Performance of Multiattribute Top-K Queries on Relational Systems

Nicolas Bruno
Columbia University
nicolas@cs.columbia.edu

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Luis Gravano
Columbia University
gravano@cs.columbia.edu

Abstract

In many applications, users specify target values for the attributes of a relation, and expect in return the k tuples that best match these values. Traditional RDBMSs do not process these "top-k queries" efficiently. In our previous work, we outlined a family of strategies to map a top-k query into a traditional selection query that a RDBMS can process efficiently. The goal of such mapping strategies is to get all needed tuples (but minimize the number of retrieved tuples) and thus avoid "restarts" to get additional tuples. Unfortunately, no single mapping strategy performed consistently the best under all data distributions. In this paper, we develop a novel mapping technique that leverages information about the data distribution and adapts itself to the local characteristics of the data and the histograms available to do the mapping. We also report the first experimental evaluation of the new and old mapping strategies over a real RDBMS, namely over Microsoft's SQL Server 7.0. The experiments show that our new techniques are robust and significantly more efficient than previously known strategies requiring at least one sequential scan of the data sets.

1 Introduction

Approximate matches of queries are commonplace in the text world. Notably, web search engines answer user queries with a rank of the pages that best match the user specification. Ranked answers are also desirable for many applications that deal with traditional relational data, as illustrated in the following example.

Example 1: *Consider a real-estate database that maintains information like the* Price *and* Number of Bedrooms *of each house that is available for sale. Suppose that a potential customer is interested in houses with four bedrooms, and with a price tag of around* $\$300,000$. *The database system should then rank the available houses according to how well they match the given user preference, and return the top houses for the user to inspect. If no houses match the query specification exactly, the system might return a house with, say, five bedrooms and a price tag close to* $\$300,000$ *as the top house for the query.* ▮

A query for this kind of applications can be as simple as a specification of the target values for each of the relevant attributes of the relation. Given such a query, a database supporting approximate matches ranks the tuples according to how well they match the stated values for the attributes. Users who issue this kind of queries are typically interested in a small number of tuples k that best match the given condition, as in the example above. We refer to such queries as *top-k selection queries*.

This paper addresses the problem of efficient execution of top-k selection queries on relational databases. In earlier work, we presented techniques for mapping a top-k query into a traditional multiattribute range selection query that any RDBMS can then optimize and execute efficiently. Our techniques used multidimensional histograms to do this mapping. Intuitively, given a target data point, we consult the histogram to derive a multiattribute range query such that k closest matches are "likely" to be included in the answer to the generated range query. If the range selection query actually returns fewer than k tuples, the query needs to be "restarted," i.e., one or more supplemental queries need to be generated to ensure that all k closest matches are returned to the users. Naturally, a desirable property of any mapping is that it generates a range query that returns all k closest matches without requiring restarts. Our previous strategies for top-k query processing treated buckets as "atomic," without modelling data skews *within* the buckets. This led to query processing strategies that did not work uniformly well over varying data distributions. More specifically, for the different queries and data distributions that we considered, there was always one variant of our technique that worked well, but we could not predict automatically which one it would be for arbitrary data distributions and queries. Additionally, an open question is whether our query processing strategies are indeed more efficient than those requiring at least one sequential scan of the data on a real RDBMS.

In this paper, we introduce a new strategy for processing top-k queries that addresses the limitations of our previous techniques. We map a given top-k query into a multiattribute selection query by analyzing the data distribution around the target value specified in the query. In particular, we no longer treat histogram buckets as "atomic," and instead develop efficient techniques for determining the optimal *fraction* of each bucket that we should include in the final selection query. The fact that our top-k queries involve *multiple attributes* makes this task especially challenging. Another key aspect of our new strategy is that we model *intra-bucket skews*. Ideally histograms would only "bucketize" data regions that exhibit reasonably uniform densities. Unfortunately, building multidimensional histograms with this characteristic is particularly difficult. To account for imperfect histogram buckets, we introduce a single value in each histogram bucket computed using a variation of the fractal dimension concept, and which models multidimensional data skews within buckets. As we will see, this measure of bucket skew is fundamental to drastically reduce the fraction of queries that require restarts. We also comment on how the quality of histograms can influence our mapping and indeed show an example of a data set where the inability of current histogram strategies to represent the data distribution adequately impacts our mapping.

As another key contribution, we report the first experimental evaluation of our multiattribute top-k query mappings over a commercial RDBMS. Specifically, we evaluate the execution time of our query processing strategies over Microsoft's SQL Server 7.0 over a number of data distributions, and other variations of relevant parameters. As we will show, our techniques are robust, and establish the superiority of our schemes over previously known mapping strategies as well as over the techniques requiring sequential scans.

The paper is organized as follows. In Section 2, we present background on top-k query processing and review our earlier techniques. Section 3 presents our new mapping strategy. Section 5 presents the experimental evaluation of our techniques on Microsoft's SQL Server 7.0 using the experimental setting of Section 4. Finally, Appendix A discusses the impact of the quality of multidimensional histograms on the performance of our top-k query processing approach.

2 Background

In a traditional relational system, the answer to a selection query is a set of tuples. In contrast, the answer to a *top-k query* is an *ordered* set of tuples, where the ordering criterion is how well each tuple matches the given query. In this section we review the query model and evaluation strategies that we introduced in [5].

2.1 Query Model

Consider a relation R with attributes $A_1, \ldots, A_n$. A top-k query over R specifies target values for the attributes in R and a *distance function* over the tuples in the domain of R. The result of a top-k query q is then an ordered set of k tuples of R that are closest to q according to the given distance function.[1]

Example 2: *Consider a relation* Employee *with attributes* age *and* salary. *The answer to the top-10 query* $q = (25,\ 40,000)$ *is an ordered sequence consisting of the 10 employees in the* Employee *relation that are* closest *to 25 years of age and $40,000 of salary, according to a given distance function.* ▮

[1]In [5] we used *scoring functions* instead of distance functions in our definition of top-k queries. These two definitions are conceptually equivalent. An advantage of the current definition is that it does not require attribute values to be "normalized" to a [0, 1] range.

In this paper, we restrict our attention to top-k queries over continuous-valued real attributes, and to distance functions that are based on *vector p-norms*, defined as:

$$||x||_p = \left(\sum_i |x_i|^p\right)^{1/p} \qquad (p \geq 1)$$

Given a p-norm $||\cdot||$, we can define a distance function $D_{||\cdot||}$ between two arbitrary points q and t as $D_{||\cdot||}(q,t) = ||q-t||$. All distance functions based on p-norms verify the following monotonicity property:

Property 1: *Let $q = (q_1, \ldots, q_n)$ be a query, and $t = (t_1, \ldots, t_n)$ and $t' = (t'_1, \ldots, t'_n)$ be two arbitrary tuples such that $\forall i\ |t'_i - q_i| \leq |t_i - q_i|$. (In other words, t' is at least as close to q as t for all attributes.) Then, for any p-norm $||\cdot||$, $D_{||\cdot||}(q,t') \leq D_{||\cdot||}(q,t)$.*

This paper focuses on the following important distance functions, which are based on p-norms for $p = 1, 2$, and ∞.

Definition 1: *Consider a relation $R = (A_1, \ldots, A_n)$ with real-valued attributes. Then, given a query $q = (q_1, \ldots, q_n)$ and a tuple $t = (t_1, \ldots, t_n)$ from R, we define the* distance *between q and t using any of the following three distance functions* [2]*:*

$$\begin{array}{rclcl} \mathit{Sum}(q,t) & = & ||q-t||_1 & = & \sum_{i=1}^{n} |q_i - t_i| \\ \mathit{Eucl}(q,t) & = & ||q-t||_2 & = & \sqrt{\sum_{i=1}^{n} (q_i - t_i)^2} \\ \mathit{Max}(q,t) & = & ||q-t||_\infty & = & \mathit{max}_{i=1}^{n} |q_i - t_i| \end{array}$$

2.2 Evaluation Strategies

The main idea for processing a top-k query q is to map it into a relational selection query that any RDBMS can execute. The procedure for doing so consists of the following three steps:

Search Given a top-k query q over R, use a multidimensional histogram H to estimate a search distance d, such that the region that contains all possible tuples at distance d or lower from q, $\mathit{reg}(q,d)$, is expected to include k tuples.

Retrieve Retrieve all tuples in $\mathit{reg}(q,d)$ using a range query that encloses this region as tightly as possible.

Verify/Restart If there are at least k tuples in $\mathit{reg}(q,d)$, sort and return them. Otherwise, *restart* the procedure using a "safe" distance that guarantees the retrieval of the top-k answers.

The first step is the most challenging one. Ideally, the search distance d that we determine encloses exactly k tuples. Unfortunately, identifying such a precise value for d using only relatively coarse histograms is not possible. In practice, we will try to find a value of d such that $\mathit{reg}(q,d)$ encloses at least k tuples, but not many more. Choosing a value of d that is too high would result in an execution that does not require restarts (**Verify/Restart** step), but that would retrieve too many tuples, which is undesirable. In contrast, choosing a value of d that is too low would result in an execution that requires restarts, which is also undesirable. Hence, determining the right distance d becomes the crucial step in our top-k query processing strategy. Once the **Search** step determines the search distance d, the **Retrieve** step builds a SQL query that encloses $\mathit{reg}(q,d)$ as tightly as possible, as illustrated in the following example.

Example 3: *Consider relation R with attributes A_1 and A_2, and a top-k query $q = (10, 30)$ with the Sum distance function. Let $d = 5$ be the search distance determined in the* **Search** *step. The following SQL query then encloses all possible tuples inside reg(q,d):* `SELECT * FROM R WHERE 5<=A1<=15 AND 25<=A2<=35.` ∎

[2] Our definitions of distance give equal weight to each attribute of the relation, but we can easily modify them to assign different weights to different attributes if this is appropriate for a specific scenario.

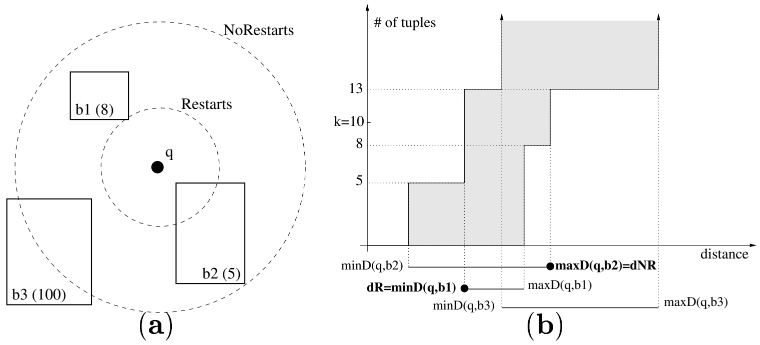


Figure 1: **(a)** Regions searched by the *Restarts* and *NoRestarts* strategies for a top-10 query q. **(b)** The corresponding *dR* and *dNR* distance values.

Our technique uses multidimensional histograms as the only source of information about relation R to determine distance d. An *n-dimensional histogram* H over R roughly describes the distribution of values in R. It consists of a set of pairs $H = \{(b_1, t_1), \ldots, (b_m, t_m)\}$, where each *bucket* b_i defines a hyper-rectangle included in *domain*(R), and each *frequency* t_i is the number of tuples in R that lie inside b_i. The buckets b_i are pairwise disjoint, and every tuple in R is contained in one bucket.

In our previous work [5] we showed a family of strategies for determining search distances. Briefly, given a relation R with an associated histogram H, we conceptually created different small, synthetic relations consistent with H, and used them to define four fixed execution strategies. The first one, *NoRestarts*, results in a search distance that is high enough to guarantee that no restarts are ever needed. In other words, the **Verify/Restart** step always finishes successfully, without having to restart the whole process. In contrast, the second strategy, *Restarts*, gives the lowest search distance that *might* result in no restarts. This strategy retrieves the minimum possible number of tuples, but it frequently, if not always, leads to restarts (hence its name). Finally, strategies *Inter1* and *Inter2* use two "hardcoded" intermediate search distances between *Restarts* and *NoRestarts*.

Example 4: *Consider a relation with 113 tuples, and the three-bucket histogram of Figure 1(a). For example, bucket* b_1 *encloses 8 tuples of the relation. Let* q *be a top-10 query. The NoRestarts strategy for this query determines a "safe" search distance that is guaranteed to enclose at least 10 tuples. In effect, we can see that the NoRestarts region encloses histogram buckets* b_1 *and* b_2 *completely, hence including at least* $8 + 5 = 13$ *tuples. Unfortunately, this strategy will most likely also retrieve a significant fraction of the 100* b_3 *tuples, and may thus be inefficient. In contrast, the Restarts strategy for query* q *determines an "optimistic" search distance that might result in 10 tuples being retrieved. As we see in the figure, the Restarts region will only enclose 10 tuples in the "best" case when the 5 tuples in bucket* b_2 *are as close to* q *as possible, and at least 5 of the* b_1 *tuples are as close to* q *as the 5* b_2 *tuples. Unfortunately, this optimistic scenario is improbable, and the Restarts strategy will most likely result in restarts (***Verify/Restart** *step) and in an overall inefficient execution.* ∎

In [5] we compared the *Restarts*, *NoRestarts*, *Inter1*, and *Inter2* strategies experimentally in terms of the number of objects retrieved by the different strategies. The performance of these strategies varies drastically with different data distributions, and, unfortunately, none of the strategies worked consistently the best. Moreover, even over the same data sets, the best strategy to choose for a query might be dependent on the specific location of the query. In the next section, we present a new technique that adapts to the characteristics of the data distribution around the vicinity of the queries.

3 Answering Top-K Queries

As discussed in the previous section, a critical step when processing a top-k query q is determining a good search distance d. This distance should be large enough so that no restarts are needed (i.e., the number of tuples at distance d or less from q should be at least k). At the same time, this distance should be as small as possible, so that we do not retrieve too many tuples during query processing. In this section, we introduce a new strategy for determining a good search distance d for a query. In Section 5 we show experimentally that our new technique outperforms all of the old strategies [5], and manages to process top-k queries in a fraction of the time required to perform a single sequential scan of the relation, even considering the time required for restarts in the rare cases when they are needed.

Our technique starts by identifying the range of values that "good" search distances can take for a given query. Distances outside of this range would result in either too few or too many tuples being retrieved. For each distance d that we consider in this range, our technique estimates the number of tuples in $reg(q, d)$. The final search distance is then the lowest distance d such that $reg(q, d)$ is estimated to include at least k tuples. In summary, we identify the search distance d for a top-k query q (**Search** step) as follows:

1. Determine the distance range $[dR, dNR]$ to explore (Section 3.1).

2. Use binary search to find $d \in [dR, dNR]$ (Section 3.2) such that: (a) the estimated number of tuples in $reg(q, d)$ is at least k, and (b) if $d' < d$ then the estimated number of tuples in $reg(q, d')$ is below k. (Reference [7] has used Golden Search in a similar context to determine a search "cutoff" for processing top-k queries where the ranking condition is on a *single attribute*.)

Once the search distance d is determined, in the **Retrieve** step we build a selection query as tightly as possible to retrieve all tuples enclosed in $reg(q, d)$ as suggested in the previous section. Finally, in the **Verify/Restart** step, if we have retrieved fewer than k tuples at distance d or lower, we then use dNR as the new "safe" search distance and restart the procedure. This time around, however, we are guaranteed to retrieve at least k tuples, as we will see in Section 3.1.

3.1 Bounding the Search Distance d

Our technique for finding the search distance d for a top-k query starts by tightly bounding the potential range of values for d. The lower and upper bounds of this range correspond to an optimistic and a pessimistic scenario, respectively. In the optimistic scenario, all of the tuples in a bucket b are assumed to be at the point in b that is closest to query q, with distance $minD(q, b)$. Analogously, the pessimistic scenario assumes that all of the tuples in b are at the point in b that is farthest from query q, with distance $maxD(q, b)$. Since the norm-based distance functions that we use are monotonic (Property 1), the *minD* and *maxD* values are easily computed. In effect, the point in a bucket b that is closest to (similarly, farthest from) a query q can be determined dimension by dimension as the following example illustrates.

Example 5: *Consider a bucket b defined by its corners* $(10, 10)$ *and* $(25, 40)$*, and a query* q=$(40, 20)$ *(Figure 2). Assume that we use the Eucl distance function. Because of the monotonicity property of Eucl the point in b that is closest to q,* q_1*, is the one that is closest dimension by dimension. Hence* $q_1 = (25, 20)$ *(Figure 2). Analogously, the point in b that is farthest from q,* q_2*, is the one that is farthest dimension by dimension. Hence* $q_2 = (10, 40)$*. Consequently,* $minD(q, b)$=$Eucl(q, q_1)$=$\sqrt{(40-25)^2 + (20-20)^2}$=$15$ *and* $maxD(q, b)$=$Eucl(q, q_2) = \sqrt{(40-10)^2 + (20-40)^2}$=$36.1$. ∎

After calculating $minD(q, b_i)$ and $maxD(q, b_i)$ for each bucket b_i, we can use these distances for defining the optimistic and pessimistic scenarios. To determine the search distance dR in the optimistic scenario, we consider the histogram buckets in increasing order of $minD(q, b_i)$ (Figure 1(b)). When we consider a bucket b_i, we assume that all its tuples are situated at distance $minD(q, b_i)$ from query q, i.e., as close as possible to q, and therefore we add b_i's frequency to the total number of tuples at that distance or lower. If after "including" a bucket b_j we have included at least k tuples, we define $dR = minD(q, b_j)$. If tuples were distributed as assumed in our optimistic scenario, then there would be at least k tuples at distance $dR = minD(q, b_j)$ or lower from query q. For the pessimistic approach, we proceed similarly, but considering $maxD(q, b_i)$ instead of $minD(q, b_i)$. In other words, we assume that all tuples in b_i are placed as far as possible from q. If after "including" a bucket b_j we have included at least k tuples, we define $dNR = maxD(q, b_j)$. Given that we assumed a worst-case scenario for the tuple distribution with respect to query q, we are guaranteed that there are at least k tuples at distance $dNR = maxD(q, b_j)$ or lower from query q.

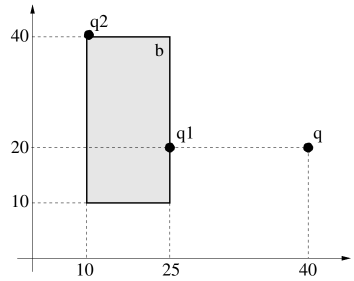


Figure 2: The points in bucket b that are closest to (q_1) and farthest from (q_2) query q.

Example 4: (cont.) *Figure 1(b) illustrates how we determine dR and dNR for the relation and histogram of Example 4 and Figure 1(a). If tuples were distributed inside each bucket as close as possible to top-10 query q (optimistic scenario) then there would be 5 tuples in bucket b_2 at distance minD(q, b_2), and 8 tuples in bucket b_1 at distance minD(q, b_1). (minD(q, b_3) is larger than both minD(q, b_2) and minD(q, b_1).) Since 5+8 exceeds $k = 10$, we can "stop" after considering bucket b_1, and define dR as minD(q, b_1). Analogously, if tuples are distributed as far away as possible from q inside each bucket, then there would be 8 tuples in bucket b_1 at distance maxD(q, b_1), and 5 tuples in bucket b_2 at distance minD(q, b_2). (maxD(q, b_3) is larger than both maxD(q, b_1) and maxD(q, b_2).) Since 8+5 exceeds $k = 10$, then we can "stop" after considering bucket b_2, and define dNR as maxD(q, b_2).* ∎

3.2 Finding the Best Search Distance d

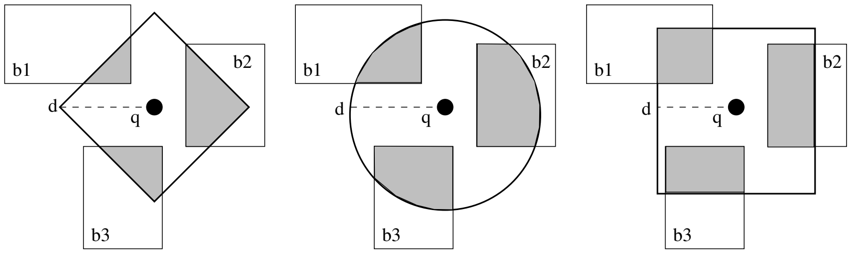
In the previous section, we described how to compute the range $[dR, dNR]$ that includes the search distance that we should use to evaluate a given top-k query. We now show how we find this search distance.

Let q be a top-k query and $[dR, dNR]$ the distance range as computed in Section 3.1. The search distance d that we use for q should be such that at least k tuples are at distance d or lower from q. In other words, $reg(q, d)$ should contain at least k tuples. Furthermore, d should be the minimum distance with this property. We saw in Section 3.1 that $d \in [dR, dNR]$. Since we only have a histogram describing relation R, we cannot expect to determine the exact number of tuples in $reg(q, d)$ for every candidate distance d that we consider. Instead, we will build an *estimate* of this number of tuples, $eTuples(reg(q, d))$, using the histogram information. More specifically,

$$eTuples(reg(q, d)) = \sum_{b \text{ bucket}} eTuples(b \cap reg(q, d))$$

In other words, we will compute an estimate of the number of tuples of $reg(q, d)$ by adding an estimate of the number of tuples in $b \cap reg(q, d)$ for every histogram bucket b. For this purpose, we consider the following three cases, assuming that bucket b contains t tuples:

- If $b \subseteq reg(q, d)$ (or, equivalently, if $maxD(q, b) \leq d$) then $eTuples(b \cap reg(q, d)) = t$.
- If $b \cap reg(q, d) = \emptyset$ (or, equivalently, if $minD(q, b) > d$) then $eTuples(b \cap reg(q, d)) = 0$.
- Otherwise $reg(q, d)$ partially overlaps with bucket b (i.e., $minD(q, b) \leq d < maxD(q, b)$). In this case, we first estimate the fraction of volume of the overlapping area, f_v (Section 3.2.1). If the t tuples in bucket b were uniformly distributed in b, then $b \cap reg(q, d)$ would contain exactly $f_v \cdot t$ tuples. Unfortunately, data skew is frequent even within histogram buckets. To take this into account, we will estimate a lower bound on the number of tuples in $b \cap reg(q, d)$ as $f_v^{\alpha} \cdot t$ tuples, where α is a bucket-dependent constant that models intra-bucket skew (Section 3.2.2).



(a) *Sum* (b) *Eucl* (c) *Max*

Figure 3: Shapes of *reg*(q,d) for different distance functions.

3.2.1 Estimating Volume Overlap

As discussed above, we now describe how to compute $Vol(b \cap reg(q,d))$ for a bucket b, query q, and distance d when b and *reg*(q,d) overlap partially. The fact that our histograms and queries are *multiattribute* makes this step particularly challenging. As Figure 3 illustrates, the shape of *reg*(q,d) depends dramatically on the distance function used. This variety in shapes will result in turn in different shapes for the $b \cap reg(q,d)$ regions. In some cases (e.g., for the *Max* distance function), we can define a closed formula for $Vol(b \cap reg(q,d))$. In some others (e.g., for *Sum*, *Eucl*, or a user-specified function) we need to resort to discrete estimation methods for this volume:

Closed formula for determining $Vol(b \cap reg(q,d))$: For some distance functions we can determine this volume analytically. The *Max* function is one such distance function. Given a query $q = (q_1, \ldots, q_n)$ and a distance d, we can determine the volume of the intersection of *reg*(q,d) and the bucket b delimited by the corners $low = (l_1, \ldots, l_n)$ and $high = (h_1, \ldots, h_n)$ as follows:

$$Vol(b \cap reg(q,d)) = \prod_{i=1}^{n} max\{0, h'_i - l'_i\}$$

where $h'_i = min\{h_i, q_i + d\}$ and $l'_i = max\{l_i, q_i - d\}$.

Discrete estimation of $Vol(b \cap reg(q,d))$: For some distance functions for which no closed formula exists we need to resort to a stochastic, discrete estimation procedure for $Vol(b \cap reg(q,d))$ such as Montecarlo [16]. We generate random points inside bucket b and count the fraction of them that lie inside *reg*(q,d)[3]. Finally, we multiply this fraction by the total volume of the bucket. This procedure works for all distance functions, and gives close-to-perfect answers. Unfortunately, the Montecarlo approximation of $Vol(b \cap reg(q,d))$ might prove expensive to perform at run time. In Section 5, we consider coarse approximations of *reg*(q,d) for *Sum* and *Eucl* that allow us to calculate their intersection with the buckets as efficiently as when we are able to use a closed formula. As we will see, these inexpensive approximations work surprisingly well in practice.

3.2.2 Modelling Intra-Bucket Data Skew

Consider a bucket b with t tuples such that $Vol(b \cap reg(q,d)) = f_v \cdot Vol(b)$ for a query q and a distance d. As discussed above, if the data were distributed completely uniformly within b, then $b \cap reg(q,d)$ would enclose exactly $t \cdot f_v$ tuples. With skewed data, sometimes the actual number of tuples in $b \cap reg(q,d)$ would be higher than its expected share of b tuples. In this case, our top-k query processing strategy would retrieve more tuples than anticipated (and needed), with a typically small impact on efficiency. Some other times, though, the actual number of tuples in $b \cap reg(q,d)$ would be lower than our estimate. This case is more serious, since retrieving fewer tuples than expected might result in costly restarts. Hence for top-k query processing, these two cases should *not* be treated as equals, and we will try to avoid overestimating the number of tuples in $b \cap reg(q,d)$ whenever possible,

[3]Of course, this Montecarlo estimation is performed using only the bucket boundaries, without accessing the actual relation.

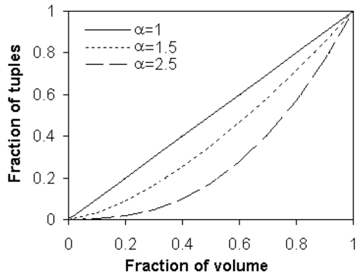


Figure 4: Fraction of tuples as a function of the corresponding fraction of volume, for different values of α.

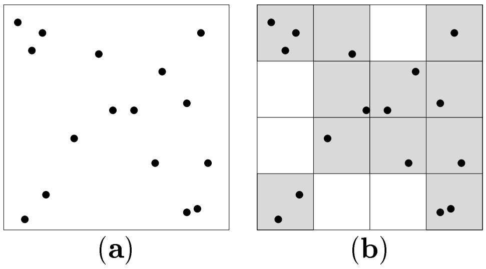


Figure 5: **(a)** Distribution of tuples inside a bucket. **(b)** The 4×4 grid used by the box-counting procedure.

even when this might result in underestimates. Therefore, when computing *eTuples*$(b \cap reg(q,d))$ we will take a more pessimistic view of the contents inside a bucket, to reduce the expected number of tuples and minimize the probability of restarts, but still without retrieving too many tuples. We estimate a lower bound on the number of tuples in $b \cap reg(q,d)$ as follows:

$$eTuples(b \cap reg(q,d)) = t \cdot f_v^{\alpha} \quad (1)$$

where $\alpha \geq 1$ is a *"deflation"* parameter in each bucket that models the local degree of skew of the data inside the bucket. This extra number is kept in each histogram bucket, and is computed during histogram construction. The introduction of α is related to the ϵ parameter that reference [7] associates with each single-attribute histogram for modelling histogram quality.

Since $\alpha \geq 1$ and $f_v \leq 1$, the new estimate in Equation 1 is never higher than the one that assumes uniformity. Intuitively, when $Vol(b \cap reg(q,d))$ is near zero, we are not covering much of the bucket, so our knowledge is minimal and we decrease the expected number of tuples. As $Vol(b \cap reg(q,d))$ increases, we can be less pessimistic about the contents of the bucket, until we reach $Vol(b \cap reg(q,d)) = Vol(b)$ and we know exactly that we will get all the tuples (Figure 4).

We experimentally tested several different functions for modelling the degree of skew, i.e., α, inside each bucket. We obtained the best results with a simple metric derived from the *box-counting* procedure, used previously in [8] for calculating the fractal dimension of data sets. Our adaptation of the box-counting procedure works as follows. Given a bucket b, we first build a multidimensional grid consisting of t cells, where t is (approximately) the number of tuples in the bucket. Then, we count the number of cells c that enclose at least one tuple. Finally, we define $\alpha = \log(t)/\log(c)$.

Example 6: *Consider the bucket in Figure 5(a), with 16 tuples. We build a* 4×4 *grid (*$t = 16$*) and count the number of cells with at least one tuple (Figure 5(b)), which results in* $c = 11$*. Then, the degree of skew associated with this bucket is* $\alpha = \log(16)/\log(11) = 1.16$. ∎

The rationale behind this choice of α is as follows. In a completely uniform distribution, every cell in a bucket would be occupied by exactly one tuple, so the associated α would be one. Then, Equation 1 reduces to the uniformity assumption. As the data distribution inside a bucket is more skewed, it generally tends to form "clusters" and leaves big regions virtually empty. In this case, the value of α increases, which results in a more "pessimistic" strategy. Then, on average we will retrieve slightly more tuples than with the uniformity assumption, but the percentage of restarts will decrease as well, leading to a better overall performance, as we will see in Section 5.

4 Experimental Setting

This section describes the data sets, histograms, and metrics for the experiments of Section 5.

4.1 Data Sets

Our experiments use several synthetic data sets built using different Zipfian distributions [17], and model different degrees of data correlation. For this purpose, we used a subroutine that generates one-dimensional Zipfian distributions with varying "Z" factors. When this factor is zero, it generates a uniform distribution. Higher values result in higher skew. For an n-dimensional data set, the generation routine is parameterized by (1) a vector $< z_1, \ldots, z_n >$ of values, one for each attribute, and (2) the number of tuples to be generated, N. We generated the data corresponding to a (Z, N) specification as follows. First, we generated a one-dimensional Zipfian distribution of N tuples for attribute A_1 using Z factor z_1. Let us say that for attribute A_1 the value v_1 occurred in N_1 out of the N tuples. We now fill in the value for attribute A_2 for each of these N_1 tuples by generating N_1 values $w_1, \ldots, w_{N_1}$ using a Zipfian distribution with Z factor z_2. At the end of this step, the first two attributes of the original N_1 tuples are filled in with values $(v_1, w_1), \ldots, (v_1, w_{N_1})$. Let us say that this results in N_2 tuples that have v_1 and w_1 as the values for attributes A_1 and A_2, respectively. We then fill in the remaining attribute values $A_3, \ldots, A_n$ for these N_2 tuples in an analogous way as above, using the Z values z_3 through z_n. For our experiments, we generated data sets of 10^4 to 10^6 records, with 2, 3, and 4 attributes. The domain for each attribute was the set of integers in $[0 \ldots 10^6)$. We varied the Zipfian vectors in the generation of the data sets with values between 0 and 3 to obtain a spectrum of skews. For conciseness, we will refer to the data set generated with $Z =< 2, 1, 1 >$, for example, as $Z211$.

4.2 Histograms

We use the *MHIST*-2 procedure as the current state of the art technique for building multidimensional histograms, with *MaxDiff*(V, F) as the underlying one-dimensional partitioning strategy [13, 14]. We tried other variations of the one-dimensional partitioning strategy and obtained worse results, so we present the results using *MaxDiff*(V, F). We refer the reader to [13, 14] for a detailed discussion.

4.3 Indexes

We tried, for several configurations, Microsoft's *Index Tuning Wizard* over SQL Server 7.0 [6], an automatic tool that determines good index configurations for a specific workload. We fed the Index Tuning Wizard with representative query workloads for our task and it always suggested an n-column index covering all attributes in the top-k queries. Therefore, we focused on multiattribute indexes in most of our experiments, assuming that in a real situation where top-k queries are heavily used, this index could be built without big penalties in the overall system efficiency. We also ran experiments for the case when only single-column indexes are available. In summary, we used two main index configurations: (a) n unclustered single-column B$^+$-tree indexes, one for each attribute mentioned in the query; and (b) one unclustered n-column B$^+$-tree index whose search key is the concatenation of all n attributes mentioned in the query.

For the n-column index configurations, we needed to define the order in which the attributes would be concatenated to form the index search keys. We considered different choices, and finally concatenated the attributes in decreasing order of their number of distinct values in the relation. This configuration results in good performance for the kind of n-attribute range queries that our top-k processing strategy generates.

4.4 Efficiency Metrics

For each configuration, we generated two different 500-query workloads. The first one, denoted *Random*, consists of queries randomly chosen from *domain(R)*. The second one, denoted *Skewed*, follows the data distribution and consists of tuples that are present in the original data set, chosen according to their original frequencies. Section 5 reports experimental results for these workloads, using the following metrics:

a) *Percentage of restarts*: This is the percentage of queries in the workload for which the associated selection query failed to contain the k best tuples, hence leading to restarts. (See the algorithm in Section 2.2.)

b) *Optimum* time: Suppose that we somehow know the top k tuples for a given query q. As a baseline, we compute the running time of the "perfect" n-dimensional range query that results from tightly enclosing these actual top k tuples. Of course, this ideal technique would only be possible with complete knowledge of the data distribution, and never requires restarts. Its running time is a lower bound for that of our strategies.

c) *α-based (NR)* time: This is the average time to run those n-attribute range queries produced by our α-based technique that did not need restarts. As we will see, the overwhelming majority of top-k queries will not require restarts, so it is interesting to report their run time separately.

d) *α-based (total)* time: This is the average time to run *all* the n-attribute range queries produced by our α-based technique, whether they required restarts or not. For those queries that needed restarts, this time includes the running time for the original (insufficient) query, plus the time for the subsequent "safe" query that retrieves all of the needed tuples using distance *dNR* (Section 3).

e) *Sequential Scan* time: The techniques in [3, 4] for processing top-k queries require one sequential scan of the relation, plus a subsequent sorting step of a small subset of the relation, as we discuss in Section 6. (We ignore this sorting step in our experiments, the same way we ignore it when evaluating our techniques. This step is negligible relative to the rest of the processing.) We compare the run time of our new technique against that of a sequential scan of the relation, which is a lower bound on the time required by [3, 4]. To make our comparison as favorable as possible to the sequential scan case, we proceed as follows. Consider a top-k query involving attributes $A_1, \ldots, A_n$ of relation R. In practice, R is likely to have additional attributes that do not participate in the query. For the cases when we have available a multiattribute B$^+$-tree over the concatenation of attributes $A_1, \ldots, A_n$, the sequential scan will do an index scan (using the leaf nodes of the tree), rather than scanning the actual relation, which is larger due to the additional attributes not involved in the query. For this, we time the sequential scan over a projected version of R with just attributes $A_1, \ldots, A_n$. For the cases when we do not have a multiattribute B$^+$-tree, we time the sequential scan over the actual relation R. We model potential additional attributes not in the queries with an attribute A_{n+1} that is a string of 30 characters. In any case, the resulting *Sequential Scan* time that we use to compare against is a "loose" lower bound on the time that the techniques in [3, 4] would require to process a multiattribute top-k query like the ones we address in this paper.

5 Experimental Results

This section presents experimental results for our technique of Section 3. We ran all our experiments over Microsoft's SQL Server 7.0, on a 550-Mhz Pentium III PC with 384 MBytes of RAM. The experiments involve a large number of parameters, and we tried many different value assignments. For conciseness, we report results on a *default setting* where appropriate. This default setting uses data sets with $d = 3$ attributes, built using $Z211$ (moderate skew), and $N = 10^5$ tuples (Section 4.1). The *MHIST* histograms use 100 buckets (corresponding to approximately 3000 bytes of storage for d=3). We used multiattribute indexes with the attribute ordering described in Section 4.3. For each experiment, we used the 500-query *Random* workload, and asked for the top-10 tuples (i.e., $k = 10$) using *Max* as our distance function. We report results for other settings of the parameters as well.

Comparison with Existing Approaches

Our first experiment compares the efficiency of our technique with that of the strategies in [5]. As discussed in Section 3, our new *α-based* strategy adapts to the data distribution in the vicinity of the query, unlike the *Restarts*,

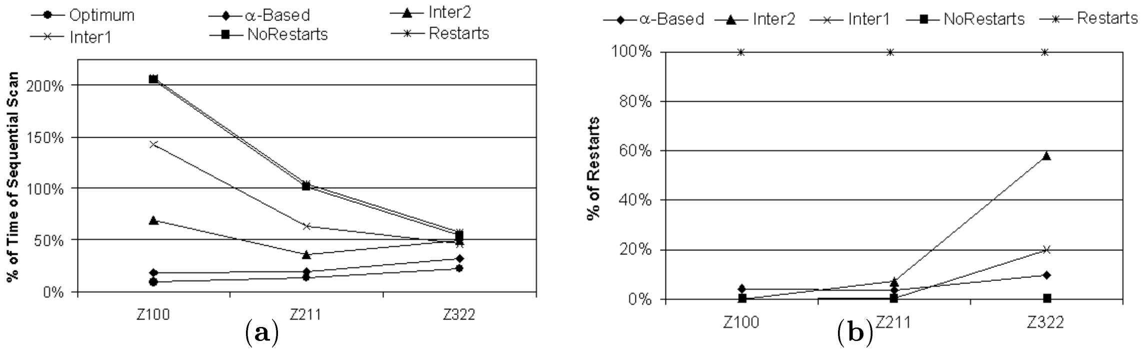


Figure 6: Run time **(a)** and percentage of restarts **(b)** of our new technique vs. that of the techniques in [5] and the optimum technique, for increasing data skew, as a percentage of the time required for a sequential scan of the relation.

NoRestarts, *Inter1*, and *Inter2* techniques. Figure 6(a) shows the running time that SQL Server took to process the selection queries produced by the various techniques for increasing data skew. We report the running times as a percentage of the time that a simple sequential scan of the relation would take, as discussed in Section 4.4. As we can see, both *Restarts* and *NoRestarts* result in the worst performance, due to the extreme assumptions they made. Also, while *Inter2* is better than *Inter1* for $Z100$ and $Z211$, *Inter1* performs better for $Z322$, that is, no one strategy performed the best in all configurations. In contrast, our new α*-based* technique consistently outperforms the four fixed strategies. Figure 6(a) shows that our new technique adapts itself well to the characteristics of the data, with run times that are always less than 35% those of a sequential scan of the data, as low as just 18% for $Z100$, and 19% for $Z211$. The percentage of queries that need restarts is also low, as shown in Figure 6(b). Finally, if we focus on the queries that do not need restarts, the average number of tuples retrieved by our α*-based* strategy is small, and close to the minimum possible, which is what the *Optimum* strategy retrieves. For example, for the $Z211$ data set, our technique retrieves on average only 78 out of the $100,000$ tuples in the relation for the 96.4% of the queries that did not require restarts. The optimum strategy retrieves on average only slightly fewer tuples for that data set, namely 44. In contrast, the four non-adaptive techniques of [5] return significantly more tuples. For example, *Inter1* returns on average 7,400 tuples, and *Inter2*, 1,300 tuples.

Effect of Modelling Intra-Bucket Skew

Figure 6 established that our α*-based* technique is better than all of the previous strategies, and significantly more efficient than performing a single sequential scan of the data set. A key part of our technique is associating an α value with each bucket, modelling how much the data inside the bucket departs from uniformity (Section 3.2.2). To analyze the impact of these local α's, we ran experiments for which we assumed $\alpha = 1$ in each bucket. In other words, we tested how our technique would perform if it assumed that the data inside each bucket was uniformly distributed. Since in this case we do not need to store the value of α in each bucket, we increased the number of buckets in the histogram accordingly to make the comparison fair. Figure 7 shows the five metrics from Section 4.4 for different data skews. Each data set (i.e., $Z100$, $Z211$, and $Z322$) has a group of four bars associated with it. The leftmost bar in each group indicates the run time of the *Optimum* strategy, while the rightmost one, that of a sequential scan of the relation. Neither of these techniques can ever need restarts. The two middle bars correspond to our technique with α set to one, with the division of run times explained in Section 4.4. The percentage of queries that needed restarts for each data set is placed in parentheses next to the corresponding label on the X axis. (For example, our technique with $\alpha = 1$ resulted in 39.8% of the queries requiring restarts for data set $Z100$.)

As we can see from Figure 7, the technique resulting from setting all the α's to one performs significantly worse than our α*-based* technique, especially regarding the percentage of queries that need restarts. In this case, 39.8% of the queries requires restarts for $Z100$, 28.4% for $Z211$ and 49.4% for $Z322$. (Contrast these numbers with those from Figure 6(b) for our α*-based* technique.) As we can see, the time spent by the technique in the cases when

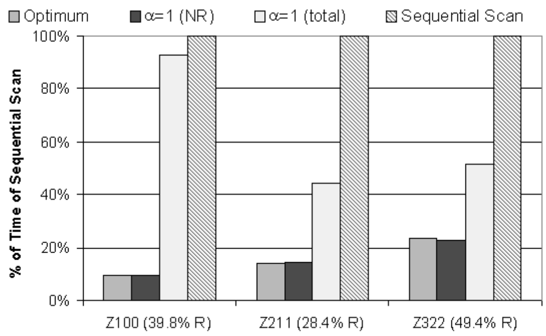


Figure 7: Run time and percentage of restarts as a function of the data skew, assuming that data is uniformly distributed inside the histogram buckets (i.e., $\alpha = 1$).

there are no restarts is quite good (in fact, almost equal to the optimum time). The overall results for the $Z100$ relation are worse than those for the more skewed relations due to the particular histogram configuration. In the $Z100$ case, each histogram bucket tends to span entire hyperplanes in the domain of the relation. Therefore, the *dNR* distance covers almost all the data set and each time the technique restarts, it retrieves over 90% of the tuples, greatly impacting the total execution time. As a conclusion, this experiment provides evidence that the use of local α's to model intra-bucket skew pays off, and achieves the goal of reducing the number of restarts significantly.

Robustness of our Approach

To analyze the robustness of our new strategy, we started with the default setting of the different experiment parameters, and varied them one at a time.

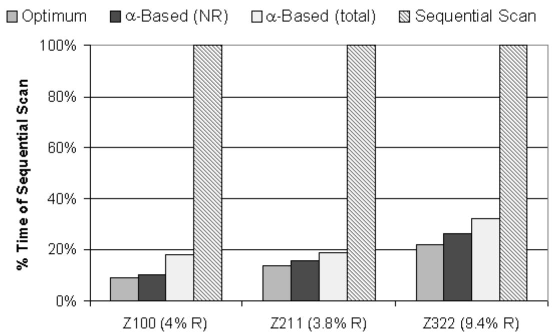
Figure 8(a) shows that, as the *data skew* increases, the total time spent to answer the queries also increases slowly relative to the time required by a sequential scan. However, the optimum time is larger too, and even when considering very skewed data sets (e.g., $Z322$) the performance of our technique is significantly better than the cost of a single sequential scan over the relation (around 20%-30% of the time of a sequential scan). Also, the percentage of queries in need of restarts is consistently low for the different data skews (4% for $Z100$, 3.8% for $Z211$, and 9.4% for $Z322$).

Figure 8(b) shows that the run time of our technique increases very moderately as the *dimensionality* of the data set increases. The percentage of queries that need restarts remains low at under 11% in all cases.

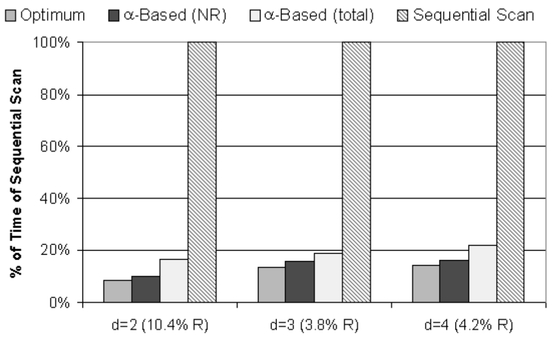
Figure 8(c) shows the effect of having *larger relations* (i.e., N is larger) while keeping the number of buckets in the histogram constant. As we can see, there is a slight increase in the percentage of restarts due to increasingly less accurate histograms (from 2.2% for 10^4 tuples to 5% for 10^6 tuples). However, the time required by our technique decreases relative to the time for a sequential scan, as the gap between the time needed to answer the small range queries issued by our technique and the time to scan sequentially all the relation becomes more pronounced as the number of tuples in the relation increases.

Figure 8(d) shows the effect of increasing the number of tuples requested by the query (i.e., k). The percentage of restarts decreases as k increases, and therefore the total time gets closer to the optimum time. This effect can be explained as follows: when k is large enough, the "granularity" of the queries becomes comparable with that of the histogram buckets, and the estimation error for the number of tuples inside $b \cap reg(q, d)$ decreases, causing fewer restarts. In contrast, the estimation errors for smaller values of k are relatively larger, and consequently, so are the percentage of restarts and the total running time.

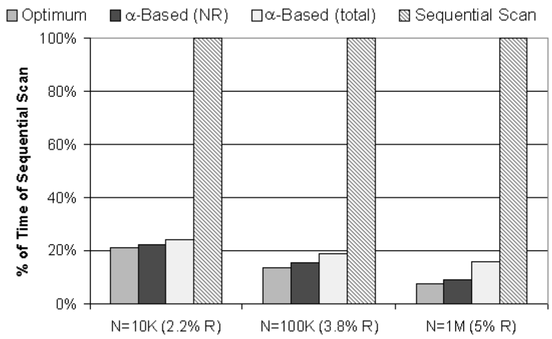
So far, all of the experiments that we have reported use a workload of 500 randomly chosen queries. Figure 8(e) shows results for the *Skewed* workload that we discussed in Section 4.4. The only significant change from the results for the *Random* workload is the slightly worse performance of the highly skewed data set $Z322$. $Z322$ has some tuples that occur with a very large frequency. In particular, one tuple has a frequency of 30,748, out of 100,000 tuples in the data set. Because of the characteristics of the skewed workload, these high frequency tuples will be



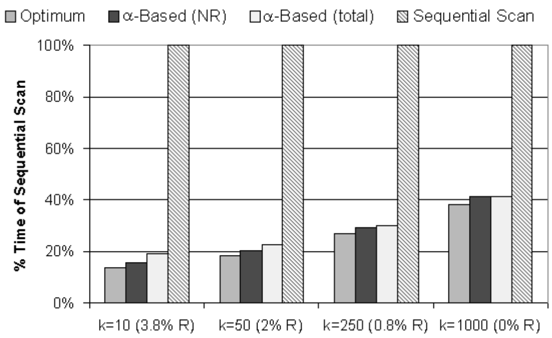
(a) Data skew



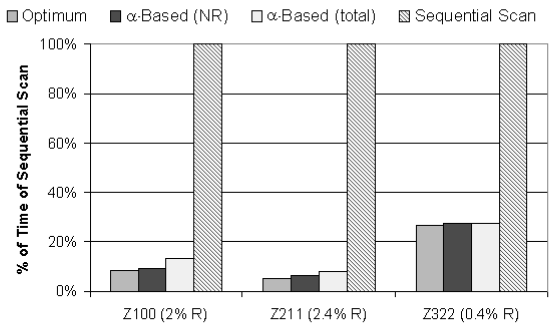
(b) Number of attributes



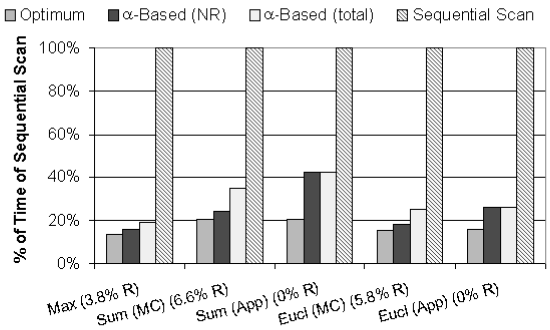
(c) Number of tuples in relation



(d) Number of tuples requested



(e) *Skewed* workload



(f) Distance function

Figure 8: Run time and percentage of restarts for different configurations.

picked as queries repeatedly. In such cases, all the many instances of these "popular" tuples have to be retrieved, resulting in degraded performance.

Effect of the Distance Function

Figure 8(f) shows the results we obtained for different distance functions, namely *Max*, *Eucl*, and *Sum*. The bars labeled *Eucl* (MC) and *Sum* (MC) correspond to the *α-based* technique using the Montecarlo approximation described in Section 3.2.1 for estimating the overlap between a bucket b and a region $reg(q, d)$ around query q. As we discussed, this Montecarlo approximation results in good estimates of $Vol(b \cap reg(q, d))$, but it might be expensive to perform at run time. For efficiency, we consider approximating $reg(q, d)$ for *Eucl* and *Sum* using "simpler" shapes that result in volumes that are as easy to compute as those for *Max*. Specifically, we approximate $reg(q, d)$ by the smallest hyper-rectangle that encloses $reg(q, d)$, or by the largest hyper-rectangle enclosed in $reg(q, d)$.

Example 7: *Suppose we use the Eucl distance function and we want, given a query q, to find the largest hyper-rectangle enclosed by $reg(q, d)$ (Figure 9). Using the monotonicity property (Property 1), the farthest points from q inside a hyper-rectangle of radius r correspond precisely to the vertices of such hyper-rectangle. Therefore, we*

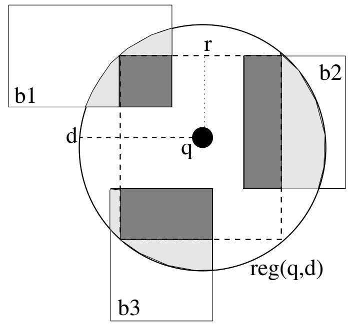
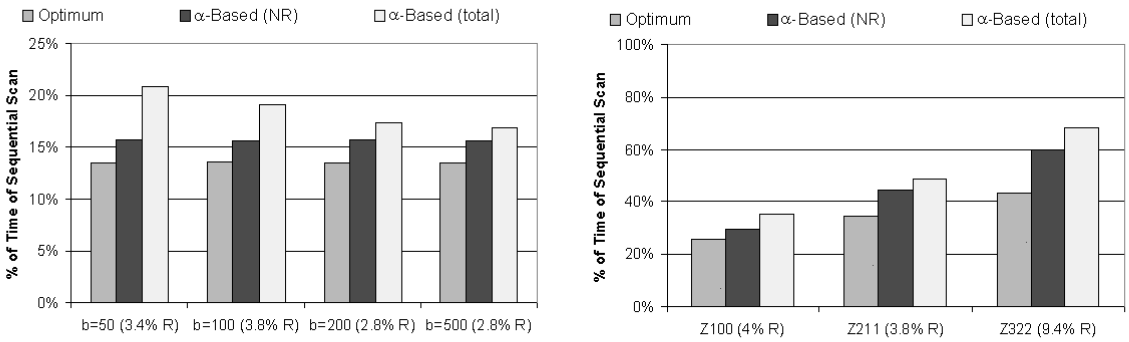


Figure 9: The largest hyper-rectangle enclosed in *reg*(q, d) (*Eucl* distance function).



$(\mathbf{a})$ $(\mathbf{b})$

Figure 10: Run time and restarts varying the number of buckets in the histogram **(a)**, and using one-attribute indexes **(b)**.

calculate, using Eucl, the distance from q to one of the vertices. Such distance equals $\sqrt{\sum_{i=1}^{n} r^2} = r\sqrt{n}$, *where* n *is the number of attributes. This distance must be equal to d, so* $d = r\sqrt{n}$, *and finally, the radius of the enclosed hyper-rectangle is* $r = d/\sqrt{n}$. *Analogously, for the* Sum *distance function we have that the radius is* $r = d/n$. ∎

We tried both alternatives experimentally and decided to use the second one. In effect, using the smallest hyper-rectangle that encloses *reg*(q, d) overestimates both the volume of *reg*(q, d) and the expected number of tuples enclosed in it. This leads to a significant number of restarts, and the total cost of the technique becomes too expensive. In contrast, using the largest hyper-rectangle enclosed in *reg*(q, d) underestimates $Vol(reg(q, d))$ and the number of tuples in it. Therefore, the technique retrieves more tuples than anticipated. However, the time spent in retrieving these extra tuples is generally negligible.

Figure 8(f) shows that, in general, the results for *Max* are the best, followed by those for *Eucl*, and those for *Sum*. As expected, the results for *Sum* and *Eucl* using the Montecarlo approximation (labeled *Sum*(MC) and *Eucl*(MC), respectively) are better than those using the coarser, more efficient enclosed hyper-rectangle approximation (labeled *Sum*(App) and *Eucl*(App), respectively). However, the performance of *Sum*(App) and *Eucl*(App) is still quite good with respect to a sequential scan of the relation, with 0% restarts.

Effect of Histogram Size and Index Configuration

These last two experiments measure the performance of our technique when varying the amount of information we have about the data set. Not surprisingly, Figure 10(a) shows that when we increase the histogram storage, the performance of our technique improves. However, this improvement is not because the buckets capture more information about the data distribution, as the α*-based (NR)* times and the percentage of restarts remain similar. Actually, the benefits come from the cases that need restarts. When we have more buckets, the *dNR* search distance becomes smaller on average, and therefore the time spent to restart a query also decreases.

Figure 10(b) shows how the technique performs when we do not have multiattribute indexes. In this experiment we constructed a one-column unclustered B$^+$-tree index for each attribute mentioned in the query (Section 4.3). We can see that the performance is worse than that for multiattribute indexes. However, even in this case the overall time, except for the highly skewed data set $Z322$, is below 50% of that of a single sequential scan over the relation. Actually, for $Z322$, even the optimal strategy spends over 40% of the time needed for a sequential scan.

6 Related Work

Carey and Kossman [3, 4] present techniques to optimize queries that require only top-k matches when the scoring is done through a *traditional* SQL "Order By" clause. Their technique leverages the fact that when k is relatively small compared to the size of the relation, specialized sorting (or indexing) techniques that can produce the first few values efficiently should be used. However, in order to apply their techniques when the distance function is not based on column values themselves (e.g., as is the case for *Max*, *Eucl*, and *Sum*) we need to first evaluate the distance function for each database object. Only after evaluating the distance for each object are we able to use the techniques in [3, 4]. Hence, these strategies require a preprocessing step to compute the distance function itself involving one sequential scan of all the data.

Donjerkovic and Ramakrishnan [7] propose a probabilistic approach to query optimization for returning the top-k tuples for a given query. Their approach is complementary to ours in that they focus on relations that might be the result of complex queries including joins, for example. In contrast, we focus on single table queries. Also, the ranking condition in [7] involves *a single attribute*, while the core of our contribution is dealing with multiattribute conditions *without assuming independence* among the attributes, for which we exploit multidimensional histograms.

In previous work [5], we presented a family of histogram-based strategies for mapping top-k queries to range selection queries (Section 2.2). Although these strategies seemed promising, no one emerged as consistently the most efficient across arbitrary data sets, as we discussed in the Introduction. In this paper we improve on these techniques and leverage information about the data distribution to define a more robust technique, which we evaluated using a real RDBMS.

There is a large body of work on finding the nearest-neighbors of a multidimensional data point. Given an n-dimensional point p, these techniques retrieve the k objects that are "nearest" to p according to a given distance metric. The state-of-the-art algorithms (e.g., [11]) follow a multi-step approach. Their key step is identifying a set of points A such that p's k nearest neighbors are no further from p than a is, where a is the point in A that is furthest from p. (A more recent paper [15] further refines this idea.) This approach is conceptually similar to the approach that we follow in [5] and also in this paper.

Multidimensional density estimation is an active research field. The main techniques comprise the use of multidimensional histograms [13]. Some variations over histograms include the use of parametric curve fitting techniques inside buckets [10], self-tuning histograms [1], and lately, multidimensional histograms for dealing with real valued attributes [9]. Other multidimensional density estimation techniques are wavelets [12] and fractal dimension concepts [8, 2].

7 Conclusions

In this paper, we have presented a new robust scheme for answering multiattribute top-k queries by mapping them to relational selection queries. We have reported the first evaluation of the performance of top-k mapping techniques over a commercial RDBMS, namely Microsoft's SQL Server 7.0. Our experiments clearly demonstrate that considering mapping top-k queries to multiattribute range queries that "cover" one or more histogram buckets only partially, and capturing skew within a histogram bucket, have been key ingredients in reducing the probability of restarts while ensuring that the required top-k matches are returned. Our new techniques are robust, and perform significantly better than existing strategies requiring at least one sequential scan of the data sets.

References

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *Proceedings of the 1999 ACM International Conference on Management of Data (SIGMOD'99)*, June 1999.

[2] A. Belussi and C. Faloutsos. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In *Proceedings of the Twenty-first International Conference on Very Large Databases (VLDB'95)*, Sept. 1995.

[3] M. J. Carey and D. Kossmann. On saying "Enough Already!" in SQL. In *Proceedings of the 1997 ACM International Conference on Management of Data (SIGMOD'97)*, May 1997.

[4] M. J. Carey and D. Kossmann. Reducing the braking distance of an SQL query engine. In *Proceedings of the Twenty-fourth International Conference on Very Large Databases (VLDB'98)*, Aug. 1998.

[5] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *Proceedings of the Twenty-fifth International Conference on Very Large Databases (VLDB'99)*, Sept. 1999.

[6] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the Twenty-third International Conference on Very Large Databases (VLDB'97)*, Aug. 1997.

[7] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top N queries. In *Proceedings of the Twenty-fifth International Conference on Very Large Databases (VLDB'99)*, Sept. 1999.

[8] C. Faloutsos and I. Kamel. Relaxing the uniformity and independence assumptions using the concept of fractal dimension. *Journal of Computer and System Sciences*, 1997.

[9] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. In *Proceedings of the 2000 ACM International Conference on Management of Data (SIGMOD'00)*, 2000.

[10] A. C. Konig and G. Weikmun. Combining histograms and parametric curve fitting for feedback-driven query result-size estimation. In *Proceedings of the Twenty-fifth International Conference on Very Large Databases (VLDB'99)*, Sept. 1999.

[11] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast nearest neighbor search in medical image databases. In *Proceedings of the Twenty-second International Conference on Very Large Databases (VLDB'96)*, Sept. 1996.

[12] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proceedings of the 1998 ACM International Conference on Management of Data (SIGMOD'98)*, June 1998.

[13] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the Twenty-third International Conference on Very Large Databases (VLDB'97)*, Aug. 1997.

[14] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of the 1996 ACM International Conference on Management of Data (SIGMOD'96)*, June 1996.

[15] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *Proceedings of the 1998 ACM International Conference on Management of Data (SIGMOD'98)*, June 1998.

[16] S. A. William, H. Press, B. P. Flannery, and W. T. Vetterling. *Numerical recipes in C: The art of scientific computing*. Cambridge University Press, 1993.

[17] G. K. Zipf. *Human behaviour and the principle of least effort*. Addison-Wesley, 1949.

A Effect of Histogram Quality

Throughout our presentation, an implicit assumption was that we had available good quality histograms that manage to correctly "bucketize" regions in the relation that exhibit reasonably uniform tuple density. Through the introduction of parameter α (Section 3.2.2) we have showed that we can model moderate intra-bucket skews, and the resulting technique is efficient and does not cause restarts for the overwhelming majority of the queries. Understandably, our histogram-based techniques will not perform as well when the underlying histograms are not as well behaved, as we discuss next.

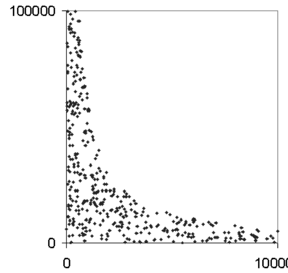


Figure 11: Typical histogram bucket generated with *MHIST* for our synthetic data set.

Consider the following synthetic data set with 10^5 tuples and three attributes A_1, A_2, and A_3. The first attribute, A_1, follows a Zipfian distribution with $Z = 1$ and has 100 distinct values. Attributes A_2 and A_3 are independent of A_1, but they are strongly correlated. Specifically, $A_2 \cdot A_3 \leq (\frac{1}{4} \cdot 10^5)^2$. The possible values for attributes A_2 and A_3 come from the integer range $[1 \ldots 10^5)$. We applied *MHIST*-2 (with 100 buckets) to this data set. In the first 98 steps, *MHIST* chose attribute A_1 as the dimension to split. Only in the last two steps were other dimensions chosen for splitting. This behavior is explained by the characteristics of the marginal frequency for the different attributes. While for A_2 and A_3 the marginal frequency rarely exceeds 10 (in fact, it is often below 3), the marginal frequencies for attribute A_1 are between 200 and 19,000. Therefore, the *frequency gaps* are much more likely to be higher on attribute A_1 than on A_2 or A_3, and therefore *MHIST* splits almost exclusively along this dimension. Figure 11 shows the distribution of tuples (projected over attributes A_2 and A_3) in a typical bucket obtained for this data set. As we can see, such a bucket has regions of very high tuple density, while other large regions are virtually empty. We ran our techniques over this intentionally bad synthetic data set. Not surprisingly, our technique does not perform as well as for the case when we do have good quality histograms available (Section 5). We obtained a high percentage of restarts (around 60%) for the *Random* workload, and a larger than needed number of tuples retrieved for the *Skewed* workload.

The focus of this paper is not on building high quality histograms for arbitrary data sets. However, our strategies critically depend on the existence of such histograms. Fortunately, the area of multidimensional histogram construction has attracted significant attention in the research community lately. Very recently, [9] proposed a new histogram technique called *GENHIST*, designed to approximate the density of multidimensional data sets with real attributes, finding buckets that may overlap and whose size is based on the local density of the data. This approach for building histograms appears promising for dealing with the kind of data sets described in this section, and it can be easily incorporated in our technique. Further research is needed in this direction and is subject for future work.