2. RELATED WORK

The threshold algorithm (TA) constitutes the state of the art for top-k computations [21, 12, 23]. Several variants of the basic TA ideas have been considered in various contexts [15, 20, 16]. [1] deals with top-k problems on web accessible data sources with limited sorted access. Nearest neighbor type of approaches have been considered in this context as well [5, 4, 26, 14]. It is assumed that sorted lists of the data items by each attribute are available, and TA scans these lists (performing *sorted accesses*) in an interleaved manner, and computes the items with top-k scores using monotone score combining functions. The algorithm has to immediately compute the complete score for each item encountered in these lists. In order to do so however, it conducts *random accesses* to all relevant lists and thus its overhead may be high depending on the application context. For the rest of this paper we refer to this algorithm as TA.

Several variants of this basic idea have been proposed. TA-Sorted [23, 12] can work in environments where random access is not available. It maintains worst and best scores for items based on partially computed total scores; the algorithm compares the worst case score of the k-ranked item with the best score of all candidates as a stopping condition. In this algorithm items are always accessed sequentially. Since expensive random access is avoided, in certain situations the performance may be much better than TA.

Optimization issues for TA algorithms have been considered as well [19, 1, 16]. The main thrust has been to reduce the number of random accesses when sources vary in several parameters, such as speed, selectivity etc. Several statistical aids have been deployed, such as histograms and probabilistic estimators for the number of random accesses.

Anytime algorithms have found numerous applications in AI and planning contexts [13, 25]. The quality of results of an anytime algorithm improves as the computation evolves. At a high level, anytime algorithms can be categorized as being either interruptible or contract. An interruptible algorithm does not have a set running time and can always be interrupted at any time during execution returning a result. The quality of the result can be determined via a performance profile. A contract algorithm has a time deadline as a parameter and no assumption about the results can be made before the deadline.

Theobald et. al., [17] presented an approach for probabilistic top-k query evaluation. This work is specifically targeted to the TA-Sorted algorithm. The basic idea is, for a newly seen item, to compute the probability with which it may belong to the top-k result. If that probability is below a user supplied threshold the item is discarded from further consideration. This way, possibly fewer items are considered during top-k query evaluation. Moreover, by carefully maintaining bounds for the scores of the most promising (as far as the top-k result is concerned) items that have been encountered the algorithm may probabilistically decide to terminate earlier than the regular TA-Sorted deterministic computation. Empirical evaluation presented in [17] demonstrated that the algorithm performs well in practice.

The work of [17] has some similarity to our work, however it is not an anytime algorithm. It applies only to the TA-Sorted algorithm, and offers guarantees only at the end of the execution, i.e., when the algorithm runs out of candidates. Further, since it focuses only on eliminating candidates that are partially seen but unlikely to be in the final top-k result, it is not directly applicable to the TA algorithm. Finally, the algorithm in [17] only gives a probabilistic guarantee that a discarded/unseen tuple is not in the top-k tuples, independently of the number of unseen tuples in the data set. Its result does not change if we have 10 vs. 1,000,000 unseen tuples. However, even if each of the unseen tuples has a very small probability of being in the top-k tuples, collectively there may be a large probability that we do not have the correct top-k tuples. For e.g. (assume $k = 1$), if each unseen tuple has only 1% chance of being the top-1 tuple, and we have 1,000,000 unseen tuples, the probability that any one of them is the true top-1 is $1 - 0.99^{1,000,000} \approx 1$.

In contrast, our work is more general in that we propose anytime enhancements to both TA and TA-Sorted. Moreover, our methods depend on a careful consideration of the number of unseen tuples, which is necessary to give correct probabilistic guarantees.

Recent work [18, 24] on probabilistic ranking of data, is orthogonal to the work presented here. The model assumed in these works is that of incomplete data and the probabilistic framework is based on possible worlds semantics. In contrast we assume complete information with or without noise and we are interested in assigning guarantees on early stopping of popular top-k algorithms.

3. FRAMEWORK

3.1 Anytime Measures

Our focus in this paper is to upgrade top-k algorithms so that they can exhibit *anytime behavior*. This means that at any point during the execution - i.e., before the algorithm has terminated - we wish to be able to (a) reveal the current top-k results calculated thus far, and (b) associate a "guarantee" with our current answers. For example, we may wish to be able to give *probabilistic guarantees*, such as: "With probability p, the current top-k tuples are likely to be the true top-k tuples". Providing such probabilistic guarantees is the most critical aspect of our approach, and much of the remainder of this paper is devoted to developing appropriate guarantee measures and efficient techniques by which such measures can be calculated. Our goal is to provide a mechanism to continuously recompute these guarantees as more data is seen.

- *Confidence:* The algorithms shall be able to determine the probability that the current top-k tuples are indeed the true top-k tuples.

- *Precision*: The algorithms shall be able to calculate a (probabilistic) lower bound on the *precision* of the current top-k tuples - i.e., this bound on the precision will hold with a given probability of p (typically, $p = 0.95$). The precision of the retrieved results is defined as r/k where r is the number of the current top-k tuples that belong to the true top-k tuples.

- *Rank Distance*: Likewise, the algorithms shall be able to compute a probabilistic upper bound on the *rank distance* of the current top-k tuples. The rank-distance is defined as follows. Let $CurRank(t)$ be the rank of a tuple t in the current top-k, and let $TrueRank(t)$ be its rank in the entire database when sorted by scores. Then

$$Rank\ Distance = \sum_{t \in CurTopk} |CurRank(t) - TrueRank(t)|$$

  Rank Distance is related to the *Spearman's Footrule* measure for comparing ranked lists [9].

- *Score Distance*: Finally, the algorithms shall be able to compute a probabilistic upper bound on the difference between the smallest score of the true top-k tuples relative to the smallest score of the current top-k tuples.

3.2 Knowledge of the Data Distribution

To be able to give probabilistic guarantees with our anytime answers, it is critical that we assume some knowledge of the data, such as the number of tuples N, as well as knowledge of the distributional properties of the data. Such knowledge can be obtained via popular parametric or non-parametric techniques (i.e., histograms). These data distribution models are assumed to be either available (e.g., histograms of the data have been pre-computed, to be used multiple times for different top-k queries), or can be computed on demand (e.g., for each top-k query, fresh histograms are computed). Our development of anytime top-k algorithms does not depend on the particular type of distributional knowledge assumed. For this reason, we employ a generic probabilistic model of the data which we assume is known to us. We choose to do so in order to keep the presentation of our techniques generic and independent of specific forms of data distribution models.

To be more specific, let our database D have N tuples over M attributes $A_1, \ldots, A_M$ and let $Dom_1, \ldots, Dom_M$ be the respective domains of the attributes. The probability distributional model of the data may either be specified (assuming attribute independence) as a product of known probability density functions $gPDF_i(x)$ associated with each ith attribute (e.g., M single-dimensional histograms), or as a joint distributional model over the space of all possible tuples $Dom_1 \times \ldots \times Dom_M$ (e.g., a multi-dimensional histogram). Our actual database D may be assumed to be a specific instance of N tuples drawn from this distribution.

4. ANYTIME TA ALGORITHM

4.1 Preliminaries

We begin with a short description of the Threshold Algorithm (TA): The algorithm proceeds in iterations, where in each iteration, the next items in each sorted list are retrieved in parallel. For each retrieved tuple-id, the entire tuple is retrieved using random access and its score is computed. The algorithm maintains a bounded buffer of size k in which the current top-k tuples (i.e., among those seen) are maintained. The algorithm terminates when a *stopping condition* is reached, i.e., when the minimum score in the top-k buffer (henceforth referred to as $kMinScore$ is larger than $Score(h)$, where $h = [h_1, \cdots, h_M]$ is a "hypothetical" tuple such that each h_i is the last attribute value read along the sorted order for A_i.

Consider a snapshot of TA after d iterations for a specific database D. Let $Seen_d$ be the "prefix" of the database that has been seen by this algorithm after these d iterations. To be able to estimate the anytime measures, the algorithm will have to make some distributional assumptions about the remaining portion of the database that has not yet been seen. Intuitively, the algorithm determines the pdf of the remainder of the database by *conditioning* the data distributional model (discussed in Section 3.2) with the prefix already seen, and then computes estimates of each of the anytime measures based on this conditional pdf. As an example, assume that the data distribution of D is defined using the distributions $gPDF_i$ along the ith attribute assuming independence among the attributes, and let $h_1, \ldots, h_M$ be the last values seen along each attribute respectively. Then the ith attribute of any unseen tuple t in the remainder of the database will be a random variable $t[i]$ distributed according $gPDF_i$ *conditioned by* $t[i] \leq h_i$.

Let $PDF(O|O \in \mathcal{O})$ represents the probability density associated with object O that belongs to a (possibly infinite) set $\mathcal{O}$. Thus if $\mathcal{D}$ refers to the space of all database tables with N tuples that can be generated by the probabilistic data model discussed in Section 3.2, then $PDF(D|D \in \mathcal{D})$ is the probability density associated with each specific database D.

Let $OneMore(Seen_d)$ refer to the space of all possible valid prefixes of databases that is defined by extending $Seen_d$ by one more iteration. Consider any specific extension of $Seen_d$ by one iteration, say $Seen_{d+1}$. We note that a pdf over this space of extensions, i.e. $PDF(Seen_{d+1} \mid Seen_{d+1} \in OneMore(Seen_d))$, can be naturally defined. To carry $OneMore(Seen_d)$ even further, let $\mathcal{D}(Seen_d)$ refer to the space of all possible valid complete databases that can be defined by extending $Seen_d$ into complete databases, i.e., after $N - d$ iterations. The pdf of these databases, $PDF(D|D \in \mathcal{D}(Seen_d))$, can be naturally defined.

Let $Score(t)$ be the score of a tuple t, defined as a linear additive function on the the individual attribute values in typical top-k algorithms, such as $Score(t) = w_1 t[1] + \ldots w_M t[M]$ where the weights are positive constants. Let the $kMinScore(Seen_d)$ refers to the kth largest score of all tuples in $Seen_d$. We can make the following observation:

OBSERVATION 1. *The minimum score of the current top-k tuples increases monotonically as the algorithm progresses on any database.*

$$kMinScore(Seen_d) \leq kMinScore(Seen_{d+1})$$

Let $kthScore(D)$ refer to kth largest true score of all tuples in a specific database D. For Anytime TA let $Confidence(Seen_d)$ be defined as the probability that

$$kMinScore(Seen_d) = kthScore(D)$$

where D is a random valid extension of $Seen_d$ into a complete database drawn from $PDF(D|D \in \mathcal{D}(Seen_d))$.

THEOREM 1. *For all database instances it holds that*

$$Confidence(Seen_d) \leq Confidence(Seen_{d+1})$$

Proof: Since the $kMinScore(Seen_d)$ is increasing in each iteration, the probability of the $kMinScore(Seen_d)$ being equal to the $kMinScore(D)$ is also always increasing. □

4.2 The Algorithm

The anytime version of TA is shown in Algorithm 1. The algorithm proceeds like the standard TA, selecting attributes in a round-robin fashion, and at each step processes the next value in the sorted list of the selected attribute. In addition, it also maintains the information necessary to compute probabilistic guarantees[1].

For each round of the algorithm a new value $< t, t[i] >$ is read along the list L_i corresponding to the i-th attribute, i.e., the i-th attribute value of tuple t. When this item is read, the algorithm has to (a) resolve $Score(t)$ (which is the sum of the attributes of t and is done by probing the lists using random access), (b) update the pdf of the i-th attribute ($gPDF(i)$) so that it reflects the distribution of the remaining values of that attribute, and (c) update the top-k buffer with the k tuples with the highest scores. At the end of each round the statistics are updated and the confidence is computed.

[1]Note that unlike the standard TA algorithm our algorithm does not have a termination condition, since the objective is to produce anytime probabilistic guarantees. Our algorithm can be easily modified to terminate, for example when the probabilistic guarantees cross a user defined threshold.

Algorithm 1 Anytime TA

```
1: topk = {dummy_1, ..., dummy_k}, Score(dummy_i) = 0
2: kMinScore = 0 // smallest score in topk buffer
3: for d = 1 to N do
4:    for all lists L_i (1 ≤ i ≤ M) in parallel do
5:       Let <tuple-id t, t[i]> be the d-th item in L_i
6:       // Compute Score(t) using random access
7:       Score(t) = 0
8:       for j = 1 to M do
9:          Score(t) += w_j t[j]
10:      end for
11:      //Update PDFs to model the remaining values
12:      Update-gPDF(gPDF_i, t[i])
13:      //Update topk buffer
14:      if Score(t) > kMinScore then
15:         if t ∉ topk then
16:            Let u be the tuple with the smallest score in topk
17:            topk = topk − {u}
18:            topk = topk ∪ {t}
19:         end if
20:         kMinScore = min{Score(v) | v ∈ topk}
21:      end if
22:      // Compute confidence
23:      Confidence = ComputeConfidence()
24:   end for
25: end for
```

4.3 Computing Anytime TA Measures

In this subsection we discuss details of how the various anytime measures are computed in each iteration of the algorithm. For an unseen tuple t, its score may be viewed as a random variable. Let $scorePDF_t(x)$ be the pdf of the score of t. In order to compute the anytime measures, we need to compute the pdf of the score of any unseen tuple, and the pdf of the *maximum score* of all the unseen tuples. If we assume attribute independence, then the score of an unseen tuple is the sum of M random variables. To compute the pdf of this sum we compute the *convolution* of the $gPDF_i$. If joint-distributions are known we can also proceed to estimate the pdf of the score. We show below how this score can be estimated by convolution of pdfs of M independent attributes.

DEFINITION 1. **Convolution of two distributions:** *Assume that $f(x)$, $g(x)$ are the probability density functions (pdfs) of the two independent random variables X, Y respectively. The pdf of the random variable $X + Y$ (the sum of the two random variables) is the convolution of the two pdfs:*

$$*(\{f, g\})(x) = \int_0^x f(z)g(x - z)dz$$

This definition can be easily extended to the sum of more than two random variables. We also give another definition that allow us to estimate other aggregates, such as $\max and \min$ of random variables.

DEFINITION 2. **Max-convolution of two distributions:** *Assume that $f(x), g(x)$ are the pdfs of two random variables X, Y respectively. The pdf of the random variable $\max(X, Y)$ (the maximum of the two values) is the max-convolution of the two pdfs:*

$$*_{\max}(\{f, g\})(x) = f(x) \int_0^x g(z)dz + g(x) \int_0^x f(z)dz$$

Figure 1 shows the result of max-convolutions over two given distributions. The max-convolution definition can be easily extended to more than two random variables.

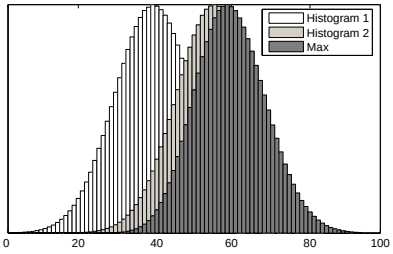


Figure 1: An example of the result of the max-convolution of two distributions.

4.3.1 Computing Confidence

Let $Seen$ ($Unseen$) refer to the set of tuples that have been seen (unseen) by the algorithm thus far. Clearly $|Unseen| = N - |Seen|$. To execute the function call $ComputeConfidence()$, we have to estimate $Prob(kMinScore > MaxUnseen)$, where $kMinScore$ is the minimum score in the top-k buffer, while the random variable $MaxUnseen$ describes the maximum score of all the unseen tuples. The pdf of $MaxUnseen$ can be computed by first computing the pdf of the score of one $Unseen$ tuple. This involves the convolution of the pdfs of the attribute values: $OneUnseenPDF = *\{gPDF_i | 1 \leq i \leq M\}$. Then the pdf of $MaxUnseen$ (i.e., $MaxUnseenPDF$) can be computed by computing the max-convolution over the multi-set containing $|Unseen|$ copies of $OneUnseenPDF$:

$$MaxUnseenPDF = *_{\max}(\{OneUnseenPDF, \ldots, OneUnseenPDF\})$$

As we shall later show in Lemma 4.3 the max-convolution of identical pdfs can be efficiently computed in constant time. Once we have computed $MaxUnseenPDF$, we can compute

$$Confidence = Prob(kMinScore > MaxUnseen)$$

4.3.2 Computing Other Anytime Measures

In this subsection we outline how, in addition to Confidence, the anytime measures of Precision, Rank Distance, and Score Distance can be computed.

In the case of Precision, we wish to determine (with a given probability p, say 95%) the fraction of the current top-k tuples that will belong to the true top-k tuples of the database. Let the worst case scores of the current $topk$ tuples be $s_1, s_2, \ldots, s_k (= kMinScore)$. Let $Prob_i$ be the probability that s_i is greater than $MaxUnseen$. These $Prob_i$'s can be computed using the same techniques used for computing Confidence above, except that we have to execute it for each s_i rather that just for $kMinScore$. Let i be the largest integer such that $Prob_i \geq p$. The algorithm outputs i/k as Precision. Note that this is a conservative bound on Precision because we only consider *prefixes* of the current top-k to be overlapping with the true top-k, and not any subset.

In order to compute Score Distance, our task is to find a "high probability" upper bound on the smallest score of the true $topk$ tuples. Thus, we wish to find the smallest positive number δ such that $Prob(kMinScore + \delta > MaxUnseen) > p$ where p is a given probability, such as 95%. Once we know the pdf of $MaxUnseen$, the answer to this question is straightforward.

Computing Rank Distance is more involved. The main task is to determine, for each tuple t_i in the current $topk$ tuples, a high probability upper bound for its true rank in the database (once we have

these estimates, we can compute a high probability upper bound for the Rank Distance). To determine an upper bound on the true rank of t_i, we need to compute how many tuples from $Unseen$ have larger scores than t_i with high probability. Further details are omitted from this version of the paper.

4.4 Approximating PDFs Using Histograms

We presented our techniques thus far using a generic probabilistic model of data. In this section we describe the practical realization of our methodologies using a widely adopted model for approximating data distributions (i.e., pdfs), namely histograms. For simplicity of exposition, we adopt equi-width histograms for our discussion, however the description is applicable to any histogram technique. We note that histograms can approximate arbitrary functions and thus our use of histograms does not place any restrictions or require any assumptions about the underlying distributions that are being approximated.

The following lemmas detail the running time of the basic operations of the algorithm.

LEMMA 4.1. *The convolution of two pdfs that are represented by two b bucket histograms can be computed in $O(b^2)$ time.*

Proof: Consider two random variables, A, B in the domain $[0, 1]$ with pdfs $f_A(x), f_B(x)$ respectively. Assume that the two pdfs are approximated by two histograms with b buckets, H_A and H_B. Assume that the bucket boundaries are the same: $H_A = [0 = A_1, \ldots, A_b = 1]$; if not we can create two equivalent histograms with $2b$ buckets and the same bucket boundaries. Consider the Cartesian product of the two histograms $C_{A,B} = H_A \times H_B$ where $C_{A,B}[i, j] = H_A[i]H_B[j]$ ($H_A[i]$ is the relative count associated with bucket i.) We can approximate the pdf of $A + B$ with a histogram with $2b$ buckets and boundaries $g_0 = 0, g_1 = A_1, \ldots, g_b = 1, g_{b+1} = 1 + A_1, \ldots, g_{2b} = 2$. To compute the histogram we have to compute the probability $Prob(g_k < A + B \leq g_{k+1})$ for the buckets of the new histogram, which may be derived as $\sum_{A_l+B_m=g_{k+1}} C_{A,B}[l, m]$ This histogram can subsequently be approximated by a b bucket histogram by merging neighboring pairs of buckets. This procedure gives an $O(b^2)$ algorithm for computing the convolution of the two pdfs. □

As a corollary, for n histograms, we can perform the convolutions in sequence, with a final running time of $O(nb^2)$.

LEMMA 4.2. *The max-convolution of two pdfs that are represented by two b bucket histograms can be computed in $O(b)$ time.*

Proof: The trick here is to avoid the Cartesian product. As before, consider two random variables A, B with pdfs approximated by two histograms H_A and H_B each with b buckets and the same bucket boundaries. We approximate the pdf of $\max(A, B)$ with a histogram with the same bucket boundaries. Then $Prob(A_k \leq \max(A, B) < A_{k+1})$ is equal to $H_A[k + 1] \cdot (\sum_{i \leq k} H_B[i]) + H_B[k + 1] \cdot (\sum_{i \leq k} H_A[i])$.

If we first compute the *cumulative distributions* of H_A and H_B, it is easy to see the the above probability can be computed in constant time. Since the cumulative distributions can be computed in $O(b)$ time, the overall time for the max-convolution is $O(b)$. □

As a corollary, we can compute the max-convolution of n histograms in $O(nb)$ time. Even more interestingly, as the following lemma shows, the max-convolution of n *identical* histograms can be computed in $O(b)$ time.

LEMMA 4.3. *The max-convolution of n identical PDFs, represented by a b bucket histogram, can be computed in $O(b)$ time.*

Proof: The proof is similar to that of Lemma 4.2, except that once the cumulative pdf of H_A has been pre-computed, each probability term $Prob(A_k \leq \max(A, A, \ldots, A) < A_{k+1})$ reduces to $n \cdot H_A[k+1] \cdot (\sum_{i \leq k} H_A[i])^{n-1}$, which can be computed in constant time. □

4.5 An Example

A_1 id,val	A_2 id,val
t_4:0.9	t_5:0.8
t_2:0.8	t_4:0.7
t_3:0.4	t_2:0.6
t_1:0.3	t_1:0.3
t_5:0.2	t_3:0.2

Table 1: Sorted lists of a sample table with two columns A_1 and A_2, tuples t_1...t_5, and values for each attribute ranging from 0 to 1.

Table 1 shows the sorted lists for a dataset with 2 attributes and 5 tuples. We have a query for the top-k tuples where k is equal to 2, and the score for a given tuple t is computed as a linear additive function of the individual attributes.

Throughout the example, assume we use equi-width histograms with at most 2 buckets. At the start, the buckets of the histogram for A_1 have counts $(3, 2)$, while the buckets of the histogram for A_2 have counts $(2, 3)$. Note that each histogram can represent the corresponding $gPDF_i$ by normalizing to relative counts.

Assume a snapshot of the algorithm where the first items of each list has been read, and t_4 and t_5 have been fully resolved and loaded into the top-k buffer. Thus t_4 and t_5 belong to the *Seen* group. Clearly $kMinScore = Score(t_5)$ is the lowest score in the top-k buffer.

The remaining tuples t_1, t_2, and t_3 are in the $Unseen$ group. We need to estimate *OneUnseenPDF*, the pdf of the score of any $Unseen$ tuple using the $gPDF_i$s for attributes A_1 and A_2. We have to first update the $gPDF_i$s to model the remaining values for each attribute. Consequently the buckets of the histogram for A_1 will now have counts $(3, 1)$, while the buckets of the histogram for A_2 will have counts $(2, 2)$. We then normalize each $gPDF_i$ by dividing by the sum for each $gPDF_i$ to get $(3/4, 1/4)$ and $(1/2, 1/2)$ respectively. We then compute *OneUnseenPDF* by taking the convolution of $gPDF_1$ and $gPDF_2$, resulting in counts $(3/8, 5/8)$. Next we need to compute *MaxUnseenPDF*, the pdf of the max score of all unseen tuples. As shown in Lemma 4.3, we do this by taking the pdf of a single unseen tuple *OneUnseenPDF* and raising it to the power of the total number of unseen tuples, which in this case is 3.

We can then compute the confidence of the current top-k buffer by comparing *MaxUnseenPDF* with the current top-k $kMinScore$ as described in Section 4.3.1.

4.6 Considering Multidimensional Distributions

The pdf of the score of a tuple for a given query depends on the joint distribution of the attributes. Many commercial systems make the attribute value independence assumption, and keep statistics only for individual attributes. In our setting as described above, the independence assumption is similarly assumed when we compute the pdf of the score of a tuple by taking convolutions of the histograms of the different attributes. We stress here that the computation of the score is the only place in our framework this as-

sumption has been made. All other computations (including max-convolutions) that take place in the computation of our any-time measures do not make any assumptions on the distributions. Although the independence assumption is commonly applied and is well validated in practice for in a wide variety of applications, it may produce inaccurate results in some cases (whether the confidence curve using one-dimensional histogram is higher or lower than the confidence curve using two-dimensional histograms depends on whether the independence-based approach is overly optimistic or pessimistic). For such cases, joint distribution models involving multiple attributes may be necessary.

Joint distributions can easily be applied in our framework. Suppose for some tuple t we have three attributes A, B and C which are unknown. Earlier we showed that we can compute the convolution of H_A, H_B, and H_C, but with *multidimensional histograms* we can now compute the convolution of the score pdf of $A + B$ and H_C, where the score pdf of $A + B$ may be directly computed from $H_{A,B}$, the two-dimensional histogram representing the join distribution of attributes A and B.

As in the case of one-dimensional histograms, multidimensional histograms are computed as a pre-processing step. Methods for computing multidimensional histograms have been throughly researched [11] [22] involving sampling and other efficient approximation techniques. In the evaluation section of this work we consider two-dimensional histograms. Since the number of possible two-dimensional histograms is quadratic in the number of attributes, we have to decide which pairs to take. We use the following simple heuristic: starting with the set of attributes, we find the most correlated pair of attributes, compute a two-dimensional histogram on these attributes, remove these two attributes, and continue with the remaining set. This approach produces a linear number of histograms, and, since there is no overlap of attributes between different histograms, greatly simplifies the selection of the histograms that have to be used to compute the convolution of a set of attributes.

Recall that for one dimensional histograms (histograms covering a single attribute) every time a new item from the sorted list is read, the corresponding bucket has to be decreased by one. In the case of multidimensional histograms we similarly decrement the histograms as new items are read. Suppose we have a database with two attributes A and B, two one-dimensional equi-width histograms H_A, H_B, as well as one 10×10 equi-width 2-dimensional histogram $H_{A,B}$. If the first tuple that is completely resolved has the values $(0.3, 0.9)$, we decrement the buckets of the histograms as follows: $H_A[3]$ would be decremented, $H_B[9]$ would be decremented, and $H_{A,B}[3, 9]$ would be decremented. This same technique follows through for higher dimensional histograms and can be performed incrementally.

5. ANYTIME TA-SORTED ALGORITHM

In this section we describe how the TA-Sorted algorithm can be extended to compute online probabilistic guarantees. In addition to $Seen$ and $Unseen$ tuples, TA-Sorted also maintains tuples in which only some of the attributes have been seen. This is a consequence of the inability of TA-Sorted to perform random access operations. Consequently, during the operation of TA-Sorted, we need to keep a set of tuples called $Partials$ that are not in the top-k, yet cannot be eliminated because we know only a lower-bound of their true score. The TA-Sorted algorithm must estimate the pdf of the maximum scores of the $Partials$ before giving any probabilistic guarantee on the confidence.

Like TA, the TA-Sorted algorithm as shown in Algorithm 2 selects attributes in a round-robin fashion, at each step processing

Algorithm 2 Anytime TA-Sorted

```
1: topk = {dummy_1, ..., dummy_k}, MinScore(dummy_i) = 0
2: Partials = {} // Partially seen tuples not currently in topk
3: kMinScore = 0 // smallest score in topk buffer
4: Assume for all tuples t, obs(t) = {}
5: for d = 1 to N do
6:   for all sorted lists L_i (1 ≤ i ≤ M) in parallel do
7:     Let <tuple-id t, t[i]> be the d-th item in L_i
8:     obs(t) = obs(t) ∪ {i}
9:     MinScore(t) = 0
10:    for j ∈ obs(t) do
11:      MinScore(t) += w_j t[j]
12:    end for
13:    //Update PDFs by conditioning with remaining values
14:    Update-gPDF(gPDF_i,t[i])
15:    //Update topk buffer
16:    if MinScore(t) > kMinScore then
17:      if t ∉ topk then
18:        Let u be tuple with smallest worst case score in topk
19:        Remove u from topk
20:        if |obs(u)| < M then
21:          Partials = Partial ∪ {u}
22:        end if
23:        topk = topk ∪ {t}
24:      end if
25:      kMinScore = min{MinScore(v)| v ∈ topk}
26:    end if
27:    if |obs(t)| < M and t ∉ topk then
28:      Partials = Partials ∪ {t}
29:    else
30:      Partials = Partials − {t}
31:    end if
32:    // Compute confidence
33:    Confidence = ComputeConfidence()
34:  end for
35: end for
```

the next (sorted by decreasing magnitude) value of the selected attribute. The differentiating factor between Anytime TA and Anytime TA-Sorted is the inclusion of $Partials$. Let $Partials$ be the set of tuples that are partially seen (some but not all of the attributes for a given tuple have been resolved), but are not in the top-k buffer.

Let $< t, t[i] >$ be the next item read by the algorithm along the sorted list L_i corresponding to the i-th attribute, i.e., the i-th attribute value of tuple t. When this item is read, the algorithm has to (a) update $MinScore(t)$ (which is the sum of the attributes that have seen for t) (b) update the pdf of the attribute i ($gPDF_i$), and (c) update the top-k buffer with the k tuples with the highest lower-bound scores. After reading $t[i]$, t will either be fully resolved (that is, all attributes of t have been seen and its final score found) and put in the $Seen$ group, or partially resolved and placed in the $Partials$ group.

5.1 Monotonicity for Anytime TA-Sorted Measures in Expectation

Let $kthScore(D)$ refer to the kth largest score of all tuples in a specific database D. The $Confidence(Seen_d)$ for Anytime TA-Sorted may be defined as the probability that

$$kMinScore(Seen_d) > (k+1)thScore(D)$$

where D is a random valid extension of $Seen_d$ into a complete

database drawn from $PDF(D|Din\mathcal{D}(Seen_d))$. Because of the use of lower-bound scores, this definition of confidence is actually even more conservative than the earlier definition of confidence in Section 3.1.

Example: There exist a database instance where

$$Confidence(Seen_d) > Confidence(Seen_{d+1})$$

Assume a database with two columns A_1 and A_2, each with domain $[0.0, 1.0]$ and a uniform distribution model. Let the score function be $Score(t) = t[1] + t[2]$. Let the database have four tuples with tuple-ids $t1, \ldots, t4$, and assume that the task is to return the top-2 tuples.

In the first iteration, assume we encounter $t1 = [0.9, ?]$ and $t2 = [?, 0.9]$, along each of the sorted lists (a ? implies that the corresponding attribute value is unresolved). After this iteration, the top-2 buffer is loaded with $t1$ and $t2$, each with a worst case score of 0.9. Since we have not seen the other two tuples, we assume that each is distributed uniformly in $[0.0, 0.9] \times [0.0, 0.9]$, and hence the probability that the current worst case score of 0.9 is larger than the scores of both these unseen tuples is $(1/2) * (1/2) = 1/4$.

Suppose in the next iteration the algorithm encounters $t3 = [0.8, ?]$ and $t4 = [?, 0.8]$. After this iteration, the top-2 buffer remains unchanged. However, the unresolved attribute of $t3$ has a probability of $7/8$ of having a value in the range $[0.1, 0.8]$, which would enable $t3$ to have larger score than the current worst case score. A similar argument can be made for $t4$. Thus, the probability that the current worst case score of 0.9 is larger than the scores of both these (now partially seen) tuples decreases to $(1/8) * (1/8) = 1/64$. □

Similar examples can be constructed to show that the other anytime measures are non-monotonic for certain database instances. These arguments bring to light a subtle issue. The uncertain (probabilistic) nature of anytime measures should of course be obvious to the reader - i.e., that at any point during execution, we cannot be completely certain that we have discovered the true top-k tuples, and therefore can only make probabilistic guarantees regarding our anytime measures. However, what the example shows is that as the iterations progress, we may have to revise, and sometimes *even reduce*, our probabilistic guarantees. We note that a similar argument will not suffice in the case of TA, because in that algorithm a tuple is never in a partially resolved state - it is either completely seen or completely unseen.

However, although the anytime measures for TA-Sorted are not monotonic for certain database instances, we can nevertheless show that the measures are monotonic *in expectation* over all database instances. We describe the result for the confidence measure. Similar results for the other anytime measures are straightforward and omitted due to lack of space.

Let $E[Confidence(Seen_{d+1})]$ be defined as the expected value of $Confidence(Seen_{d+1})$, where $Seen_{d+1}$ is randomly drawn from $PDF(Seen_{d+1} \mid Seen_{d+1} \in OneMore(Seen_d))$.

THEOREM 2 (EXPECTED MONOTONICITY THEOREM).

$$Confidence(Seen_d) \leq E[Confidence(Seen_{d+1})]$$

Proof: From the definition of confidence, we know that

$$Confidence(Seen_d) = \sum_{D \in \mathcal{D}(Seen_d)} (kMinScore(Seen_d) = kthScore(D)) \cdot Prob(D|D \in \mathcal{D}(Seen_d))$$

Partitioning all valid database extensions D as follows, we get

$$Confidence(Seen_d) = \sum_{Seen_{d+1} \in OneMore(Seen_d)} \Big(\sum_{D \in \mathcal{D}(Seen_{d+1})} (kMinScore(Seen_d) = kthScore(D)) \cdot Prob(D|D \in \mathcal{D}(Seen_{d+1})) \Big) \cdot Prob(Seen_{d+1}|Seen_{d+1} \in OneMore(Seen_d))$$

From Theorem 1 we have

$$kMinScore(Seen_d) \leq kMinScore(Seen_{d+1})$$

for any extension $Seen_{d+1}$. Thus the above reduces to:

$$Confidence(Seen_d) \leq \sum_{Seen_{d+1} \in OneMore(Seen_d)} \Big(\sum_{D \in \mathcal{D}(Seen_{d+1})} (kMinScore(Seen_{d+1}) = kthScore(D)) \cdot Prob(D|D \in \mathcal{D}(Seen_{d+1})) \Big) \cdot Prob(Seen_{d+1}|Seen_{d+1} \in OneMore(Seen_d))$$

Thus,

$$Confidence(Seen_d) \leq \sum_{Seen_{d+1} \in OneMore(Seen_d)} Confidence(Seen_{d+1}) \cdot Prob(Seen_{d+1}|Seen_{d+1} \in OneMore(Seen_d))$$

Thus, $Confidence(Seen_d) \leq E[Confidence(Seen_{d+1})]$. □

5.2 Computing Anytime TA-Sorted Measures

In this subsection we discuss how the anytime measures are computed in each iteration of the TA-Sorted algorithm. Our focus is on the Confidence measure; the details of the computation of other anytime measures are omitted due to lack of space.

5.2.1 Computing Confidence

At any instance during the execution of the algorithm, consider the set of tuples $Others = Partials \cup Unseen$. Let $MaxOthers$ be the random variable that denotes the maximum score of all tuples in $Others$. To execute the function $ComputeConfidence()$, we have to estimate $Prob(kMinScore > MaxOthers)$. To compute this probability, we need to first compute the pdf of the random variable $MaxOthers$. This can be accomplished if we compute the pdfs of two random variables, $MaxPartials$ and $MaxUnseen$ and then compute the pdf of the maximum of these two random variables.

The pdf of $MaxUnseen$ (i.e., $MaxUnseenPDF$) can be computed as was done in TA, i.e., by raising $OneUnseenPDF$ to the power of the total number of unseen tuples. To compute the pdf of $MaxPartials$, we first define $ScorePDF_t$, the distribution of the score of a partially seen tuple t. The definition is similar to the definition of the score pdf of an unseen tuple (i.e., $OneUnseenPDF$), except that the convolutions are taken only over the pdfs of the unresolved attributes of t, to which the aggregate of the resolved attribute values (i.e., $MinScore(t)$) is combined.

More formally, given a real number a, let $\delta_a(x)$ denote the "delta distribution" where all the probability mass is concentrated at a and is 0 elsewhere. Then

$$ScorePDF_t = *(\{\delta_{MinScore(t)}\} \cup \{gPDF_i | i \notin obs(t)\})$$

$MaxPartialsPDF$ may now be defined as:

$$MaxPartialsPDF = *_{\max}(\{scorePDF_t | t \in Partials\})$$

This operation is linear in the number of partially seen tuples, and so it can become slow for large data sets. In the following Section 5.2.2 we present an efficient implementation by clustering partially resolved tuples. Once we have computed $MaxUnseenPDF$ and $MaxPartialsPDF$, we can compute $MaxOthersPDF$ and use that to compute

$$Confidence = Prob(kMinScore > MaxOthers)$$

5.2.2 *Efficiently Computing* $MaxPartialsPDF$

The straightforward way to compute $MaxPartialsPDF$ is to compute the max-convolution of the score pdfs of the partially seen tuples. This operation is linear in the number of partially seen tuples, and so it may become slow for large data sets.

To improve the running time, we cluster the partially seen tuples. Consider a subset of the attributes, S, and let $Partials_S$ be the set of tuples that have exactly these S attributes resolved. That is, $Partials_S = \{t | obs(t) = S\}$. Since all the tuples in $Partials_S$ have the same attributes unresolved, we can speed up the computation of the max-convolution of their scores:

$*_{\max}(\{scorePDF_t | t \in Partials_S\}) =$
$*_{\max}(\{*(\{\delta_{MinScore(t)}\} \cup \{gPDF_i | i \notin S\}) | t \in Partials_S\})$

Then, let us consider the worst case scores (i.e., $MinScore(t)$) of the tuples t in $Partials_S$, and consider an equi-width B-bucket histogram H with these values (where B may be different from the b used to denote the number of buckets in the score/attribute histograms). Let $U(t)$ be the upper bound of the range of the histogram bucket of H that $MinScore(t)$ falls in. Let us replace the worst case score of each tuple with this upper bound of the corresponding histogram bucket. We have then,

$*_{\max}(\{scorePDF_t | t \in Partials_S\}) \leq$
$*_{\max}(\{*(\{\delta_{U(t)}\} \cup \{gPDF_i | i \notin S\}) | t \in Partials_S\})$

Thus, any two tuples in $Partials$ that have the same set of resolved attributes and whose worst case scores map to the same bucket have approximately identical score distributions. Since there are 2^M possible subsets of attributes, and we use a B-buckets histogram for each subset, we have essentially partitioned all tuples in $Partials$ into at most $2^M B$ clusters. Using Lemma 4.3 for each of these clusters we can compute an upper bound for the pdf of their maximum score. We can then compute the max-convolution of the resulting $2^M B$ histograms to finally compute $MaxPartialsPDF$.

To efficiently do this computation we have to maintain one counter for each of the $2^M B$ histogram buckets (which are in the beginning initialized at 0). Every time a new value is read in, one of the tuples has one more attribute resolved. If this is a new tuple, we increment the corresponding bucket and add this tuple to the $Partials$ set. If the tuple is already in $Partials$, one bucket will have its counter reduced by one. If the tuple is still not fully resolved, another bucket will have its counter increased by one.

Using Lemmas 4.1, 4.2 and 4.3, we can state the following lemma:

LEMMA 5.1. *An upper bound for $MaxPartialsPDF$ can be computed in $O(2^M B b^2)$ time.*

We note that the running time of this update is independent of N, the total number of tuples in the database.

6. EXPERIMENTAL EVALUATION

In this section we present an experimental evaluation of our framework. The implementation of our techniques is in C++ and our evaluations are performed on a dual AMD Opteron 280 processor system with 8GB of memory.

We have conducted series of experiments using synthetic and two real-world data sets varying the distribution and size. The data sets range in size from 4,990 to 1,000,000 rows, and four to ten attributes (we vary the number of attributes when we report on performance). Our experiments focus on the comparison of the accuracy of our estimated results with the expected performance of the TA and TA-Sorted algorithms. We also generate data with Zipfian distributions and conduct similar sets of experiments. Due to space constraints we do not include illustrations for comparing Zipfian distributed scores but we briefly discuss the highlights of the results.

6.1 Real World Data Sets

In our experiments we use two real-world data sets. Our first data set is atmospheric data collected from several independent sensor locations in Washington and Oregon by the Department of Atmospheric Science at the University of Washington. The second is the Internet Movie Database IMDB[2].

For the sensor data, 25 sensors independently obtained temperature readings on an hourly basis between June 2003 and June 2004, for a total of 208 days. For each sensor there is a total of 4,990 readings. Each of the readings taken from a sensor were combined with readings from other sensors which had taken a reading during the same time period. These readings were grouped to make individual rows based on their time-stamps. Sensor data such as the temperature data provided can specifically benefit from our algorithm due to the *anytime* behavior. For our experiments we use the readings from five to ten randomly selected sensors.

The IMDB database is composed of more than 860,000 titles and details about each. For the IMDB data set, we extracted a list totaling 863,049 titles. For each title, we queried the following attributes: budget, gross income, opening weekend gross income, and number of keywords describing the title.

We experimented with several different histogram sizes; we found that the accuracy did not improve much with histograms of more than 20 buckets for our real-world experiments.

6.2 Anytime Measures

Our experimental evaluation validates our measures on real-world and synthetic data sets. As a baseline we compare our approach against the actual confidence, TA, and TA-Sorted algorithms.

In the case when the distribution of scores is skewed, the confidence of the algorithm may stay relatively low for a large portion of the data set. This is due to a high density of values keeping the $kMinScore$ and $MaxOthers$ close for a larger portion of the running time (i.e., there is a low-sloping increase in the confidence, but eventually it reaches 100% confidence). In cases when the distribution of the data set contains a distinct cluster of K or more high scores (row-level correlation) the confidence quickly climbs. For the IMDB data set, there are few large values with the majority of the scores being clustered toward the lower end of the value range for each attribute. This is reasonable considering that there are only a few big budget movies and of these movies an even

[2]http://www.imdb.org

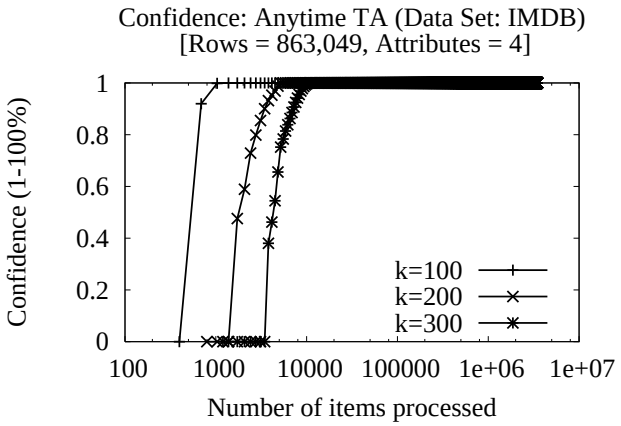


Figure 2: In this experiment we evaluate the confidence for varying *k* as the number of seen tuples is increased for the IMDB data set.

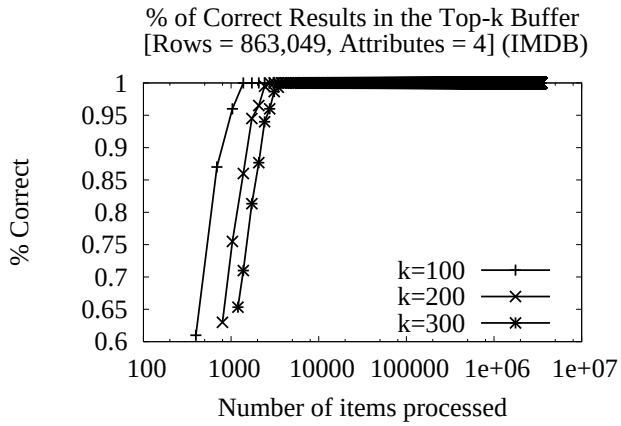
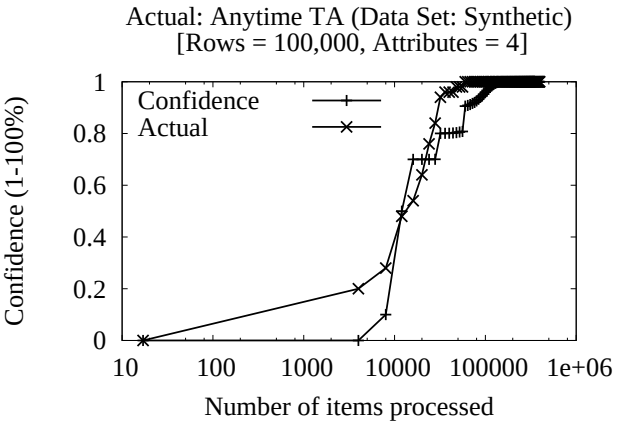


Figure 3: In this experiment we show the precision, defined as the percentage of the current top-k buffer that is actually in the top-k result for the IMDB dat set.



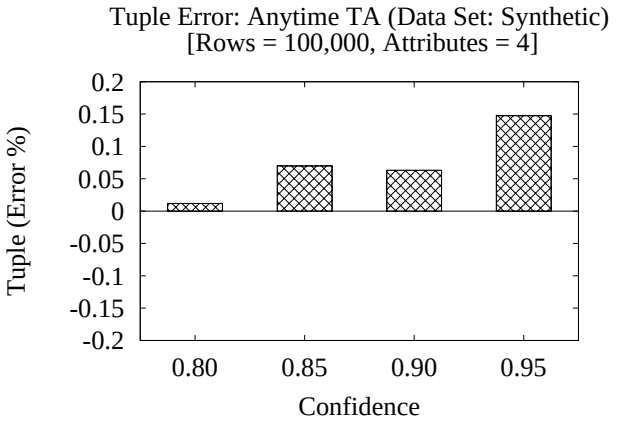


Figure 4: In this experiment we compare the actual and Anytime TA confidence. The two figures show the difference in the number of items read for various levels of confidence using the synthetic data set.

smaller subset that gross a large sum of money. This creates a data set with a small number of high score tuples. Similarly in the case of the sensor data there is row-level correlation around temperature spikes with the majority of the readings being located around the average temperature for each sensor. As shown in figures 2 and 9, both the IMDB and sensor data sets illustrate how correlation of attributes can quickly cause the Anytime TA algorithm to climb to 100% confidence, this can be accounted for by the fact that the correlation of data cause the the $kMinScore$ and $MaxOthers$ groups to quickly diverge.

Accuracy: Our results show good performance for both real-world and synthetic data sets. In figures 2 and 3 we show the confidence and percentage of correct results in the top-k buffer during the execution of the algorithm. These figures illustrate how our estimates coincide with the number of correct results in the top-k buffer.

Further, in Figure 4 we show that our estimates for the confidence accurately approximates the actual confidence. In order to compare the accuracy of our estimations, we computed the actual confidence by running the TA algorithm for 10 independent runs (we generated 10 randomly distributed synthetic data sets and ran the algorithm for each) building a vector for each run where each element of the vector contains one of two values (1="Top-k found", 0="Top-k not found yet"). We then computed the average over all runs (i.e., we built a new vector that represents the element-wise average of the vector set) creating a new vector of real values where each element of the vector represents the actual confidence for each respective run.

We evaluate the accuracy of readings by comparing the number of items read given a user-defined confidence using Anytime-TA with the number of items retrieved had the actual confidence (defined above) been known. We can estimate the accuracy of a reading by comparing the number of items read for Anytime TA and the actual confidence. In Figure 4 we shown the error percentage for confidence levels of 0.80 through 0.95. Our algorithm performs well for various levels of confidence. The results suggest that there is little correlation between the confidence level and the accuracy of our results. For the experiment presented in Figure 4, the number of items read by the Anytime TA algorithm never deviates more than 16% from the number of items read for the corresponding actual confidence.

6.3 Scalability & Performance

Efficiency: Our results show that sizable savings can be achieved in comparison to the TA and TA-Sorted algorithms. As a baseline we ran TA and TA-Sorted on the IMDB and sensor data sets. In each case we computed how many tuples were read before the TA

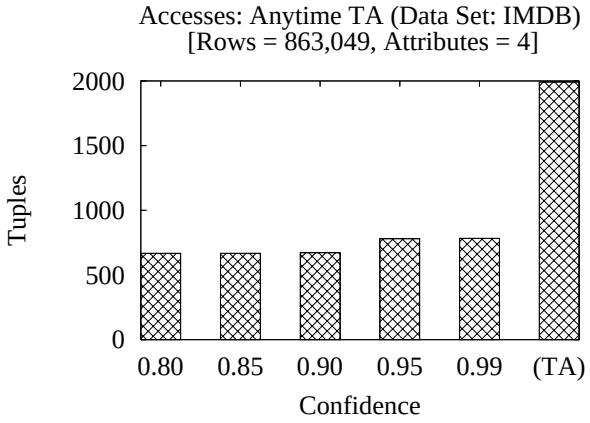


Figure 5: In this experiment we compare the number of tuples retrieved for Anytime TA with various levels of confidence using the IMDB data set where K=100.

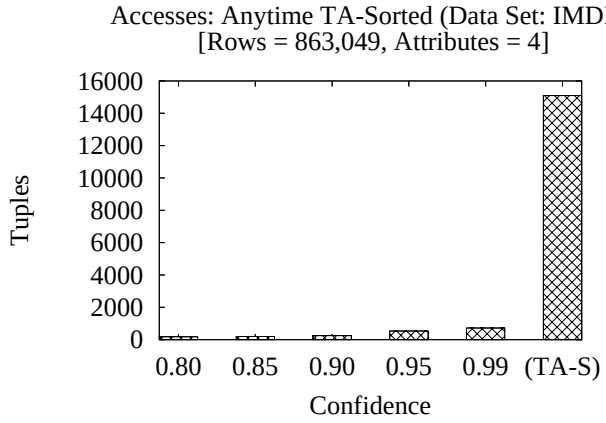


Figure 6: In this experiment we compare number of tuples retrieved for Anytime TA-Sorted with various levels of confidence using the IMDB data set where K=100. (TA-S) = TA-Sorted

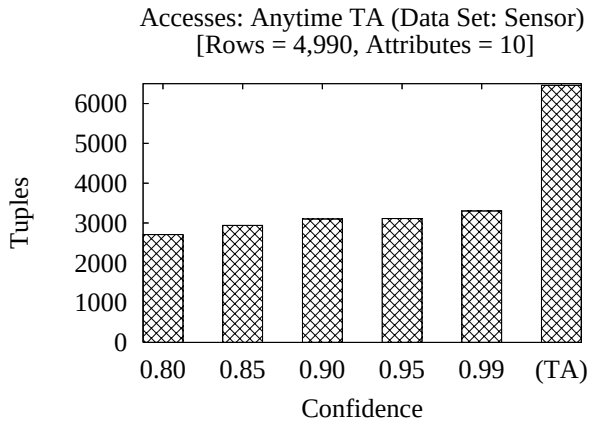


Figure 7: In this experiment we compare the number of tuples retrieved for Anytime TA with various levels of confidence using the sensor data set where K=300.

or TA-Sorted stopping condition was reached. We then compared these results with our algorithm. As shown in Figure 5 Anytime TA provides sizable savings over TA. We achieve a saving of over 70% (1,200 tuples) for a confidence level of 99% using the IMDB data set. Similarly, Anytime TA works well for high dimensional (sensor) data sets. As shown in Figure 7, we achieve savings of over 50% (3,000 tuples) for a confidence level of 99% using the sensor data set. Since TA-Sorted does not allow for random accesses, the number of tuples read is usually much greater than TA (allowing for greater savings). As shown in Figure 6 we compare the Anytime TA-Sorted algorithm with TA-Sorted. In this case, for TA-Sorted and a confidence level of 99% we achieve an even greater savings of over 95% (14,000 tuples).

Scalability: To evaluate the overhead of our approach we ran scalability experiments with a synthetic data set totaling 1,000,000 rows and 4 attributes. We used histograms of 5 to 25 buckets to describe attribute distributions. In this set of experiments we set $K = 1,000$, but similar results were obtained for different values.

Table 2 shows the runtime performance of the Anytime TA algorithm, as well as the overhead that the technique imposes over the TA algorithm. In the first column we report the running time of the TA algorithm. In the second column we report the running time of our implementation of the Anytime TA algorithm. This does not include the time that it takes to compute the anytime measures. In other words, this includes the time it takes to run TA and the time it takes to maintain the *gPDF*s for each round. Note that this time is dependent upon the users confidence bound. The third column shows the average time for computing the anytime measure (confidence, rank distance, and so on) every time this computation is invoked. The total running time of our algorithm is the sum of the time it takes to run Anytime TA (column 2) and the time it takes to compute the anytime measures (column 3) times the number of times the anytime computation is invoked.

The experimental results in Table 2 suggest that the overhead of our approach is relatively small for Anytime TA. There is little variation in runtime between the TA and Anytime TA algorithm (this is attributed to the fact that histograms are not utilized for computation until a reading is taken). Varying the histogram size between 5 and 25 buckets make little difference in effecting the runtime of the Anytime TA algorithm.

For the Anytime TA-Sorted algorithm as shown in Table 3 there is a sizable difference in the running time for TA-Sorted and Anytime TA-Sorted algorithms. This is attributed to the overhead incurred from the maintenance of the partially seen tuples. In other words, this includes the time it takes to run Anytime TA-Sorted, update the *gPDF*s and maintain partially seen clusters for each round as defined in Section 5.2.2. Varying the histogram size between 5 and 25 buckets make little difference in effecting the runtime of the Anytime TA-Sorted algorithm. Overall, the overhead for the partials remains a fixed cost over Anytime TA and increases when the size of the histograms increases, as expected.

Performance: We evaluate performance in terms of how many tuples we read, and how long it takes to run the algorithm using our implementation. We compare Anytime TA with TA. To evaluate our approach we ran experiments using a synthetic data set totaling 100,000 rows, 4 attributes, and a uniform distribution for each attribute; we use a histogram size of 20 to describe the distribution. In this set of experiments we set $K = 1000$, but similar results were obtained for different values. Proper selection of skip size (i.e. the number of tuples sampled between readings) can greatly affect the runtime and total number of tuples sampled. A large skip size ensures that the number of readings is minimal. If the skip size is too large then there is a coarsening of the confidence levels between readings, generally causing additional tuples to be read

TA	Anytime TA	Estimation Time Average Time Per Readings	Histogram Size
1.0908	1.1238	0.0001	5
	1.1598	0.0004	10
	1.2068	0.0009	15
	1.1778	0.0014	20
	1.2107	0.0020	25

Table 2: Run time performance for synthetic data set comparing *Anytime TA*, *TA* and time required to take an *Anytime TA* measure. Synthetic data set (1,000,000 tuples, 4 attributes, histograms size 20, random distribution). Time is reported in seconds.

TA-Sorted	Anytime TA-Sorted	Estimation Time Average Time Per Readings	Histogram Size
2.7696	20.8258	0.0001	5
	26.2369	0.0004	10
	32.3550	0.0007	15
	41.4386	0.0013	20
	51.6661	0.0019	25

Table 3: Run time performance for synthetic data set comparing *Anytime TA-Sorted*, *TA-Sorted* and time required to take an *Anytime* measure. Synthetic data set (1,000,000 tuples, 4 attributes, histograms size 20, random distribution). Time is reported in seconds.

from the database. On the other hand, if the skip size is small then fewer tuples may be sampled but the runtime will increase due to the inflation of reading overhead.

Table 4 offers a comparison of runtime performance for Anytime TA and TA for several confidence levels. We have omitted results for Anytime TA-Sorted due to space constraints. For each confidence level we report both the runtime and number of tuples sampled for each algorithm. The experimental results in Table 4 show that sizable gains can be achieved over TA for both runtime and the number of tuples read from the database. For our experiments we achieved a reduction of about 35,000 - 38,000 tuples read from the database and a 34% - 44% decrease in runtime over TA. Overall, we have found that the our approach works well in a variety of settings that can be further tuned using different histogram and skip sizes.

6.4 Multidimensional Histograms

We consider the effects of joint distributions (multidimensional histograms) by comparing the accuracy and performance of our algorithm using one- and two-dimensional histograms. Like the one-dimensional *gPDF*s we have used thus far, we assume that multi-dimensional histograms are provided as a pre-processing step. We want to compare the accuracy of our results using various levels of knowledge about the scores in the database. For the experiments using joint distributions we assume all combinations of two attribute joint distributions $gPDF_i$s are available as described in Section 4.6.

As shown in Figure 10 we compared the performance of one- and two-dimensional histograms for the IMDB data set. The inclusion of multidimensional histograms did not greatly effect the number of tuples read from the database. We experimented with confidence levels (85% and 95%) and in each case we compared

Confidence	TA Tuples Read	TA Time	Anytime TA Tuples Read	Anytime TA Time
0.75	54,082	0.0650	16,000	0.0359
0.85			17,000	0.0399
0.95			19,000	0.0429

Table 4: Run time performance comparing TA and $Anytime-TA$ for varying confidence levels. Synthetic data set (100,000 tuples, 4 attributes, histograms size 20, random distribution, skip size=1000). Time is reported in seconds.

the number of tuples retrieved for the Anytime TA algorithm using independent and joint distributions. In both cases the algorithm retrieved roughly the same number of tuples.

In addition, in figures 8 and 9 we show how the inclusion of multidimensional histograms can increase the accuracy of our algorithm. This is shown in the (1D,2D Histogram) results by the increase in slope for the confidence. This is expected since joint distributions offer more information pertaining to the scores of the unseen tuples. Therefore, at any given point during execution there is less uncertainty of the remaining unseen scores in the database.

Accuracy: As shown in figures 8 and 9 as the dimensionality of the *gPDF*s increases, the confidence measure for both the sensor and the IMDB data sets become increasingly accurate as expected. For the IMDB and sensor data sets there is a strong correlation among the high score values. This correlation is not detected well assuming independence as illustrated in the quick rise in confidence. In contrast, the two-dimensional histograms can better predict high score values for unseen tuples. As shown in figures 8 and 9, the confidence stays low until a sufficient number of high value scores have been seen by the algorithm.

Histogram Size	One-Dimensional Histograms	Two-Dimensional Histograms
5	1.1238	11.5802
10	1.1598	11.6392
15	1.2068	11.5983
20	1.1778	11.7991
25	1.2107	11.7562

Table 5: Run time performance for Anytime TA using one- and two-dimensional histograms. Synthetic data set (1,000,000 tuples, 4 attributes, histograms size 20, random distribution). Time is reported in seconds.

Performance: In order to evaluate the performance of multidimensional histograms for Anytime TA we used a synthetic data sets and compared the running times for one- and two-dimensional histograms (*gPDF*s). Each of our results are averaged over five independent experiments. As shown in Table 5, using multidimensional *gPDF*s requires a significant overhead regardless of the size of the histograms. This is due to the increased number of updates required to maintain the multidimensional *gPDF*s for each round.

7. CONCLUSIONS

In this paper we have presented an anytime framework for top-k computations. Our framework can be applied on a variety of popular top-k algorithms (TA and TA-Sorted) and enable anytime behavior. We have discussed and analytically demonstrated several properties of our framework regarding the behavior of several measures of interest to anytime top-k computations. Through a de-

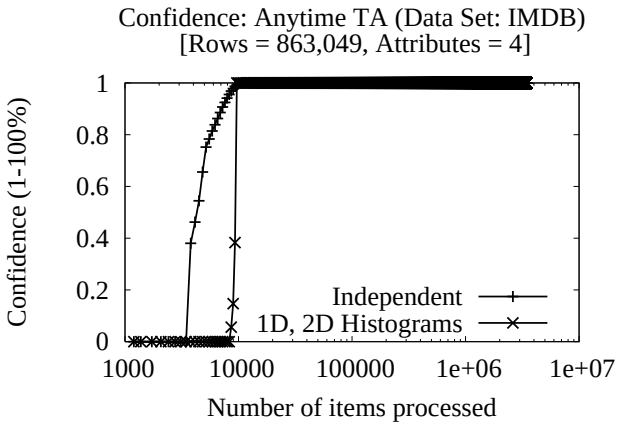


Figure 8: In this experiment we compare the confidence for one- and two-dimensional histograms using the IMDB data set where K=300.

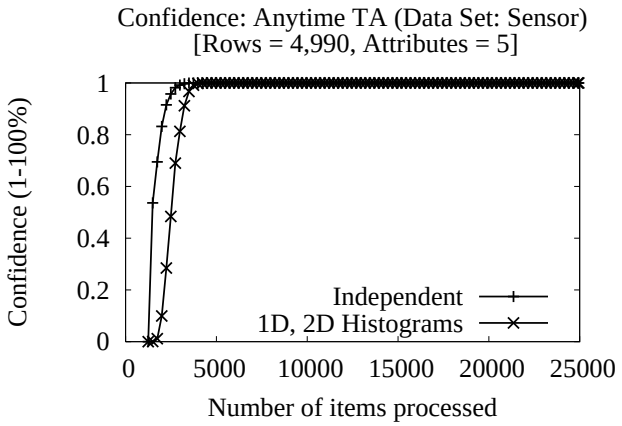


Figure 9: In this experiment we compare the confidence for one- and two-dimensional histograms using the sensor data set where K=100.

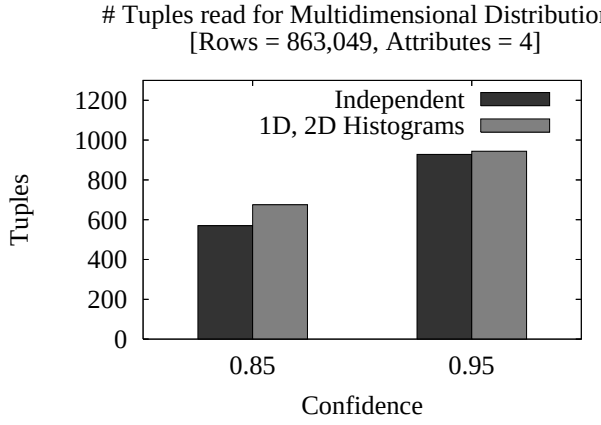


Figure 10: In this experiment we compare the number of tuples retrieved from the database using one- and two-dimensional histograms for Anytime TA using the IMDB data set and K=100.

tailed experimental study we have demonstrated the practical utility of our approach.

8. ACKNOWLEDGMENTS

The work of Dimitrios Gunopulos was supported by NSF (IIS 0330481, IIS 0534781). The work of Gautam Das was supported by unrestricted gifts from Microsoft Research and start-up funds from the University of Texas, Arlington.

9. REFERENCES

[1] L. G. A. Marian, N. Bruno. Evaluating Top-k Queries Over Web Accesible Sources. *TODS 29(2)*, 2004.

[2] N. Bruno, L. Gravano, and A. Marian. Evaluating Top-k Queries Over Web Accessible Databases. *Proceedings of ICDE*, Apr. 2002.

[3] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. pages 397–410, 1999.

[4] Y. chi Chang, L. Bergman, V. Castelli, C. Li, M. L. Lo, and J. Smith. The onion technique: Indexing for linear optimization queries. *Proceedings of ACM SIGMOD*, pages 391–402, 2000.

[5] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search Metric Spaces. *Proceedings of VLDB*, pages 426–435, Aug. 1997.

[6] D. Donjerkovic and R. Ramakrishnan. Probabilistic Optimization of Top-N Queries. *Proceedings of VLDB*, Aug. 1999.

[7] R. Fagin. Combining Fuzzy Information from Multiple Systems. *PODS*, pages 216–226, June 1996.

[8] R. Fagin. Fuzzy Queries In Multimedia Database Systems. *PODS*, pages 1–10, June 1998.

[9] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SODA*, pages 28–36, 2003.

[10] R. Fagin and E. Wimmers. Incorporating User Preferences in Multimedia Queries. *ICDT*, pages 247–261, Jan. 1997.

[11] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. *SIGMOD*, pages 463–474, 2000.

[12] U. Guntzer, W.-T. Balke, and W. Kiesling. Optimizing multi-feature queries for image databases. *The VLDB Journal*, pages 419–428, 2000.

[13] E. Horvitz. Reasoning About Belifs and Actions Under Computational Resource Constraints. *Proceedings of the Third Workshop on Uncertainty in AI*, 1987.

[14] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. *SIGMOD Conference*, pages 259–270, 2001.

[15] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB J.*, 13(3):207–221, 2004.

[16] S. H. K. Chang. Minimal Probing: Supporting Expensive Predicates for Top-k Queries. *SIGMOD*, 2002.

[17] R. S. M. Theobald, G. Weikum. Top-k Query Evaluation With Probabilistic Guarantees. *Proceedings of VLDB*, 2004.

[18] K. C. C. Mohamed Soliman, Ihab Ilyas. Top-k query processing in uncertain databases. *ICDE*, 2007.

[19] L. G. N. Bruno, S. Chaudhuri. Top-k Selection Queries Over Relational Databases: Mapping Strategies and Performance Evaluation. *TODS 27(2)*, 2002.

[20] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 281–290, 2001.

[21] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. *ICDE*, pages 22–29, 1999.

[22] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *The VLDB Journal*, pages 486–495, 1997.

[23] R. Fagin and A. Lotem and M. Naor. Optimal Aggregation Algorithms For Middleware. *PODS*, June 2001.

[24] C. Re, N. N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. pages 886–895, 2007.

[25] M. B. T. Dean. An Analysis of Time Dependent Planning. *Proceedings of the National Conference on AI*, 1988.

[26] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava. Ranked join indices. *ICDE*, 2003.