Evaluating Top-k Selection Queries

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Luis Gravano
Columbia University
gravano@cs.columbia.edu

Abstract

In many applications, users specify target values for certain attributes, without requiring exact matches to these values in return. Instead, the result to such queries is typically a rank of the "top k" tuples that best match the given attribute values. In this paper, we study the advantages and limitations of processing a top-k query by translating it into a single range query that traditional relational DBMSs can process efficiently. In particular, we study how to determine a range query to evaluate a top-k query by exploiting the statistics available to a relational DBMS, and the impact of the quality of these statistics on the retrieval efficiency of the resulting scheme.

1 Introduction

Internet Search engines rank the objects in the results of selection queries according to how well these objects match the original selection condition. For such engines, query results are not flat sets of objects that match a given condition. Instead, query results are ranked starting from the top object for the query at hand. Given a query consisting of a set of words, a search engine returns the matching documents sorted according to how well they match the query. For decades, the information retrieval field has studied how to rank text documents for a query both efficiently and effectively [13]. In contrast, much less attention has been devoted to supporting such *top-k queries* over relational databases.

As the following example illustrates, top-k queries arise naturally in many applications where the data is exact, as in a traditional relational database, but where users are flexible and willing to accept non-exact matches that are close to their specification. The answer to such a query is a ranked set of the k tuples in the database that "best" match the selection condition.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

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. ∎

Unfortunately, despite the conceptual simplicity of top-k queries and the expected performance payoff, they are not yet supported by today's relational database systems. This support would free applications and end-users from having to add this functionality in their client code. To provide such support efficiently, we need processing techniques that do not involve full sequential scans of the underlying relations. The challenge in providing this functionality is that the database system needs to handle efficiently top-k queries *for a wide variety of scoring functions*. In effect, these scoring functions might change by user, and they might also vary by application, or by database. It is also important that we are able to process such top-k queries with as few extensions to existing query engines as possible, since today's relational systems are significantly complex and performance sensitive.

As in the case of processing traditional selection queries, one must consider the problem of execution as well as optimization of top-k queries. We assume that the execution engine is a traditional relational engine that supports single as well as possibly multidimensional indexes. Therefore, the key challenge is to *augment the optimization phase* such that top-k selection queries may be compiled into an execution plan that can leverage the existing data structures (i.e., indexes) and statistics (e.g., histograms) that a database system maintains. Simply put, we need to develop new techniques that make it possible to map a top-k query into a traditional selection query. It is also important that any such technique preserves the following two properties: (1) it handles a variety of scoring functions for computing the top-k tuples for a query, and (2) it guarantees that there are no false dismissals (i.e., we never miss any of the top-k tuples for the given query).

In this paper, we undertake a comprehensive study of the problem of mapping top-k queries into execution plans that use traditional selection queries. In particular, we use the database histograms to map a top-k query to a suitable range that encapsulates k best matches for the query. In particular, we study the sensitivity of the mapping algorithms to the following parameters: types of histograms available and their memory budgets, scoring functions, data distribution, and number of query attributes.

The rest of the paper is organized as follows. Section 2 formally defines the problem of querying for top-k matches. Section 3 discusses related work. Section 4 is the core of the paper, and outlines the techniques that form the basis of our approach. Finally, Section 6 presents an experimental evaluation of our approach, using the experimental setting of Section 5.

2 Query Model

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 reflects how closely each tuple matches the given query. This section defines our query model precisely.

Consider a relation R with attributes $A_1, \ldots, A_n$. A top-k query over R simply specifies target values for the attributes in R. Thus, a query is an assignment of values $v_1, \ldots, v_n$ to the attributes $A_1, \ldots, A_n$ of R. In this paper, we will focus on top-k queries on continuous attributes (e.g., `age`, `salary`). Without loss of generality, we will also assume that the values of these attributes are normalized to be real numbers between 0 and 1.

Example 2 : Consider a relation S with two attributes, A_1 and A_2. These attributes have real values that range between 0 and 1. An example of top-10 query over this relation is $q = (0.4, 0.3)$. Such a query asks for the 10 tuples in S that are the closest to the $(0.4, 0.3)$ point, for some definition of proximity, as we discuss below. ∎

Given a top-k query q, the database system with relation R uses some *scoring function Score* to determine how closely each tuple in R matches the target values $v_1, \ldots, v_n$ specified in query q. Given a tuple t and a query q, we assume that $Score(q,t)$ is a real number that ranges between 0 and 1. In this paper, we focus on three important scoring functions, namely *Min*, *Euclidean*, and *Sum*.

Definition 1: *Consider a relation $R = (A_1, \ldots, A_n)$. $A_1, \ldots, A_n$ are real-valued attributes ranging between 0 and 1. Then, given a query $q = (q_1, \ldots, q_n)$ and a tuple $t = (t_1, \ldots, t_n)$ from R, we define the* score *of t for q using any of the following three scoring functions:*

$$\begin{aligned} Min(q,t) &= \min_{i=1}^{n}\{1 - |q_i - t_i|\} \\ Euclidean(q,t) &= 1 - \sqrt{\sum_{i=1}^{n} \frac{(q_i - t_i)^2}{n}} \\ Sum(q,t) &= 1 - \sum_{i=1}^{n} \frac{|q_i - t_i|}{n} \end{aligned}$$

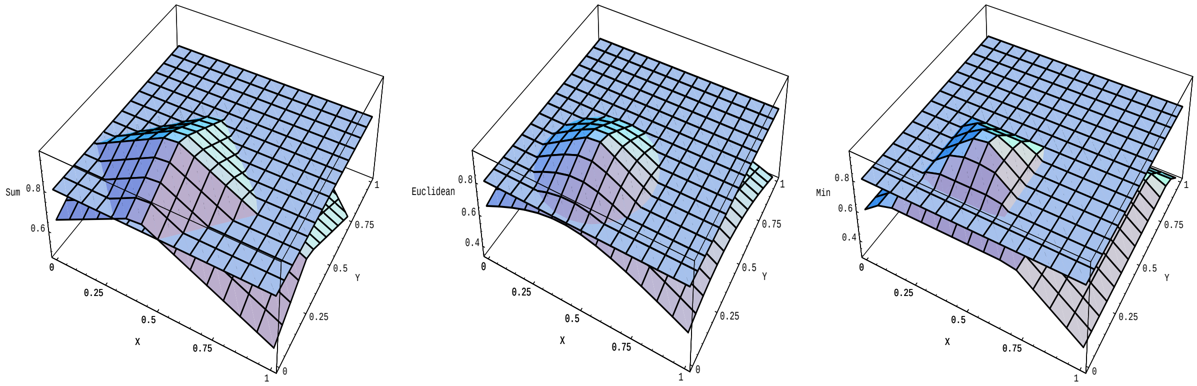
Example 3 : Consider a tuple $t = (0.3, 0.8)$ in our sample database S from Example 2, and query $q = (0.4, 0.3)$. Then, t will then have a score of $Min(q,t) = \min\{1 - |0.3 - 0.4|, 1 - |0.8 - 0.3|\} = 0.5$ for the *Min* scoring function, a score of $Euclidean(q,t) = 1 - \sqrt{\frac{|0.3-0.4|^2}{2} + \frac{|0.8-0.3|^2}{2}} = 0.64$ for the *Euclidean* scoring function, and a score of $Sum(q,t) = 1 - (\frac{|0.3-0.4|}{2} + \frac{|0.8-0.3|}{2}) = 0.7$ for the *Sum* scoring function. ∎

Figure 1(c) shows the distribution of scores for the *Min* scoring function and query $q = (0.4, 0.3)$. The horizontal plane in the figure consists of the tuples with $z = 0.8$, so what "emerges" above this plane are those tuples with score 0.8 or higher. Note that the tuples with score 0.8 or higher for q are enclosed in a box around q. In contrast, the tuples with score 0.8 or higher for the *Euclidean* scoring function (Figure 1(b)) are enclosed in a circle around q. Finally, the top tuples according to the *Sum* scoring function lie within a rotated box around q (Figure 1(a)). This difference in the shape of the region enclosing the top tuples for the query will have crucial implications on query processing, as we will discuss in Section 4.

A simple variation of the definition of the scoring functions above results from letting the different attributes have different weights. In general, the *Min*, *Euclidean*, and *Sum* functions that we use in this paper are just a few of many possible scoring functions. Our strategy for processing top-k queries can be adapted to handle a wide variety of such functions, as we will discuss. The key property that we ask from scoring functions is as follows:

Property 1: Monotonicity of Scoring Functions: *Consider a relation R and a scoring function Score defined over it. Let $q = (v_1, \ldots, v_n)$ be a top-k query over R, and let $t = (t_1, \ldots, t_n)$ and $t' = (t'_1, \ldots, t'_n)$ be two tuples in R such that $|t'_i - q_i| \le |t_i - q_i|$ for $i = 1, \ldots, n$. (In other words, t' is at least as close to q as t for all attributes.) Then, $Score(q,t') \ge Score(q,t)$.*

Intuitively, this property of scoring functions implies that if a tuple t' is closer, along each attribute,



(**a**) (**b**) (**c**)

Figure 1: The scores (z axis) for query $q = (0.4, 0.3)$ for the different (x, y) pairs and scoring functions *Sum* **(a)**, *Euclidean* **(b)**, and *Min* **(c)**.

to the query values than some other tuple t is, then, the score that t' gets for the query cannot be worse than that of t. Fortunately, all interesting scoring functions that we could think of satisfy our monotonicity assumptions. In particular, the *Euclidean*, *Min*, and *Sum* scoring functions that we defined above satisfy this property.

A possible SQL-like notation for expressing top-k queries is as follows [3]:

```
SELECT * FROM R
WHERE A1=v1 AND ...  AND An=vn
ORDER k BY Score
```

The distinguishing feature of the query model is in the `ORDER BY` clause. This clause indicates that we are interested in only the k answers that best match the given `WHERE` clause, according to the *Score* function. Section 4 discusses how we will evaluate top-k queries for different definitions of the *Score* function.

3 Related Work

Motro [9] emphasized the need to support approximate and ranked matches in a database query language. He extended the language `Quel` to distinguish between exact and vague predicates. He also suggested a composite scoring function to rank each answer. Motro's work led to further development of the idea of query relaxation that weakens a given user query to provide approximate matches using additional metadata (e.g., concept hierarchies). The querying model for top-k queries that we use in this paper is consistent with Motro's definitions. Our key focus is on exploring opportunities and limitations of efficiently mapping top-k queries into traditional relational queries.

Recently, Carey and Kossman [1, 2] presented techniques to optimize queries that require only top-k matches. 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 scoring function is not based on column values themselves (e.g., as is the case for *Min*, *Euclidean*, and *Sum* as defined in Section 2), we need to first evaluate the scoring function for each database object. Thus, when a query requests the top-k values according to a scoring function like *Min*, their technique would need to first evaluate the *Min* score for every data object. Only after evaluating the score for each object are we able to use the techniques in [1, 2]. Hence, these strategies require a preprocessing step to compute the scoring function itself involving one sequential scan of all the data. In contrast, in this paper we explore techniques that avoid accessing the entire data set.

In [4, 5], Fagin addresses the problem of finding top-k matches for a user query q involving several *multimedia* attributes. Each of these attributes (e.g., an image attribute) is assumed to have a native sub-system that answers top-k queries involving only the corresponding attribute. In the first phase of Fagin's A_0 algorithm, the query processing system obtains a stream L_i of top matches for condition c_i on attribute A_i from the corresponding sub-system. When there are at least k objects in the intersection of all the single-attribute streams L_i, the system is guaranteed to have already accessed k top objects for query q. (These top objects are not necessarily in the intersection of the streams.) The second phase of algorithm A_0 computes the score of each of the retrieved objects, and returns the best k objects. In Section 4.3, we present an adaptation of Fagin's strategy to the case when the top-k query is issued against a relational database system. In [3], we presented an algorithm for processing queries over a multimedia database. Our query model built on Fagin's to also include Boolean conditions to the top-k component of the multimedia queries.

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., [7]) 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 [14] further refines this idea.) This approach is conceptually similar to the approach that we follow in this paper (and also in [3]), where we first find a suitable score S, and then we use it to build a relational query that will return the top-k matches for the original query. Our focus in this paper is to study the practicality and limitations of using the information in the *histograms* kept by a relational system for query processing. In contrast, the nearest-neighbor algorithms mentioned above use the data values themselves to identify a cut-off "score."

Finally, references [6, 8] study how to merge and reconcile top-k query results obtained from distributed databases when the databases use arbitrary, undisclosed scoring algorithms.

4 Mapping a Top-k Query into a Traditional Selection Query

This section shows how to map a top-k query q into a relational selection query C_q that any traditional RDBMS can execute. Our goal is to obtain k tuples from relation R that are the best tuples for q according to a scoring function *Score*. Our query processing strategy consists of the following steps:

1. Use statistics on relation R to find a search score S_q (Section 4.1).
2. Build a selection query C_q to retrieve all tuples in R with score S_q or higher for q (Section 4.2).
3. Evaluate C_q over R.
4. Compute $\mathit{Score}(q,t)$ for every tuple t in the answer for C_q.
5. If there are at least k tuples t in the result for C_q with $\mathit{Score}(q,t) \geq S_q$, then output k tuples with the highest scores. Otherwise, choose a lower value for S_q and *restart* the process.

Section 4.3 introduces a related mapping strategy that does *not* follow the five steps above, and is an adaptation of Fagin's A_0 algorithm (Section 3).

4.1 Choice of Search Score S_q

The key step for evaluating a top-k query q is determining score S_q: our algorithm retrieves all tuples t such that $\mathit{Score}(q,t) \geq S_q$. If there are at least k such tuples, then our algorithm above succeeds in finding the top k matches for q. Otherwise, our choice of S_q is too high, and hence the query needs to be *restarted* with a lower value for S_q. Consequently, we should choose a value of S_q that is not too low, so that we do not retrieve too many candidate tuples from the database, but that is not too high either, so that we can obtain the top-k tuples without restarting the query.

Our choice of S_q will be guided by the statistics that the query processor keeps about relation R. In particular, we will assume that we have an n-dimensional histogram H that describes the distribution of values of R. We discuss this issue further in Section 5.2. Until then, we assume that H consists of a series of non-overlapping *buckets*. Each bucket has associated with it an *n-rectangle* $[a_1, b_1] \times \ldots \times [a_n, b_n]$, and stores the number of tuples in R that lie within the n-rectangle, together with other information.

For efficiency, our choice of S_q will be based on histogram H, and not on the underlying relation R itself. More specifically, we choose S_q as follows:

a. Create (*conceptually*) a small, "synthetic" relation R', consistent with histogram H. R' has one distinct tuple for each bucket in H, with as many instances as the frequency of the corresponding bucket.

b. Compute $\mathit{Score}(q,t)$ for every tuple t in R'.

c. Let T be the set of the top-k tuples in R' for q. Output $S_q = \min_{t \in T} \mathit{Score}(q,t)$.

We can conceptually build synthetic relation R' in many different ways. We will study two "extreme" query processing strategies that result from two possible definitions of R'.

The first query processing strategy, *NoRestarts*, results in a search score S_q that is low enough to guarantee that no restarts are ever needed as long as histograms are kept up to date. In other words, Step (5) above always finishes successfully, without ever having to reduce S_q and restart the process. For this, the *NoRestarts* strategy defines R' in a "pessimistic" way: given a histogram bucket b, the corresponding tuple t_b that represents b in R' will be as bad for query q as possible. More formally, t_b is a tuple in b's n-rectangle with the following property:

$$\mathit{Score}(q, t_b) = \min_{t \in T_b} \mathit{Score}(q,t)$$

where T_b is the set of all potential tuples in the n-rectangle associated with bucket b.

Example 4: Consider our example relation S, with two attributes A_1 and A_2, query $q = (0.4, 0.3)$, and the 2-dimensional histogram H shown in Figure 2(a). Histogram H has three buckets, b_1, b_2, and b_3. Relation S has 40 tuples in bucket b_1, 5 tuples in bucket b_2, and 55 tuples in bucket b_3. As explained above, the *NoRestarts* strategy will "build" relation S' based on H by assuming that the tuple distribution in S is

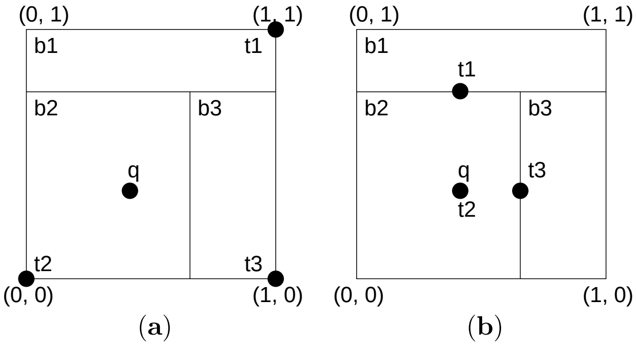


Figure 2: A 3-bucket histogram H and the choice of tuples representing each bucket that strategies *NoRestarts* **(a)** and *Restarts* **(b)** make for query q.

as "bad" as possible for query q. So, relation S' will consist of three tuples (one for each bucket in H) t_1, t_2, and t_3, which are as far from q as their corresponding bucket boundaries permit. Tuple t_1 will have a frequency of 40, t_2 will have a frequency of 5, and t_3 will have a frequency of 55. Assume that the user who issued query q wants to use the *Min* scoring function to find the top 10 tuples for q. Since $Min(q, t_1) = 0.3$, $Min(q, t_2) = 0.6$, and $Min(q, t_3) = 0.4$, to get 10 tuple instances we need the top tuple, t_2 (frequency 5), and t_3 (frequency 55). Consequently, the search score S_q will be $Min(q, t_3) = 0.4$. From the way we built S', it follows that the original relation S is guaranteed to contain at least 10 tuples with score $S_q = 0.4$ or higher for query q. Then, if we retrieve all of the tuples with that score or higher, we will obtain a superset of the set of top-k tuples for q. ▮

Lemma 1: *Let q be a top-k query over a relation R. Let S_q be the search score computed by strategy NoRestarts for q. Then, there are at least k tuples t in R such that $Score(q, t) \geq S_q$.*

The second query processing strategy, *Restarts*, results in a search score S_q that is highest among those search scores that *might* result in no restarts. This strategy defines R' in an "optimistic" way: given a histogram bucket b, the corresponding tuple t_b that represents t_b in R' will be as good for query q as possible. More formally, t_b is a tuple in b's n-rectangle with the following property:

$$Score(t_b, q) = \max_{t \in T_b} Score(q, t)$$

where T_b is the set of all potential tuples in the n-rectangle associated with bucket b.

Example 4: (cont.) The *Restarts* strategy will now build relation S' based on H by assuming that the tuple distribution in S is as "good" as possible for query q (Figure 2(b)). So, relation S' will consist of three tuples (one per bucket in H) t_1, t_2, and t_3, which

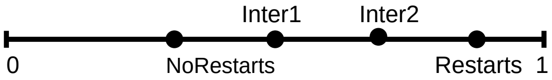


Figure 3: The four strategies for computing the search score S_q.

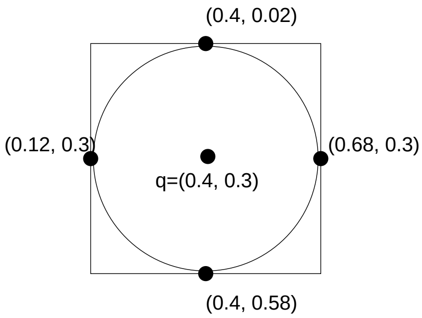


Figure 4: The circle around query $q = (0.4, 0.3)$ contains all of the tuples with *Euclidean* score of 0.8 or higher for q.

are as close to q as their corresponding bucket boundaries permit. In particular, tuple t_2 will be defined as q proper, with frequency 5, since its corresponding bucket (i.e., b_2) has 5 tuples in it. After defining the bucket representatives t_1, t_2, and t_3, we proceed as in the *NoRestarts* strategy to sort the tuples on their score for q. For *Min*, we pick tuples t_2 and t_3, and define S_q as $Min(q, t_3)$. This time it is indeed possible for fewer than k tuples in the original table S to have a score of S_q or higher for q, so restarts are possible. ▮

The S_q score that *Restarts* computes is the highest score that might result in no restarts in Step (5) of the algorithm above. In other words, using a value for S_q that is higher than that of the *Restarts* strategy will *always* result in restarts. In practice, as we will see in Section 6, the *Restarts* strategy results in restarts in virtually all cases, hence its name.

Lemma 2: *Let q be a top-k query over a relation R. Let S_q be the search score computed by strategy Restarts for q. Then, there are fewer than k tuples t in R such that $Score(q, t) > S_q$.*

In addition to the two extreme score-selection strategies *NoRestarts* and *Restarts*, we will study two other intermediate strategies, *Inter1* and *Inter2* (Figure 3). Given a query q, let S_q be the search score selected by *NoRestarts* for q, and let S'_q be the corresponding score selected by *Restarts*. Then, the *Inter1* strategy will choose score $\frac{2 \times S_q + S'_q}{3}$, while the *Inter2* strategy will choose a higher score of $\frac{S_q + 2 \times S'_q}{3}$. As our experiments will show, *Inter1* and *Inter2* are often the best strategies that we can follow in terms of the efficiency of the resulting techniques.

4.2 Choice of Selection Query C_q

Once we have determined the search score S_q (Section 4.1), the algorithm in Section 4 uses a query C_q to retrieve all tuples t such that $Score(q,t) \geq S_q$, where q is the original top-k query, and *Score* is the scoring function being used. In this section we describe how to define query C_q.

Ideally, we would like to ask our database system to return exactly those tuples t such that $Score(q,t) \geq S_q$. Unfortunately, indexing structures in relational DBMSs do not natively support this kind of predicates, as discussed in Section 3. Our approach is to build C_q as a simple selection condition defining an n-rectangle. In other words, we define C_q as a query of the form:

```
SELECT * FROM R
WHERE (a1<=A1<=b1) AND ...  AND (an<=An<=bn)
```

The n-rectangle $[a_1, b_1] \times \ldots \times [a_n, b_n]$ in C_q should tightly enclose all tuples t in R with $Score(q,t) \geq S_q$.

Example 5 : Consider our example query $q = (0.4, 0.3)$ over relation S, with *Euclidean* as the scoring function. Suppose that our search score S_q is 0.8, as computed by any of the strategies in Section 4.1. Each tuple t with $Euclidean(q,t) \geq 0.8$ lies in the circle around q that is shown in Figure 4. Then, the tightest n-rectangle that encloses that circle is $[0.12, 0.68] \times [0.02, 0.58]$. Hence, the final SQL query C_q is:

```
SELECT * FROM S
WHERE (0.12<=A1<=0.68) AND (0.02<=A2<=0.58)
```

∎

Given a search score S_q, the n-rectangle $[a_1, b_1] \times \ldots \times [a_n, b_n]$ that determines C_q follows directly from the scoring function used, the search score S_q, and the query q.

Example 5: (cont.) Let us assume that the search score for our query $q = (0.4, 0.3)$ is $S_q = 0.8$, as above. We calculate the 2-rectangle that encloses all tuples with 0.8 score or higher by focusing on one attribute at a time. First, consider a tuple $r = (t_1, 0.3)$ that has the same attribute values as query q in all attributes except for maybe attribute A_1. We will compute the range of values that t_1 can have while $Euclidean(q,r) \geq 0.8$. In effect, $Euclidean(q,r) = Euclidean((0.4, 0.3), (t_1, 0.3)) = 1 - \sqrt{\frac{(t_1 - 0.4)^2}{2}}$. Consequently, $Euclidean(q,r) \geq 0.8$ if and only if $0.12 \leq t_1 \leq 0.68$. Hence, the range of values that attribute A_1 can take is $[a_1, b_1] = [0.12, 0.68]$. Analogously for attribute A_2, $[a_2, b_2] = [0.02, 0.58]$. Putting both pieces together, the final 2-rectangle that encloses all tuples with score 0.8 or higher for q is $[0.12, 0.68] \times [0.02, 0.58]$ (Figure 4). ∎

Score	a_i'	b_i'
Min	$q_i - (1.0 - S_q)$	$q_i + (1.0 - S_q)$
Sum	$q_i - (1.0 - S_q) \cdot n$	$q_i + (1.0 - S_q) \cdot n$
Euclidean	$q_i - (1.0 - S_q) \cdot \sqrt{n}$	$q_i + (1.0 - S_q) \cdot \sqrt{n}$

Table 1: The n-rectangle $[a_1, b_1] \times \ldots \times [a_n, b_n]$ for C_q's selection condition and search score S_q, for different scoring functions, where $a_i = \max\{0, a_i'\}$ and $b_i = \min\{1, b_i'\}$.

Table 1 summarizes how to compute the n-rectangle $[a_1, b_1] \times \ldots \times [a_n, b_n]$ for the three scoring functions from Section 2. The *Min* scoring function presents an interesting property: the region to be enclosed by the n-rectangle is already an n-rectangle. (See Figure 1(c).) Consequently, the query C_q that is generated for *Min* for query q and its associated search score S_q will retrieve only tuples with a score of S_q or higher. This property will result in efficient executions of top-k queries for *Min*, as we will see. Unfortunately, this property does not hold for the *Sum* and *Euclidean* scoring functions (Figures 1(a) and (b)).

4.3 An Alternative Mapping Strategy

This section adapts Fagin's A_0 algorithm (Section 3) to produce a new technique for mapping a top-k query into a traditional relational query. Unlike the Section 4.2 strategies, the selection query resulting from this new mapping is a disjunction, not a conjunction.

Our goal is, again, to build a "one-shot" relational query that avoids restarts whenever possible. We proceed as in strategy *NoRestarts* (Section 4.1) to build a "database" with one tuple representing each bucket in the available n-dimensional histogram. We find the top "tuples" as in the *NoRestarts* strategy. We then compute an n-rectangle $F = [a_1, b_1] \times \ldots [a_n, b_n]$ that encloses these top tuples tightly, and that has been extended so that it is "symmetric" with respect to the given query q. (In other words, $a_i \leq q_i \leq b_i$ and $b_i - q_i = q_i - a_i$, for $i = 1, \ldots, n$.) The tuples matching range $[a_i, b_i]$ are the top tuples for q along attribute A_i. The selection query consists of the disjunction of the $a_i \leq A_i \leq b_i$ conditions. By retrieving all tuples that match *at least one* of these conditions, we retrieve the top tuples for each of the individual attributes. Furthermore, from the way we constructed F, there will be at least k tuples matching all n conditions. As with the original A_0 algorithm, we compute the score for all the one-dimensional matches. The k retrieved tuples having the highest score for q are the final answer to the original top-k query. The correctness of this algorithm follows from that of algorithm A_0 [4]. Due to space constraints, we do not discuss this algorithm any further in this paper.

5 Experimental Setting

We now describe the data sets, histograms, and metrics for the experiments of Section 6.

5.1 Data Sets

Our experiments use a real-world data set as well as synthetic data. The real-world data set is a fragment of US Census Bureau data, and was obtained from the University of California, Irvine archive of machine-learning databases (`ftp://ftp.ics.uci.edu/pub/-machine-learning-databases`). The data set has 45,000 rows. Each row is a record for an individual, with 14 attributes. We picked four continuous attributes that were especially well suited for our top-k query model: `age`, `wage`, `education level`, and `hours of work per week`. We also scaled down the attribute values so that the resulting values ranged between 0 and 1, to simplify our experimental setting. We refer to this database as the *Census database*.

In addition to the Census database, we generated a number of synthetic databases with different data distributions. For this, we wrote a seed program that is capable of generating one-dimensional Zipfian distribution [15] 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, our generation program is parameterized by (1) a vector of n Z values (one for each attribute), $Z_n =< z_1, \ldots, z_n >$; (2) the number of tuples to be generated, N. We created 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 databases of 100,000 records with n = 2, 3, and 4 attributes. The domain of each attribute is the real numbers between 0 and 1, with a spacing of 0.00001 between attribute values. We varied the Zipfian vectors in the generation of the databases so we obtained databases with a spectrum of skews. More specifically, Section 6 reports experiments for three families of databases, $Z10$, $Z21$, and $Z32$. $Z10$, $Z21$, and $Z32$ represent the skew of databases built using Zipfian vectors $< 1, 0, \ldots, 0 >$, $< 2, 1, \ldots, 1 >$, and $< 3, 2, \ldots, 2 >$, respectively. Table 2 summarizes the synthetic databases for which we report experiments in the next section.

Data Skew	n		
	2	3	4
$Z10$	100,000	100,000	100,000
$Z21$	27,022	52,554	66,426
$Z32$	739	2878	7034

Table 2: The number of distinct tuple values for different data skews and number of attributes n.

5.2 Histograms

As outlined above, we map a top-k query over a table R into a relational selection query. To do this mapping, we exploit the statistics (e.g., histograms) kept by the relational DBMS where relation R resides. One of our goals in this paper is to study the effect on our mapping of the different n-dimensional histogram structures proposed in the literature. These structures rely on an underlying strategy for building one-dimensional histograms. In this paper we focus on the *AVI*, *PHASED*, and *MHIST-p* n-dimensional techniques, with *MAXDIFF* as the underlying one-dimensional strategy [11, 12]. Below we briefly describe these structures. We refer the reader to [11, 12] for a detailed discussion.

Constructing a *MAXDIFF* histogram on an attribute of a relation is logically a two-step process. First, the data values are sorted and, for each distinct value, its frequency of occurrence is calculated. Let the sorted values be $v_1, \ldots, v_n$ with corresponding frequencies $f_1, \ldots, f_n$. We can then define *frequency-gap*$(i) = |f_{i+1} - f_i|$. This function records the difference in frequency of attribute values v_i and v_{i+1}. The bucket boundaries are placed at those attribute values that correspond to the highest values of the *frequency-gap* function. The *MAXDIFF* histogram structure has been shown to have a good trade-off between accuracy and building cost [12]. For the experiments that we report in the next section, we have implemented n-dimensional variants of *MAXDIFF* histograms using the *AVI*, *PHASED*, and *MHIST-p* techniques, as described in [11].

The *AVI* technique for constructing an n-dimensional histogram is to simply assume statistical independence of the one-dimensional attributes. Thus, to determine the fraction of data in an n-dimensional bucket, we multiply the fraction of the data in each one-dimensional projection of the bucket.

The *PHASED* technique for constructing an n-dimensional histogram consists of n steps. In the first step, one of the dimensions is used to partition the dataset into k_1 buckets. In the j^{th} step, each of the buckets obtained at the end of the previous step is divided into k_j buckets along one of the unused dimensions. The order in which dimensions are chosen is determined prior to doing any of the partitioning.

For each dimension (attribute), we compute the variance in the frequency of values on that dimension. We then choose the attributes for partitioning the buckets in descending order of their variance. This order reflects the criticality for separating the values in buckets. This technique for constructing n-dimensional histogram was first used in [10] in the context of equidepth histogram structures.

The *MHIST-p* technique for constructing an n-dimensional histogram is an adaptation of the *PHASED* approach. More specifically, during the j^{th} step (see the description of *PHASED* above), we determine the bucket in most need of partitioning, and we partition it along the attribute that exhibits the highest variance in frequency within the bucket. The factor p designates the number of buckets into which each bucket is split at every step.

The performance of our mapping techniques (Section 4) depends on the accuracy of the available histograms. The accuracy of a histogram depends in turn on the technique with which it was generated, and on the amount of memory that has been allocated for it. In our experiments, in addition to trying several histogram structures, we also study the effect of varying memory on the accuracy of histograms. We assume throughout that histograms are kept up to date with the data. If histograms are not up to date, then the performance of our techniques might decrease. However, the correctness of the answers produced will remain unaffected, at the expense of a potentially higher number of restarts (Section 4).

5.3 Measuring the Efficiency of the Query Execution Strategies

A top-k query q will typically involve several attributes. We might have indexes available for a number of combinations of the query attributes, and the efficiency of processing the query will be greatly affected by the particular index configuration available. We focus on two configurations: (a) a single-column index exists for every attribute mentioned in the query; or (b) a single n-column index exists, covering all attributes mentioned in the query.

Whenever an n-dimensional index is present, we retrieve exactly as many index "entries" as there are tuples in the n-rectangle defining query C_q, as described in Section 4.2, followed by the actual retrieval of the k top tuples for q. (The index entries provide all the information that we need to decide which k tuples are the ones with the highest score for q.) Alternatively, when only one-dimensional indexes are available, we can intersect one or more indexes to determine the data tuples to be retrieved. When all necessary single-column indexes are present, this strategy results in no redundant retrieval of data tuples, as in the case when an n-dimensional index is available. However, unlike the case with n-dimensional indexes, we must now pay the overhead of the index intersection. The cost of the index intersection can be traded off against the cost of retrieving redundant data tuples (i.e., data tuples that do not belong to the n-rectangle of Section 4.2).

For each top-k query q, we measure the number of objects that match the associated n-dimensional selection query C_q (Section 4.2). In Section 6, we report the average over all queries of the number of tuples retrieved as the fraction of the number of (not necessarily distinct) tuples in the database (*% of tuples retrieved*). This metric reveals the tightness of our mapping of a top-k query into a traditional selection query. A complementary metric is *% of restarts*, the percentage of queries in our workload for which the associated selection query failed to contain the k best tuples, hence leading to restarts. (See Step (5) of the algorithm of Section 4.)

It is important to distinguish between the tightness of the mapping of a top-k query to a traditional selection query, and the efficiency of execution of the latter. The tightness of the mapping depends on the mapping algorithms (Section 4) and on their interaction with the quality of the available histograms. The efficiency of execution of the selection query produced by our mapping algorithm depends in turn on the indexes available on the database and on the optimizer's choice of an execution plan. The cost estimator in an optimizer determines the best access path among the available choices. (These choices include performing a sequential scan of the data.) In this paper, we will not discuss further details of efficient execution of selection queries on databases but rather focus on the problem of mapping top-k queries to selection queries efficiently using histogram structures.

6 Experimental Results

This section presents experimental results for our techniques of Section 4 for evaluating top-k queries. In particular, we study the role of several factors on the efficiency of our strategies, including the size and type of n-dimensional histograms available, the scoring function used in the queries, and the dimensionality and skew of the data sets. Our experiments then 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 databases built with the $Z21$ (moderate) skew (Section 5.1), the *PHASED* technique for building n-dimensional histograms (Section 5.2), and allocates 5KB per histogram. For each experiment, we generated 100 different queries. Each query was created by picking each attribute value randomly from the $[0, 1]$ range. In the default setting, these queries ask for top 10 tuples (i.e., $k = 10$). We report results for other settings of the parameters as well.

Validity of our General Approach

Our general approach for processing a top-k query q (Section 4.2) is to find an n-rectangle that contains all the top k tuples for q, and use this rectangle to build a traditional selection query. Our first experiment studies the intrinsic limitations of our approach, i.e., whether it is possible to build a "good" n-rectangle around query q that contains all top k tuples and little else. To answer this first question, independent of any available histograms or search-score selection strategies (Section 4), we first scanned the database to find the actual top k tuples for a given query q, and determined a tight n-rectangle T that encloses all of these tuples. We then computed what fraction of the database tuples lies within rectangle T. Table 3 reports these figures. As we can see from the table, the fraction of tuples that lie in this "ideal" rectangle is extremely low, which validates our approach: if the database statistics (i.e., histograms) are accurate enough, then we should be able to find a tight n-rectangle that encloses all the best tuples for a given query, with few extra tuples.

Data Distribution	*Scoring*	n = 2	3	4
$Z10$	*Min*	0.01	0.01	0.01
	Sum	0.01	0.01	0.01
	Euclidean	0.01	0.01	0.01
$Z21$	*Min*	0.03	0.03	0.02
	Sum	0.04	0.02	0.02
	Euclidean	0.04	0.02	0.01
$Z32$	*Min*	0.38	0.76	0.16
	Sum	0.15	0.05	0.09
	Euclidean	0.10	0.04	0.06

Table 3: The percentage of tuples in the database included in an n-rectangle enclosing the actual top-k tuples for a query ($k = 10$; $N = 100,000$ tuples).

Effect of Multidimensional Histograms

For this experiment, we considered the *AVI*, *PHASED*, and *MHIST*-2 histogram structures (Section 5.2). *AVI* proved to be significantly worse than *MHIST* and *PHASED* since it tended to require restarts in most cases, while retrieving only an extremely low fraction of the database tuples. In effect, the *NoRestarts* strategy of Section 4.1 guarantees no restarts only in the presence of an accurate n-dimensional histogram. *AVI* can only *estimate* the holdings of each n-dimensional bucket by assuming that attributes follow independent distributions. The results for *AVI* were so poor that we omit this histogram structure from the rest of the discussion.

For *PHASED* and *MHIST*, we varied the amount of storage that we allocated for the histograms. Figure 5 shows the effect of this variation for the *Euclidean* scoring function. (The results for *Min* and *Sum* are analogous.) In this figure, we report the results for the *NoRestarts* and the *Inter1* policies of Section 4.1. When we increase the histogram size from 1KB to 5KB, there is a sharp improvement in the efficiency of our technique, as evidenced by the drop in the percentage of tuples retrieved. *PHASED* performs (marginally) better than *MHIST* and therefore for the rest of this section we report results mainly using *PHASED*. Although higher memory allocation clearly increases accuracy, as shown by the figures, we decided to settle on a 5KB budget for each histogram in the rest of this paper.

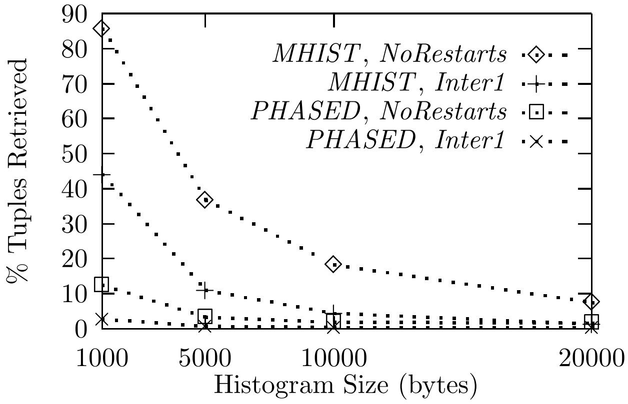


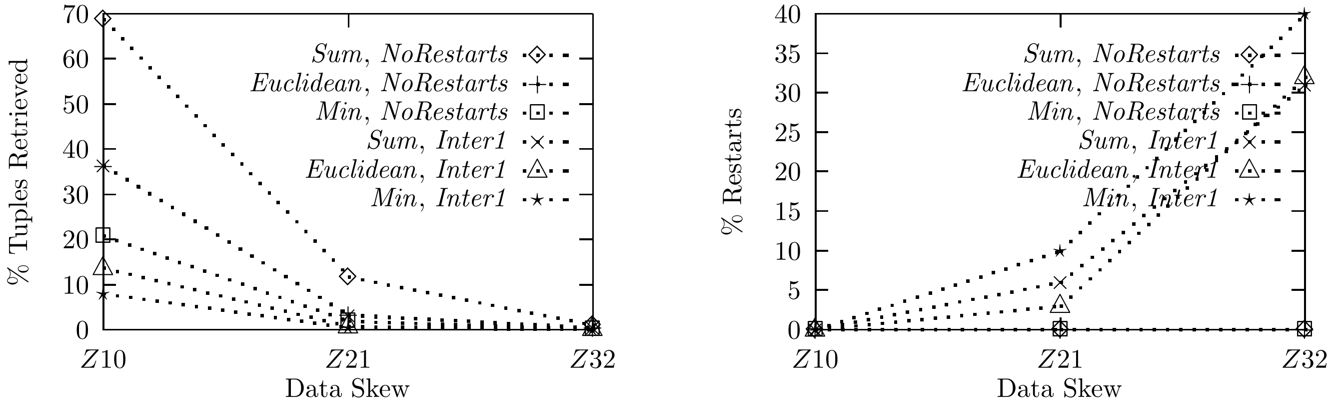
Figure 5: The percentage of tuples retrieved, as a function of the number of bytes dedicated to the n-dimensional histogram (*Euclidean* scoring function; $n = 3$; $Z21$ data distribution).

Effect of Different Scoring Functions

The goal of this experiment is to measure the differences among scoring functions as the data skew and the number of dimensions are varied (Section 5.1). Figure 6 shows that, as the data skew increases, the percentage of tuples retrieved decreases sharply and consistently across all scoring functions. On the other hand, as the number of attributes n is increased (Figure 7), the performance of our techniques drops. Interestingly, the *Min* scoring function copes significantly better with the increase in n than the other scoring functions. As mentioned in Section 4.2, the shape of the region containing the top tuples for a query matches an n-rectangle perfectly, unlike the case for *Sum* and *Euclidean*. The performance of *Euclidean*, though, is better than that of *Sum*. As can be observed from Table 1 and Figures 1(a) and (b), the size of the n-rectangle enclosing the top tuples for *Sum* is much larger than that for *Euclidean* (Sections 4.1 and 4.2).

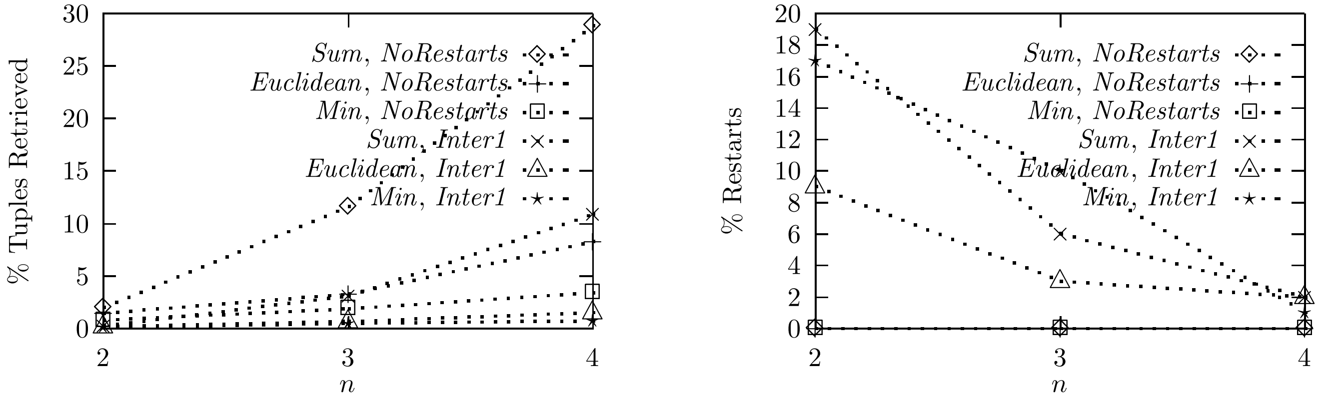
Effect of the Number of Tuples Requested k

Figure 8 studies the effect of increasing k, the number of tuples requested in a top-k query. As k is increased from 10 to 100, the performance drops. As in the pre-



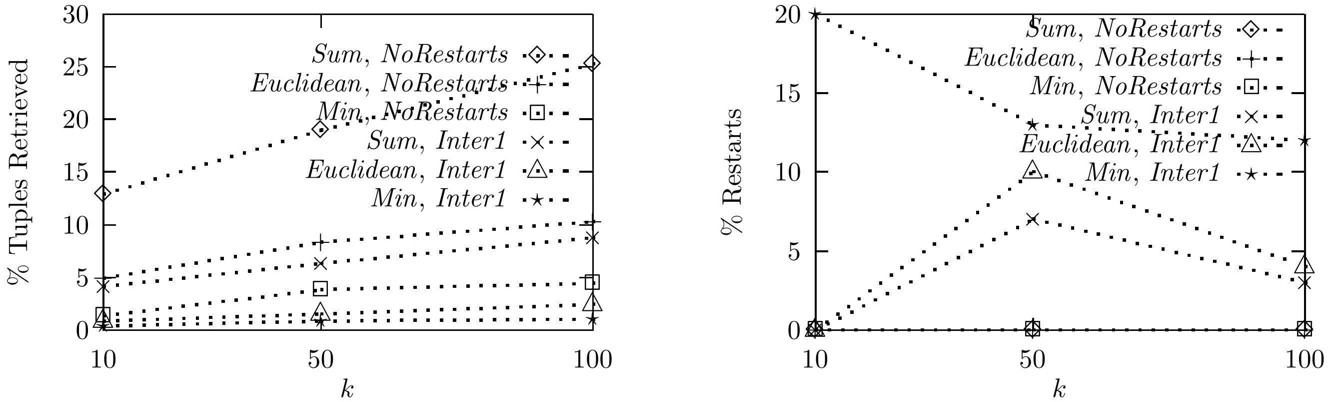
(a) (b)

Figure 6: The percentage of tuples retrieved **(a)**, and the percentage of queries that needed restarts **(b)**, for increasing data skew (*PHASED* histogram of 5KB; $n = 3$).



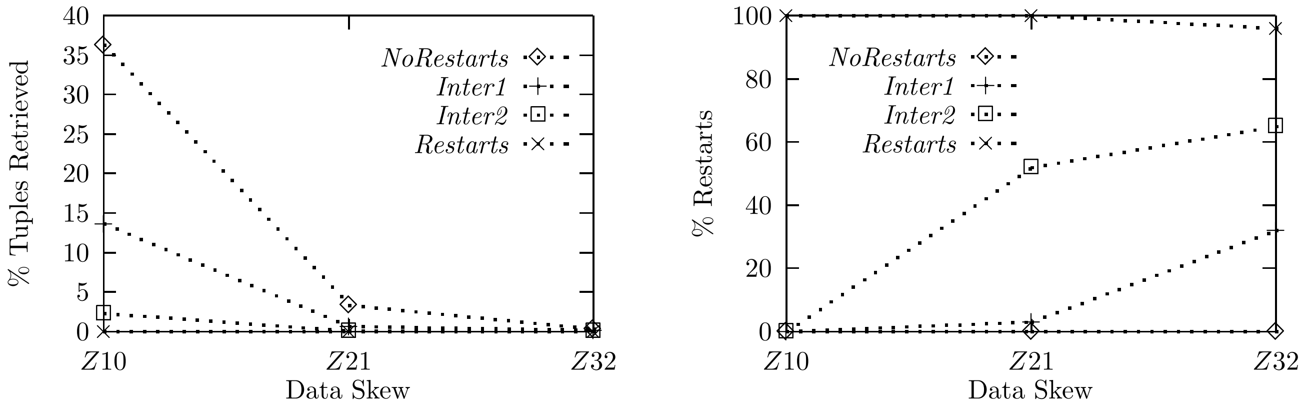
(a) (b)

Figure 7: The percentage of tuples retrieved **(a)**, and the percentage of queries that needed restarts **(b)**, as a function of the number of attributes n (*PHASED* histogram of 5KB; $Z21$ data distribution).



(a) (b)

Figure 8: The percentage of tuples retrieved **(a)**, and the percentage of queries that needed restarts **(b)**, for different values of k (*PHASED* histogram of 5KB; $Z21$ data distribution; $n = 3$).



(**a**) (**b**)

Figure 9: The percentage of tuples retrieved **(a)**, and the percentage of queries that needed restarts **(b)**, for increasing data skew (*Euclidean* scoring function; *PHASED* histogram of 5KB; $n = 3$).

vious experiment, the percentage of tuples retrieved for *Min* grows the slowest, followed by *Euclidean*. The combination of scoring function *Sum* and the *NoRestarts* strategy performs the worst.

Comparing Query Processing Strategies

Figure 9 compares the relative merits of the query processing strategies of Section 4.1. At low data skews, the *NoRestarts* strategy results in a relatively larger number of matching tuples. However, as skew increases, the performance of *NoRestarts* improves significantly and dominates that of the other strategies, since, by definition, it incurs no query restarts with up-to-date histograms. Strategy *Inter1* proves to be a robust technique, since it maintains good performance for all data skews.

Effect of Using n-Rectangle Queries

As explained in Section 4.2, we process a top-k query q by first finding a score S_q and then finding an n-rectangle that encloses all tuples with a *Score* of S_q or higher. Our goal is for the n-rectangle to have as few "bad" tuples as possible, i.e., as few tuples with *Score* lower than S_q as possible. Figure 10 examines this issue by computing the actual number of tuples t with $Score(q,t) \geq S_q$. In other words, we take the score S_q computed by using a histogram and a query processing strategy (Section 4.1), and we count the tuples in the database with that score or higher. We can then compare these numbers against those in Figure 9(a) to conclude that using n-rectangles for retrieving the database tuples does not result in a major source of inefficiency, since the percentage of tuples in both cases is quite comparable.

Results for the Census Database

Figure 11 shows how our query processing strategies perform on the Census data set (Section 5.1). While none of the strategies resulted in a significant number

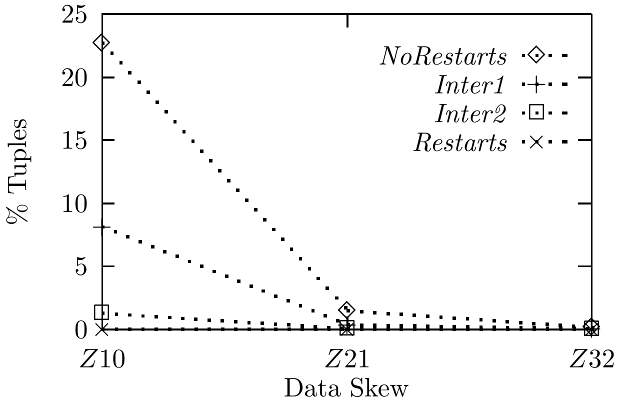


Figure 10: The average number of tuples (as a percentage of N) with score S_q or higher (Step (1) of the Section 4 algorithm) for increasing data skew (*Euclidean* scoring function; *PHASED* histogram of 5KB; $n = 3$).

of restarts (hence we do not show the corresponding plot here), the robustness of strategy *Inter1* for increasing histogram size can be seen clearly. The performance for the different scoring functions is consistent with the results obtained for the synthetic databases described above.

7 Conclusions and Future Work

In this paper, we studied the problem of mapping a top-k query on a relational database to a traditional selection query such that the mapping is "tight," i.e., we retrieve as few tuples as possible. Our mapping algorithms exploit the histogram structures and are able to cope with a wide variety of scoring functions. Our experiments highlighted the effect of different scoring functions, data distributions, as well as histogram-building strategies on the performance of this mapping.

Our focus in this paper has been primarily on queries over continuous attributes. In the future, we will extend our techniques to handle top-k queries over

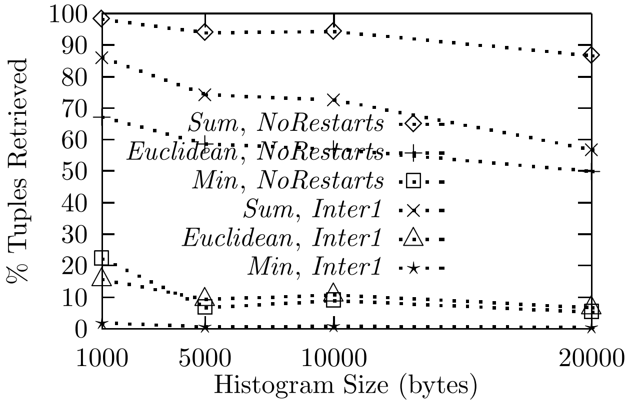


Figure 11: The percentage of tuples retrieved, as a function of the number of bytes dedicated to the histogram (*Census* database; *PHASED* histogram).

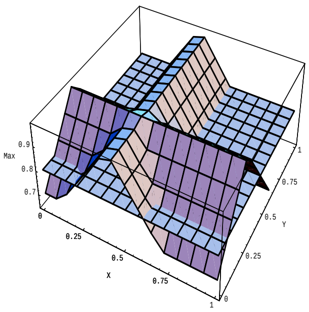


Figure 12: The scores for query $q = (0.4, 0.3)$ for scoring function *Max*.

discrete attributes. Another direction for future work is to explore approaches to support top-k queries with scoring functions (e.g., *Max*) that cannot be mapped tightly to the family of traditional selection queries that we used in this paper (Figure 12).

Acknowledgments

We thank Eugene Agichtein and David Lomet for their useful comments.

References

[1] 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.

[2] 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.

[3] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. In *Proceedings of the 1996 ACM International Conference on Management of Data (SIGMOD'96)*, June 1996.

[4] R. Fagin. Combining fuzzy information from multiple systems. In *Proceedings of the Fifteenth ACM Symposium on Principles of Database Systems (PODS'96)*, June 1996.

[5] R. Fagin. Fuzzy queries in multimedia database systems. In *Proceedings of the Seventeenth ACM Symposium on Principles of Database Systems (PODS'98)*, June 1998.

[6] L. Gravano and H. García-Molina. Merging ranks from heterogeneous Internet sources. In *Proceedings of the Twenty-third International Conference on Very Large Databases (VLDB'97)*, Aug. 1997.

[7] 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.

[8] W. Meng, K.-L. Liu, C. Yu, X. Wang, Y. Chang, and N. Rishe. Determining text databases to search in the Internet. In *Proceedings of the Twenty-fourth International Conference on Very Large Databases (VLDB'98)*, Aug. 1998.

[9] A. Motro. VAGUE: A user interface to relational databases that permits vague queries. *ACM Transactions on Office Information Systems*, 6(3):187–214, July 1988.

[10] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multidimensional queries. In *Proceedings of the 1988 ACM International Conference on Management of Data (SIGMOD'88)*, June 1988.

[11] 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.

[12] 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.

[13] G. Salton and M. J. McGill. *Introduction to modern information retrieval.* McGraw-Hill, 1983.

[14] 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.

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