Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation

NICOLAS BRUNO
Columbia University

SURAJIT CHAUDHURI
Microsoft Research

and

LUIS GRAVANO
Columbia University

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 a traditional relational database management system (RDBMS) 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 an RDBMS, and the impact of the quality of these statistics on the retrieval efficiency of the resulting scheme. We also report the first experimental evaluation of the 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.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems—Query processing; Relational databases; H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval—Retrieval models; Search process; H.3.4 [Information Storage and Retrieval]: Systems and Software—Performance evaluation


Additional Key Words and Phrases: Top-k query processing, multidimensional histograms.

1. INTRODUCTION

Approximate matches of queries are commonplace in the text world. Notably, web 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, the query result is not just a set of objects that match a given condition, but a ranked list of objects. 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 efficiently rank text documents for a query [Salton and McGill 1983]. 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.

Example 1.1. 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, six 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, or top-k queries, for short. Unlike the case with a traditional selection query, there may be many ways to define how to match a query and a database tuple.

Example 1.1. (cont.) In our example scenario above, the database system picked a house without a perfect number of bedrooms (i.e., six) but with a price tag close to the target price (i.e., $300,000) as the best house. For a wealthy customer, the system might choose to match the query against the tuples differently. In particular, the system might then prefer number of bedrooms over price, and return a house with four bedrooms with an exorbitant price tag as the best match for the given query.

A large body of work has addressed how to find the nearest neighbors of a multi-dimensional data point [Korn et al. 1996; Seidl and Kriegel 1998]. Many techniques use specialized data structures and indexes [Guttman 1984; Nievergelt et al. 1984; Lomet and Salzberg 1990] to answer nearest-neighbor queries. These index structures and access methods are not currently supported by many traditional relational database management systems (RDBMS). Therefore, despite the conceptual simplicity of top-k queries and the expected performance payoff, these queries are not yet effectively supported by most RDBMSs. Adding this necessary 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 necessarily require full sequential scans of the underlying relations. The challenge in providing this functionality is that the database system needs to handle top-k queries efficiently for a wide variety of ranking functions. In effect, these ranking
functions might change by user, 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.

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 B+-tree indexes over single as well as possibly multi-column attributes. 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 ranking 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).

Note that our goal is not to develop new stand-alone algorithms or data structures for the nearest-neighbor problem over multidimensional data. Rather, this paper addresses the problem of mapping a top-k selection query to a traditional range selection query that can be optimized and executed by any vanilla RDBMS. We undertake a comprehensive study of the problem of mapping top-k queries into execution plans that use traditional selection queries. In particular, we consult the database histograms to map a top-k query to a suitable 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., a supplemental query needs 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 in most cases.

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 for 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 the techniques requiring sequential scans.

The paper is organized as follows. In Section 2, we formally define the problem of querying for top-k matches. In Section 3, we outline the basis of our approach and present some static mapping techniques. In Section 4, we present a dynamic technique that adapts to the query workload and results in better results than the static approaches. Finally, in Section 6 we present the experimental evaluation of our techniques on Microsoft’s SQL Server 7.0 using the setting of Section 5. A preliminary version of this paper appeared in [Chaudhuri and Gravano 1999].

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 selection query is an ordered set of tuples, where the ordering criterion is how well each tuple matches the given query. In this section
we present the query model precisely.

Consider a relation $R$ with attributes $A_1, \ldots, A_n$. A top-$k$ selection 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-$k$ selection query $q$ is then an ordered set of $k$ tuples of $R$ that are closest to $q$ according to the given distance function.\footnote{In \cite{Chaudhuri and Gravano 1999} 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.}

**Example 2.1.** Consider a relation Employee with attributes Age and Hourly Wage. The answer to the top-10 selection query $q = (30, 20)$ is an ordered sequence consisting of the 10 employees in the Employee relation that are closest to 30 years of age and to making an hourly wage of $20, according to a given distance function, as discussed below.

A possible SQL-like notation for expressing top-$k$ selection queries is as follows \cite{Chaudhuri and Gravano 1996}:

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

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 Dist function.

Given a top-$k$ query $q$ and a distance function $Dist$, the database system with relation $R$ uses $Dist$ to determine how closely each tuple in $R$ matches the target values $q_1, \ldots, q_n$ specified in query $q$. Given a tuple $t$ and a query $q$, we assume that $Dist(q,t)$ is a positive real number. 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:

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

Given a $p$-norm $||\cdot||$, we can define a distance function $D_{||\cdot||}$ between two arbitrary points $q$ and $t$ as $D_{||\cdot||}(q,t) = ||q-t||$. This paper focuses on the following important distance functions, which are based on $p$-norms for $p = 1, 2, \infty$.

**Definition 2.2.** 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 :

- $Sum(q,t) = ||q-t||_1 = \sum_{i=1}^n |q_i - t_i|$
- $Eucl(q,t) = ||q-t||_2 = \sqrt{\sum_{i=1}^n (q_i - t_i)^2}$
- $Max(q,t) = ||q-t||_{\infty} = \max_{i=1}^n |q_i - t_i|$

**Example 2.3.** Consider a tuple $t = (50, 35)$ in our sample database Employee from Example 2.1, and a query $q = (30, 20)$. Then, tuple $t$ will have a distance of
Fig. 1. The distances (z axis) between all points and query $q = (30, 20)$ for the different $(x, y)$ pairs and for distance functions $\text{Sum}$ (a), $\text{Eucl}$ (b), and $\text{Max}$ (c).

$\text{Max}(q, t) = \text{Max}\{|30 - 50|, |20 - 35|\} = 20$ for the $\text{Max}$ distance function, a distance of $\text{Eucl}(q, t) = \sqrt{(30 - 50)^2 + (20 - 35)^2} = 25$ for the $\text{Eucl}$ distance function, and a distance of $\text{Sum}(q, t) = |30 - 50| + |20 - 35| = 35$ for the $\text{Sum}$ distance function.

Figure 1(c) shows the distribution of distances for the $\text{Max}$ distance function and query $q = (30, 20)$ for the $\text{Employee}$ relation of Example 2.1. The horizontal plane in the figure consists of the tuples with $z = 15$, so the tuples below this plane are at distance 15 or less from $q$. Note that the tuples at distance 15 or less from $q$ are enclosed in a box around $q$. In contrast, the tuples at distance 15 or less for the $\text{Eucl}$ distance function (Figure 1(b)) are enclosed in a circle around $q$. Finally, the tuples at distance 15 or less for the $\text{Sum}$ distance 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 3.2.

In general, the $\text{Sum}$, $\text{Eucl}$, and $\text{Max}$ functions that we use in this paper are just a few of many possible distance functions. Our strategy for processing top-$k$ queries can be adapted to handle a larger number of functions. For instance, 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.

In general, the key property that we ask from distance functions is as follows:

**Property 2.4.** Consider a relation $R$ and a distance function $\text{Dist}$ defined over $R$. Let $q = (q_1, \ldots, q_n)$ be a top-$k$ query over $R$, and let $t = (t_1, \ldots, t_n)$ and $t' = (t'_1, \ldots, t'_n)$ be two arbitrary tuples in $R$ 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, $\text{Dist}(q, t') \leq \text{Dist}(q, t)$.

Intuitively, this property of distance functions implies that if a tuple $t'$ is closer along each attribute to the query values than some other tuple $t$ is, then, the distance that $t'$ gets for the query cannot be worse than that of $t$. Fortunately, most interesting distance functions seem to satisfy our monotonicity assumptions. In particular, all distance functions based on $p$-norms satisfy this property. In the next section we discuss how we will evaluate top-$k$ queries for different definitions of the $\text{Dist}$ function.
3. STATIC EVALUATION STRATEGIES

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 distance function \( Dist \). Our query processing strategy 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 \( d_q \), such that the region \( reg(q, d_q) \) that contains all possible tuples at distance \( d_q \) or lower from \( q \) is expected to include \( k \) tuples (Section 3.1).

Retrieve

Retrieve all tuples in \( reg(q, d_q) \) using a range query that encloses this region as tightly as possible (Section 3.2).

Verify/Restart

If there are at least \( k \) tuples in \( reg(q, d_q) \), return the \( k \) tuples with the lowest distances. Otherwise, choose a higher value for \( d_q \) and restart the procedure (Section 3.3).

In the next sections we discuss the above steps in detail.

3.1 Choice of Search Distance \( d_q \)

The first step for evaluating a top-k query \( q \) is the most challenging one. Ideally, the search distance \( d_q \) that we determine encloses exactly \( k \) tuples. Unfortunately, identifying such a precise value for \( d_q \) using only relatively coarse histograms is not possible. In practice, we will try to find a value of \( d_q \) such that \( reg(q, d_q) \) encloses at least \( k \) tuples, but not many more. Choosing a value of \( d_q \) 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 \( d_q \) that is too low would result in an execution that requires restarts, which is also undesirable. Hence, determining the right distance \( d_q \) becomes the crucial step in our top-k query processing strategy.

For efficiency, our choice of \( d_q \) will be guided by the statistics that the query processor keeps about relation \( R \), and not by the underlying relation \( R \) itself. In particular, we will assume that we have an \( n \)-dimensional histogram \( H \) that describes the distribution of values of \( R \). Histogram \( H \) consists of a set of pairs \( H = \{(b_1, f_1), \ldots, (b_m, f_m)\} \), where each bucket \( b_i \) defines a hyper-rectangle included in \( \text{domain}(R) \), and each frequency \( f_i \) 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. Figure 2 shows an example of a 50-bucket histogram that summarizes a synthetically generated data distribution.

Specifically, we choose \( d_q \) 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 \(^2\).

\(^2\)In previous work [Bruno et al. 2000] we tried alternative ways to define synthetic relations \( R' \) consistent with histogram \( H \). For instance, we applied the uniformity assumption inside buckets and conceptually distributed the tuples of each bucket \( b \) in a uniform grid inside \( b \)'s bounding box. Those approaches are much more computationally expensive than the one we present in this paper.
b. Compute $\text{Dist}(q,t)$ for every tuple $t$ in $R'$.

c. Let $T$ be the set of the top-$k$ (i.e., closest $k$) tuples in $R'$ for $q$. Output $d_q = \max_{t \in T} \text{Dist}(q,t)$.

We can conceptually build synthetic relation $R'$ in many different ways based on the particular choices for the bucket’s representative tuples. We will first study two “extreme” query processing strategies resulting from two possible definitions of $R'$.

The first query processing strategy, \textit{NoRestarts}, results in a search distance $d_{NR}$ that is high enough to guarantee that no restarts are ever needed as long as histograms are kept up to date. In other words, the \textit{Verify/Restart} step always finishes successfully, without ever having to enlarge $d_q$ and restart the process. For this, the \textit{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:

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

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

\textbf{Example 3.1.} Consider our sample relation \textit{Employee} with attributes age and hourly wage, query $q = (20,15)$, and the 2-dimensional histogram $H$ shown in Figure 3(a). Histogram $H$ has three buckets, $b_1$, $b_2$, and $b_3$. Relation \textit{Employee} has 40 tuples in bucket $b_1$, 5 tuples in bucket $b_2$, and 15 tuples in bucket $b_3$. As explained above, the \textit{NoRestarts} strategy will “build” relation \textit{Employee'} based on $H$ by assuming that the tuple distribution in Employee is as “bad” as possible for query $q$. So, relation \textit{Employee'} 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 15. Assume that the user who issued query $q$ wants to use the $\text{Max}$ distance function to find the top 10 tuples for $q$. Since $\text{Max}(q,t_1) = 35$, $\text{Max}(q,t_2) = 20$, and $\text{Max}(q,t_3) = 30$, to get 10 tuple instances we need the top tuple, $t_2$ (frequency 5), and $t_3$ (frequency 15). Consequently, the search distance $d_{NR}$ will be $\text{Max}(q,t_3) = 30$. From the way we built Employee', it follows that the original relation \textit{Employee} is guaranteed to contain at least 10 tuples with distance

and they result in many restarts, mostly because of the often not-so-uniform buckets produced by state-of-the-art multidimensional construction techniques. Therefore, we do not consider those alternatives in this paper.
Fig. 3. A 3-bucket histogram $H$ and the choice of tuples representing each bucket that strategies NoRestarts (a) and Restarts (b) make for query $q$.

dNR_q = 30 or lower to query $q$. Then, if we retrieve all of the tuples at that distance or lower, we will obtain a superset of the set of top-$k$ tuples for $q$.

**Lemma 3.2.** Let $q$ be a top-$k$ query over a relation $R$. Let $dNR_q$ be the search distance computed by strategy NoRestarts for query $q$ and distance function $Dist$. Then, there are at least $k$ tuples $t$ in $R$ such that $Dist(q,t) \leq dNR_q$.

The second query processing strategy, Restarts, results in a search distance $dR_q$ that is the lowest among those search distances 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 $b$ 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:

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

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

**Example 3.1. (cont.)** The Restarts strategy will now "build" relation Employee based on $H$ by assuming that the tuple distribution in $S$ is as "good" as possible for query $q$ (Figure 3(b)). So, relation Employee will consist of three tuples (one per bucket in $H$) $t_1$, $t_2$, and $t_3$, which 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 distance from $q$. For Max, we pick tuples $t_2$ and $t_3$, and define $dR_q$ as $\max(q,t_3)$. This time it is indeed possible for fewer than $k$ tuples in the original table Employee to be at a distance of $dR_q$ or lower from $q$, so restarts are possible.

The search distance $dR_q$ that Restarts computes is the lowest distance that might result in no restarts in the Verify/Restart step of the algorithm in Section 3. In other words, using a value for $d_q$ that is lower 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 3.3.** Let $q$ be a top-$k$ query over a relation $R$. Let $dR_q$ be the search distance computed by strategy Restarts for query $q$ and distance function $Dist$. Then, there are fewer than $k$ tuples $t$ in $R$ such that $Dist(q,t) < dR_q$.

The norm-based distance functions that we use are monotonic (Property 2.4). For that reason, the coordinates for the tuples in the Restarts and NoRestarts strategies can be easily computed. Specifically, the point in the set of all potential tuples associated with bucket $b$ that is closest to (similarly, farthest from) a query $q$ can be determined dimension by dimension, as the following example illustrates.

**Example 3.4.** Consider a bucket $b$ that is defined by its corners $(10,10)$ and $(25,40)$, and a query $q=(40,20)$ (Figure 4). Assume that we use the Euclidean distance function. Because of the monotonicity property of Euclidean distance, 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 4). 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, $\min_{t \in T_b} \text{Eucl}(q,t) = \text{Eucl}(q,q_1) = \sqrt{(40-25)^2 + (20-20)^2} = 15$ and $\max_{t \in T_b} \text{Eucl}(q,t) = \text{Eucl}(q,q_2) = \sqrt{(40-10)^2 + (20-40)^2} = 36.1$

In general, the two distance-selection strategies NoRestarts and Restarts are not efficient in practice due to the extreme assumptions they make, as we illustrate in the following example and confirm in Section 6.

**Example 3.5.** Consider the relation and histogram of Example 3.1. Figure 5 shows the Restarts and NoRestarts search distances for query $q$, $k=10$ and the Euclidean distance function. As explained above, 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 $b_2$ and $b_3$ completely, hence including at least $15 + 5 = 20$ tuples. Unfortunately, this strategy will most likely also retrieve a significant fraction of the 40 $b_1$ 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 5 tuples in bucket $b_3$ are as close to $q$ as possible and the 5 $b_2$ tuples are at least as close to $q$ as the 5 $b_3$ tuples are. Unfortunately, this optimistic scenario is improbable, and the Restarts strategy will most likely result in restarts (Verify/Restart step) and in an inefficient execution overall.

For those reasons, we will study two intermediate strategies, Inter1 and Inter2 (Figure 6). Given a query $q$, let $d_{NR}q$ be the search distance selected by NoRestarts for $q$, and let $d_{R}q$ be the corresponding distance selected by Restarts. Then, the
Restarts NoRestarts

Fig. 5. Regions searched by the Restarts and NoRestarts strategies for a top-10 query $q$.

Fig. 6. The four static strategies for computing the search distance $S_q$.

$Inter_1$ strategy will choose distance $\frac{2d_R + 2d_{NR}}{3}$, while the $Inter_2$ strategy will choose a higher distance of $\frac{d_R + 2d_{NR}}{3}$. We will define even more alternatives in Section 4.

3.2 Choice of Selection Query $C_q$

Once the Search step has determined the search distance $d_q$, the Retrieve step builds and evaluates a SQL query $C_q$ that encloses all tuples with distance $d_q$ or lower from $q$ tightly. In this section we describe how to define such query $C_q$.

Ideally, we would like to ask our database system to return exactly those tuples $t$ such that $Dist(q, t) \leq d_q$. Unfortunately, typical indexing structures in relational DBMSs do not natively support such predicates (Section 7). Hence, our approach is to build $C_q$ as a simple selection condition that defines an $n$-rectangle. In other words, we define $C_q$ as a query of the form:

\[
\text{SELECT * FROM } R \\
\text{WHERE } (a_1 \leq A_1 \leq b_1) \land \ldots \land (a_n \leq A_n \leq b_n)
\]

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 $Dist(q, t) \leq d_q$.

Example 3.6. Consider our example query $q = (20, 30)$ over relation Employee, with Sum as the distance function. Let $d = 15$ be the search distance determined in the Search step using any of the strategies previously discussed. Each tuple $t$ with $\text{Eucl}(q, t) < 15$ lies in the circle around $q$ that is shown in Figure 7. Then, the tightest $n$-rectangle that encloses that circle is $[5, 35] \times [15, 45]$. Hence, the final SQL query $C_q$ is:

\[
\text{SELECT * FROM Employee} \\
\text{WHERE } (5 \leq \text{AGE} \leq 35) \land (15 \leq \text{HOURLY-WAGE} \leq 45)
\]
Given a search distance $d_q$, the $n$-rectangle $[a_1, b_1] \times \ldots \times [a_n, b_n]$ that determines $C_q$ follows directly from the distance function used, the distance $d_q$, and the query $q$. In particular, for the three distance functions discussed in this paper the $n$-rectangle for $C_q$ is the $n$-rectangle centered on $q$ with sides of length $2d_q$. The Max scoring function presents an interesting property: the region to be enclosed by the $n$-rectangle is already an $n$-rectangle (Figure 1(c)). Consequently, the query $C_q$ that is generated for Max for query $q$ and its associated search distance $d_q$ will retrieve only tuples with a distance of $d_q$ or lower. This property will result in efficient executions of top-$k$ queries for Max, as we will see. Unfortunately, this property does not hold for the Sum and Eucl distance functions (see Figures 1(a) and (b)).

### 3.3 Choice of Restarts Distance

Since we use coarse statistics from histograms to choose the search distance $d_q$, the Retrieve step might yield fewer than $k$ tuples at distance $d_q$ or less. If this is the case, we need to choose a higher search distance $d'_q$ and restart the procedure. There are several ways to select $d'_q$. In this paper, we use a simple approach: whenever we need to restart, we choose $d_{NR_q}$, the search distance returned by the NoRestarts strategy, as the new search distance $d'_q$. This choice guarantees success this second time since, by definition, at least $k$ tuples in the relation are at distance $d_{NR_q}$ or less from the query.

### 4. A DYNAMIC WORKLOAD-BASED MAPPING STRATEGY

As we will see in Section 6, the strategies described in the previous section perform reasonably well in practice. However, no strategy is consistently the best across data distributions. Moreover, even over the same data sets, which strategy works best for a query $q$ sometimes depends on the specifics of $q$. In this section, we introduce a parametric mapping strategy that can be seen as a generalization of the four strategies of Section 3. We also derive a simple procedure to choose the parameter that leads to the “best” strategy for a given workload. That is, future queries from similar workloads, i.e., queries whose probabilistic spatial distribution is similar to that of the training workload, will result in efficient executions. Since the resulting mapping strategy will depend on the particular workload (as opposed to the static techniques of Section 3) we call this new technique Dynamic.
4.1 Adapting to the Query Workload

The four static mapping strategies that we introduced in Section 3 for answering top-k queries can be seen as special cases of the following parametric strategy with parameter $\alpha$:

$$d_q(\alpha) = dR_q + \alpha \cdot (dNR_q - dR_q) \quad 0 \leq \alpha \leq 1$$

where $dR_q$ and $dNR_q$ are the Restarts and NoRestarts search distances for query $q$. In fact, by instantiating $\alpha$ with 0, $\frac{1}{3}$, $\frac{2}{3}$ and 1, we obtain the Restarts, Inter1, Inter2 and NoRestarts mapping strategies, respectively.

In general, for each query $q$ that we consider, there is an optimum value $\alpha_q$ such that at least $k$ tuples are at distance $d_q(\alpha_q)$ or less from $q$ and the number of such tuples is as close to $k$ as possible. Unfortunately, it is not possible to determine $\alpha_q$ a priori without examining the actual tuples. Our approach will be, given a workload $Q$, to find a single value $\alpha^*$, $0 \leq \alpha^* \leq 1$, such that $d_q(\alpha^*)$ minimizes the average number of tuples retrieved for similar workloads.

More formally, consider a workload $Q = \{q_1, \ldots, q_n\}$ of top-k queries. The total number of tuples retrieved for search distance $d_q(\alpha)$ includes:

$$totalTuples(Q, \alpha) = \sum_{q_i \in Q} \left(\text{tuples}(q_i, d_{q_i}(\alpha)) + \begin{cases} 0 & \text{if tuples}(q_i, d_{q_i}(\alpha)) \geq k \\ \text{tuples}(q_i, dNR_{q_i}) & \text{otherwise (i.e., we restart)} \end{cases}\right)$$

where $\text{tuples}(q, d)$ is the number of tuples in the data set at distance $d$ or lower from $q$. (Additional tuples will be retrieved for Eucl and Sum because these non-rectangular regions are mapped to range queries for processing (Section 3.2).)

A good value for $\alpha$ should be high enough, so that at least $k$ tuples are retrieved, but not too high, so that not too many extra tuples are retrieved. Although a value of $\alpha$ that is too low will result in few tuples being retrieved during the Retrieve step, we might require to restart the query, hence retrieving many tuples during the Verify/Restart step. We then define the Dynamic mapping strategy as using search distance $d_q(\alpha^*)$, where $\alpha^*$ is such that:

$$totalTuples(Q, \alpha^*) = \min_{\alpha} totalTuples(Q, \alpha)$$

The following example illustrates the tension between the number of tuples retrieved in the Retrieve and Verify/Restart steps, i.e., between the two components of the function $totalTuples$ that we want to minimize.

**Example 4.1.** Consider a Gauss data set and a workload consisting of 500 top-k queries. Figure 8(a) reports the average number of tuples retrieved in the Retrieve and Verify/Restart steps, and Figure 8(b) reports the percentage of queries that needed restarts. When $\alpha$ is close to zero, the number of tuples retrieved

---

3In [Bruno et al. 2000] we also investigated associating a value of $\alpha$ for each histogram bucket. The gains in selectivity estimation accuracy do not justify the added storage requirements to record these $\alpha$’s.

4See Section 5 for more information about data sets.
in the Retrieve step is small. However, the percentage of restarts is near 100%, meaning that in almost all cases those initial queries returned fewer than $k$ tuples, so supplemental (expensive) queries were issued in the Restart step. Therefore, the total number of tuples for $\alpha$ near zero in Figure 8(a) is high. As $\alpha$ increases, the percentage of restarts and the number of tuples retrieved in the Verify/Restart step decreases, since the resulting search distances are closer to those of the NoRestarts strategy. However, for the same reason, the average number of tuples retrieved in the Retrieve step increases as well. When $\alpha$ is near one, there are almost no restarts, but the original queries in the Retrieve step are much more expensive due to the larger search distances being used. The net result is, again, a high number of tuples retrieved for $\alpha$ near one. In this example, a value of $\alpha$ around 0.2 results in the lowest number of tuples retrieved.

If $\alpha^*$ is calculated accurately enough, the Dynamic mapping strategy will consistently result in better performance that any of the static strategies of Section 3, as illustrated in the following example and verified experimentally in Section 6.

Example 4.2. Consider the Gauss data set of the previous example. Figure 9 shows the total number of tuples retrieved (totalTuples) as a function of $\alpha$ for two different workload configurations. For one workload (denoted Biased in Figure 9) the minimal value of totalTuples occurs for $\alpha^*$ around 0.2. In this case, strategy Inter1 ($\alpha = 0.33$) would have been the best of the four static strategies of Section 3. However, using $\alpha^* = 0.2$ results in an even smaller number of tuples retrieved.
the other hand, for the Uniform workload, the optimum value of \( \alpha^* \) is near 0.68. Strategy \( \text{Inter2}(\alpha = 0.67) \) would have been the best strategy among the static ones in this case.

4.2 Implementation Considerations

The previous section introduced the Dynamic mapping strategy based on parameter \( \alpha^* \). In this section we describe how to efficiently approximate the optimal \( \alpha^* \) for a given workload.

Once the training workload is fixed, \( \text{totalTuples} \) becomes a unidimensional function on \( \alpha \). Therefore, we can use some unidimensional optimization technique such as golden search \cite{William et al. 1993} \(^5\) to find \( \alpha^* \). The golden search minimization technique needs to evaluate the function \( \text{totalTuples} \) at arbitrary points during its execution. We can precalculate the values \( \text{tuples}(q_i, d_{NRq_i}) \) in the definition of \( \text{totalTuples} \) above for all queries \( q_i \in Q \) so that the “restart” term in the definition of \( \text{totalTuples} \) need not be recalculated at each iteration. However, we still need to calculate the value of \( \text{tuples}(q_i, d_{q_i}(\alpha)) \) for arbitrary values of \( \alpha \) at each iteration of the golden search procedure. We could issue a sequential scan over the data to calculate \( \text{tuples}(q_i, d_{q_i}(\alpha)) \) each time, but this strategy would be too expensive. Even if we use multi-query evaluation techniques, i.e., we calculate \( \text{tuples}(q_i, d_{q_i}(\alpha)) \) for all queries \( q_i \in Q \) at once, we would still have to perform several sequential scans over the data set (as many as the underlying golden search procedure needs).

Instead, we propose to estimate the function \( \text{tuples}(q, d) \) in a preprocessing step. The resulting estimated function, denoted \( \text{tuples}'(q, d) \), should be (1) accurate enough so that the optimum value \( \alpha \) determined by using \( \text{tuples}' \) is close to the actual optimum \( \alpha^* \), and (2) efficiently computed, since we want to avoid repeated sequential scans over the data sets. We now present a simple definition of the function \( \text{tuples}' \) and a procedure for computing it that only needs one sequential scan over the data set (or, as we will see, even less than a sequential scan if we use sampling).

Suppose that we know, for each query \( q \) in the workload \( Q \), the following \( G + 1 \) discrete values:

\[
T^i_q = \text{tuples}(q, d_{Rq} + \frac{i}{G}(d_{NRq} - d_{Rq})) \quad i \in \{0, 1, \ldots, G\}
\]

where \( G \) is some predetermined constant. Then, we can use linear interpolation (see Figure 10) to approximate \( \text{tuples}(q, d_{q}(\alpha)) \) for arbitrary values of \( \alpha \), \( 0 \leq \alpha < 1 \):

\[
\text{tuples}'(q, d_{q}(\alpha)) = T^I_q + \alpha(T^{I+1}_q - T^I_q), \quad \text{where } I = \lfloor \alpha \cdot G \rfloor
\]

Since we also have that \( T^G_q = \text{tuples}(q, d_{NRq}) \), we can efficiently approximate the function \( \text{totalTuples} \). The procedure below calculates the values \( T^i_q \) by first filling an array \( \tau \) where \( \tau^i_t \) is the number of tuples \( t_i \) in \( D \) such that \( d_{q_i}(\frac{t_i}{I}) \leq \text{Dist}(t_i, q) \leq d_{q_i}(\frac{t_i}{I} + 1) \) and then adding up these partial results to obtain \( T^i_q \):

\(^5\)Note that the function \( \text{totalTuples} \) we defined is not continuous on \( \alpha \) and might have local minima since we have a finite workload and the restarts include a non continuous component. However, if the workload is large enough, we can consider \( \text{totalTuples} \) as a continuous function with only one minimum.
Top-k Selection Queries over Relational Databases

Fig. 10. Using linear interpolation to approximate \( tuples(q, d) \) from a set of discrete pairs \( \left( \frac{\alpha}{G}, T^i \right) \).

Procedure \( \text{calculateT}(D: \text{Data Set}, Q: \text{Workload}, G: \text{integer}) \)

- Set \( \tau_j^k = 0 \), for \( j \in \{0, 1, \ldots, |Q|\} \) and \( k \in \{0, 1, \ldots, G\} \)
- for each tuple \( t_i \) in \( D \) // Sequential scan over \( D \)
  - for each query \( q_j \) in \( Q \)
    - \( d = \text{Dist}(t_i, q_j) \)
    - if \( (d < dR_{q_j}) \) \( \tau_j^0 += \) // we count \( t_i \) in \( \tau_j^0 \)
    - else if \( (d < dNR_{q_j}) \)
      - \( g = \left\lceil \frac{G \cdot (d - dR_{q_j})}{dNR_{q_j} - dR_{q_j}} \right\rceil \) // \( 0 \leq g \leq G \)
      - \( \tau_j^g += \)
  - // At this point, \( T^k_j = \sum_{k'=0}^{k} \tau_j^{k'} \)
  - Calculate and return \( T^k_j \)

The value \( G \) specifies the granularity of the approximation. Higher values of \( G \) result in better accuracy of \( tuples' \). It is interesting to note that increasing the value of \( G \) results in more accurate approximations, but it does not increase the running time of the algorithm (memory does increase linearly with \( G \)). In our experiments, we set \( G = 50 \). To obtain the optimum value \( \alpha^* \) for a given data set \( D \) and workload \( Q \) when using histogram \( H \) we simply need to perform the following steps:

a. Calculate \( dR_q \) and \( dNR_q \) for each \( q \in Q \) using histogram \( H \).
b. Compute \( T = \text{calculateT}(D, Q, G) \).
c. Use golden search to return the value of \( \alpha \in [0, 1] \) that minimizes:

\[
totalTuples(\alpha) = \sum_{q_i \in Q} \left( tuples'(\alpha) + \begin{cases} 0 & \text{if } tuples'(\alpha) \geq k \\ T_q^{\alpha-G} & \text{otherwise (we restart)} \end{cases} \right)
\]

where \( tuples'(\alpha) = T_q^{[\alpha-G]} + \alpha(T_q^{[\alpha-G]+1} - T_q^{[\alpha-G]}) \)

The efficiency of the procedure \( \text{calculateT} \) can be dramatically improved if we use sampling instead of processing all tuples in the data set via a sequential scan. In fact, sampling provides an efficient and accurate way to approximate the function \( totalTuples \). Figure 11 shows the exact and approximated values of \( totalTuples \) for different values of \( p \), the fraction of tuples sampled, for one of the real data sets.
of Section 5 and a value of \( G \) fixed at 50. We can see that for \( p = 10\% \) the exact and approximated values for \( \text{totalTuples} \) are indistinguishable. Even for \( p = 1\% \) the differences between the exact and approximated \( \text{totalTuples} \) are minimal. In contrast, for \( p = 0.1\% \) the approximated \( \text{totalTuples} \) is significantly different. (Note that for this setting we only examine around 210 tuples out of about 210,000.)

5. EXPERIMENTAL SETTING

This section defines the data sets, histograms, and workloads used for the experiments of Section 6.

5.1 Data Sets

We use both synthetic and real data sets for the experiments. The real data sets we consider [Blake and Merz 1998] are: Census2D and Census3D (two- and three-dimensional projections of a fragment of US Census Bureau data), and Cover4D (four-dimensional projection of the CovType data set, used for predicting forest cover types from cartographic variables). The dimensionality, cardinality, and attribute names for each real data set are in Table I.

We also generated a number of synthetic data sets for our experiments [Bruno et al. 2001], following different data distributions:

—\textit{Gauss}: The Gauss synthetic distributions [William et al. 1993] consist of a predetermined number of overlapping multidimensional gaussian bells. The parameters for these data sets are: the number of gaussian bells \( p \), the variance of each peak \( \sigma \), and a zipfian parameter \( z \) that regulates the total number of tuples contained in each gaussian bell.

—\textit{Array}: Each dimension has \( v \) distinct values, and the value sets of each dimension are generated independently. Frequencies are generated according to a zipfian distribution and assigned to randomly chosen cells in the joint frequency distribution matrix. The parameters for this data set are the number of distinct

\begin{table}[h]
\centering
\caption{Characteristics of the real data sets.}
\begin{tabular}{|l|c|c|l|}
\hline
\textbf{Data Set} & \textbf{Dim.} & \textbf{# of tuples} & \textbf{Attribute Names} \\
\hline
Census2D & 2 & 210,138 & \textit{Age}, \textit{Income}. \\
Census3D & 3 & 210,138 & \textit{Age}, \textit{Income}, \textit{Weeks worked per year}. \\
Cover4D & 4 & 545,424 & \textit{Elevation}, \textit{Aspect}, \textit{Slope}, \textit{Distance to roadways}. \\
\hline
\end{tabular}
\end{table}
attributes by dimension $v$ and the zipfian value for the frequencies $z$. When all the data points are equidistant, this data set can be seen as an instance of the Gauss data set with $\sigma = 0$ and $p = v^d$.

The default values for the synthetic data set parameters are summarized in Table II.

Finally, Figure 12 shows three examples of (two-dimensional) data sets used in our experiments. In the figure, each circle represents a tuple and its radius is proportional to the tuple’s frequency. Those circles are almost indistinguishable from points except in Figure 12(c), since the Array data set has considerable frequency skew.

### Table II. Default parameter values for the synthetic data sets.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Attribute</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>$d$: Dimensionality</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>$N$: Cardinality</td>
<td>500,000</td>
</tr>
<tr>
<td></td>
<td>$R$: Data domain</td>
<td>$[0 \ldots 10,000)^d$</td>
</tr>
<tr>
<td></td>
<td>$z$: Skew</td>
<td>1</td>
</tr>
<tr>
<td>Gauss</td>
<td>$p$: Number of peaks</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>$\sigma$: Standard deviation of each peak</td>
<td>100</td>
</tr>
<tr>
<td>Array</td>
<td>$v$: Distinct attribute values</td>
<td>60</td>
</tr>
</tbody>
</table>

![Fig. 12. Data sets.](image)

5.2 Histograms

We use Equi-Depth and MHist multidimensional histograms as our source of statistics about the data distributions. A multidimensional version of the Equi-Depth histogram [Piatetsky-Shapiro and Connell 1984] presented in [Muralikrishna and DeWitt 1988] recursively partitions the data domain, one dimension at a time, into buckets enclosing the same number of tuples. Reference [Poosala and Ioannidis 1997] introduced MHist based on MaxDiff histograms [Poosala et al. 1996]. The main idea is to iteratively partition the data domain using a greedy procedure. At each step, MaxDiff analyzes unidimensional projections of the data set and identifies the bucket in most need of partitioning. Such a bucket will have the largest “area gap” [Poosala et al. 1996] between two consecutive values along one dimension. Using this information, MHist iteratively splits buckets until it reaches the desired number of buckets. We refer the reader to [Muralikrishna and DeWitt 1988; Poosala and Ioannidis 1997; Poosala et al. 1996] for a detailed discussion of these techniques.
5.3 Workloads

For our experiments, we used workloads consisting of 100 queries each that follow two distinct query distributions, which are considered representative of user behavior [Pagel et al. 1993]:

— **Biased**: The query centers follow the data distribution, i.e., each query is an existing point in the data set. The probability that a point \( p \) in data set \( D \) is included in the workload is \( f_p / |D| \), where \( f_p \) is the frequency of \( p \) in \( D \).

— **Uniform**: The query centers are uniformly distributed in the data domain.

For each experiment, we generated two 100-query workloads. The first workload, the *training* workload, is used to find the optimal value of \( \alpha \) for the *Dynamic* strategy of Section 4. The second workload, the *validation* workload, is statistically similar to the first one, i.e., follows the same distribution, and is used to test the performance of the different mapping strategies.

Figure 13 shows two sample 100-query workloads for the Census2D data set.

5.4 Indexes

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 selection query. The tightness of the mapping depends on the mapping algorithms (Sections 3 and 4) and on their interaction with the quality of the available histograms. The efficiency of execution of the selection query depends on the indexes available on the database and on the optimizer’s choice of an execution plan.

To choose appropriate index configurations for our experiments, we first tried Microsoft’s *Index Tuning Wizard* over SQL Server 7.0 [Chaudhuri and Narasayya 1997], a tool that automatically determines good index configurations for a specific workload. We fed the Index Tuning Wizard with different data sets and representative query workloads for our task and it always suggested an \( n \)-column concatenated-key B\(^+\)-tree index covering all attributes in the top-k queries. Therefore, we focused on multi-column indexes in most of our experiments. 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. We do not consider clustered indexes in our

---

(a) *Census2D* data set.  (b) *Biased* workload.  (c) *Uniform* workload.

**Fig. 13.** Two different workloads for the *Census2D* data set.
experiments, since in a real situation the layout of the data could be determined by other applications.

For the \( n \)-column index configurations, we need to define the order in which the attributes would be concatenated to form the index search keys. We considered different choices and found that the attribute order is an important factor in the efficiency of the overall method. In fact, sometimes a poor choice of the multiattribute index results in even worse performance than that for when we just use unidimensional indexes. To determine attribute order in the multiattribute index we proceed as follows. We issue a small number of representative top-\( k \) queries \( q \) in the same way as we do for training the Dynamic mapping strategy. For each of those queries we determine the optimal range selection query \( C_q \) that tightly encloses \( k \) tuples, as described in Section 3.2. Then, for each attribute \( A_i \) in the data set, we find \( t_i \), the number of tuples that lie in the unidimensional projection of \( C_q \) (see Figure 14 for an example using the Census2D data set). A multiattribute index built with \( A_i \) in the first position will need to traverse the search keys of all \( t_i \) tuples in the projection of \( C_q \) (not just those corresponding to the tuples enclosed by \( C_q \)) when answering \( C_q \). Therefore, we sort the attributes in increasing number of \( t_i \). This configuration results in good performance for the kind of \( n \)-attribute range queries that our top-\( k \) processing strategy generates.

### 5.5 Evaluation Techniques

In our experiments, we compare our proposed mapping strategies of Sections 3 and 4 against each other, and also against other proposed approaches in the literature. Specifically, we will study the following techniques for answering top-\( k \) queries:

--- *Optimum technique:* As a baseline, we consider the execution of an ideal technique that results from enclosing the actual top \( k \) tuples for a given query as tightly as possible. 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.
Histogram-based techniques: The static and dynamic mapping strategies described in Sections 3 and 4.

Techniques requiring sequential scans: The techniques in [Carey and Kossmann 1997; 1998] 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 7. (We ignore this sorting step in our experiments, the same way we ignore it when evaluating the other techniques. This step can always be implemented by pipelining the retrieved tuples to a k-bounded priority queue, and its run time is negligible relative to the rest of the processing.) Therefore, we model this technique as a simple sequential scan of the relation, which is a lower bound on the time required by [Carey and Kossmann 1997; 1998]. 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 B$^+$-tree), rather than scanning the actual relation, which might be larger due to 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 20 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 [Carey and Kossmann 1997; 1998] would require to process a multiattribute top-k query like the ones we address in this paper.

5.6 Metrics

We will report experimental results for the techniques presented above using the following metrics:

- Percentage of restarts: This is the percentage of queries in the validation workload for which the associated selection query failed to contain the $k$ best tuples, hence leading to restarts. (See the algorithm in Section 3.) This metric makes sense only for the histogram-based mapping strategies of Sections 3 and 4, since by definition, the Optimum strategy and techniques requiring full sequential scans do not incur in restarts.

- Execution time, as a percentage of the sequential scan time: This is the average run time for executing all queries in the validation workload. We will present run times of the different techniques as a percentage of that of a sequential scan. We will discriminate the total execution time as:
  - SOQ (Successful Original Query) time: As we will see, in some cases the majority of top-k queries will not require restarts, so it is interesting to report their average run time separately from that of the small fraction of queries that require restarts.
  - IOQ (Insufficient Original Query) time: This is the average increase in time when also considering the queries in the workload that required restarts.

those queries the total execution time includes the running time for the original (insufficient) query (Retrieve step), plus the time for the subsequent “safe" query that retrieves all of the needed tuples using distance $d_{NR}$ (Restart step).

—Number of tuples retrieved, as a percentage of the number of tuples in the relation: This is the average number of tuples retrieved for all queries in the validation workload, as a percentage of the total number of tuples in the relation. Just as for execution time, we report $SOQ$ and $IOQ$ tuples retrieved.

6. EXPERIMENTAL RESULTS

This section presents experimental results for the top-$k$ processing techniques. 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 a 100-query Biased workload, multiattribute indexes with the attribute ordering as described in Section 5.4, $k = 100$ and $Max$ as the distance function. We report results for other settings of the parameters as well.

Section 6.1 studies the intrinsic limitations of our mapping approach. Section 6.2 compares the static techniques of Section 3 and the dynamic technique of Section 4. Section 6.3 studies the performance of different multidimensional histogram structures. Sections 6.4, 6.5, and 6.6 discuss the robustness of our Dynamic approach for various data distributions, distance functions, and values of $k$ in the top-$k$ queries, respectively. Section 6.7 analyzes the case when only unidimensional indexes are present. Section 6.8 compares our Dynamic strategy against a recently proposed technique that uses sampling instead of histograms to define the range query boundaries. Finally, Section 6.9 summarizes the evaluation results.

6.1 Validity of the General Approach

Our general approach for processing a top-$k$ query $q$ (Section 3) 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 distance selection strategies (Section 3), we first scanned each data set to find the actual top 100 tuples for a given query $q$, and determined a tight $n$-rectangle $T$ that encloses all of these tuples. We then computed the number of tuples in the data set that lie within rectangle $T$. Figure 15 reports the results. As we can see from the figure, the number of tuples that lie in this “ideal” rectangle is close to the optimal 100, and even in the worst case, for the Cover4D data set and the Sum distance function, the number of tuples corresponds to less than 0.4% of the 500,000-tuple data set. These results validate 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
Fig. 15. The number of tuples in the data set included in an $n$-rectangle enclosing the actual top-100 tuples.

6.2 Analysis and Comparison of the Techniques

This experiment compares the relative performance of the four static techniques of Section 3 and the Dynamic technique of Section 4. As we will see, the Dynamic technique always results in lower execution times than any of the static techniques of Section 3, which in turn are more efficient than a sequential scan over the relations.

Figures 16(a) and 17(a) show the execution time for different data sets for the Biased and Uniform workloads, respectively, over Microsoft SQL Server 7.0, as explained above. Each group of six bars corresponds to a different data set, reporting the percentage of time of a sequential scan taken by each of the six techniques, i.e., Optimum, Dynamic, Restarts, NoRestarts, Inter1 and Inter2. Each bar shows the SOQ and IOQ times as discussed in Section 5.6. Each technique has the associated percentage of restarts reported next to its name. For instance, in Figure 16(a) and for the Gauss data set, Dynamic results in 5% of restarts. Among the cases that did not restart (95%), the Dynamic technique uses only 6% of the time of a sequential scan. If we consider all cases, whether they needed restart or not, the Dynamic technique uses 7% of the time of a sequential scan. Analogously, the percentage of time of a sequential scan that the Restarts, NoRestarts, Inter1 and Inter2 strategies take for the same data set is 31%, 30%, 12%, and 21% respectively.

Figures 16(b) and 17(b) report the percentage of tuples retrieved for each scenario. In all cases, the static techniques result in better performance than a sequential scan. However, no one static strategy consistently outperforms the other static strategies. More precisely, if we do not consider the Optimum and Dynamic strategies in Figures 16(a) and 17(a), we see that Inter1 results in the best performance for the Biased workloads, but in general Inter2 is the best strategy for Uniform workloads. This can be explained in the following way. Usually, data sets form dense clusters and consequently they also contain several regions with very low tuple density. It is more likely for the Uniform workload to have queries that lie in such void areas. In contrast, queries from Biased workloads usually lie near the centers of the clusters, which are denser regions. Therefore, for Biased workloads, the optimal search distances are closer to Restarts than to NoRestarts, and

---

6This property does not hold in general for high numbers of dimensions [Beyer et al. 1999]. However, in this paper we focus only on low-to-moderate number of dimensions, mostly because of limited accuracy of state-of-the-art histograms for higher dimensions.

strategy Inter1 performs the best overall. In contrast, for Uniform workloads the situation is the opposite. The optimal search distances are closer to NoRestarts than to Restarts, and in general Inter2 is the most efficient technique among the static ones.

The Dynamic technique, due to its workload-adaptive nature, results in better performance than that of the static techniques across data sets and workloads. Dynamic needs less than 35% of the time of a sequential scan in all cases (for Biased workloads, it needs less than 10% of the time of a sequential scan). Figures 16(b) and 17(b) show that the percentage of tuples retrieved by Dynamic is always below 2%.

Generally, execution times for the Biased workloads are lower than those for the Uniform workloads. By an argument similar to that presented above, the average search distances produced by the techniques are larger for the Uniform than for the Biased workload mostly because we use (multiattribute) B-tree indexes: Although the number of tuples included in these larger selection queries is small, the query processor still has to traverse several search keys with associated tuples that might lie far away in the multidimensional space (see Figure 14). In contrast, queries in Biased workloads tend to have lower associated search distances, which results in fewer search keys traversed and lower execution times.

Since our Dynamic technique never results in higher execution times than any of the static techniques of Section 3, we focus on the Dynamic mapping strategy for the rest of the paper.

6.3 Effect of Multidimensional Histograms

Figure 18(a-d) shows execution times and the percentage of tuples retrieved for different data sets and for different multidimensional histograms. With the only exception of the Gauss data set and Biased workloads in Figures 18(a) and 18(b), the results are significantly better when we use Equi-Depth histograms than when we use MHist histograms to guide our mapping strategy. Also, the number of tuples retrieved is sometimes as much as 10 times larger for MHist histograms than for Equi-Depth histograms (Cover4D data set in Figure 18(d)). As noted in [Bruno et al. 2001], MHist histograms generally devote too many buckets to the densest tuple clusters in the data sets, and almost none to the rest of the data domain,
which tends to degrade the overall histogram accuracy. $MHist$ histograms have buckets with very heterogeneous tuple density, which degrades the performance of our technique. For that reason, we focus on $Equi-Depth$ histograms for the rest of the discussion. It is important to note that our techniques are flexible enough so that other recent multidimensional histogram structures (e.g., [Bruno et al. 2001; Gunopulos et al. 2000]) can be exploited without changes in the proposed framework.

6.4 Robustness across Data Sets

To analyze the robustness of our Dynamic strategy, we started with the default synthetic data sets, and varied their skew and dimensionality.

Figure 19 shows that as the data skew $z$ increases, the total time taken to answer top-$k$ queries also increases slowly relative to the time required by a sequential scan. For the Array data set, the optimum execution time and percentage of retrieved tuples increases sharply with $z$. For $z = 2$, the most frequent tuple is repeated 17% of the time, and the Biased workload picks this tuple with high probability. In those cases, there is no choice but to return all the repeated tuples as the top-$k$ ones, thus increasing the processing time of any strategy.

Figure 20 shows that the execution time of our technique increases moderately as the dimensionality of the data set increases. The unexpected peak in the tuples retrieved for the Array data set for $d = 2$ can be explained as follows. In two dimensions, the Array data set behaves similarly as the three-dimensional Array data set with $z = 2$ (see above). In fact, the combined frequency of the five most popular tuples accounts for more than 15% of the whole data set. The Biased workload frequently picks those tuples as queries, which results in all tuples with the same values being retrieved as well. Since the available storage for the histograms is fixed in our experiments, histograms for high-dimensional data sets become coarser and less accurate, which impacts negatively on the efficiency of our technique. However, it is important to note that even for four dimensions, the time taken by our Dynamic technique is below 20% of the time of a sequential scan in all our experiments. Also, the percentage of tuples retrieved is below 5% and the percentage of queries that need restarts remains low at at most 7% in all cases.

6.5 Effect of the Distance Function

In this experiment we measure the performance for different distance functions and for different data sets. Not surprisingly, we see in Figure 21 that the Max distance function performs the best overall, followed by Eucl and Sum. As we discussed in Section 3.2, the region of all tuples at Max distance $d$ or lower from a query $q$ is already an $n$-rectangle, so the range selection query $C_q$ does not retrieve any (useless) tuple at distance higher than $d$. In contrast, the regions defined by the
Eucl and Sum distance functions are not rectangular, so in general we have no choice but to retrieve some extra tuples at distance higher than $d$ (Figure 1). Unfortunately, this negative effect gets worse as the data set dimensionality increases since the ratio between the volume of the region of all possible tuples at distance $d$ or lower from $q$ and the volume of the tight $n$-rectangle that encloses such region decreases as the number of dimensions increases. Figure 21(b) shows that for the two-dimensional data set Census2D the difference in performance among distance functions is minimal. In contrast, for Cover4D (four dimensions) we have a significant increase in the percentage of tuples retrieved, which in turn affects the execution time. However, the percentage of tuples retrieved is below 5% in all our experiments.

6.6 Effect of the Number of Tuple Requested $k$

Figure 22 reports execution times and the percentage of tuples retrieved as a function of $k$, the number of tuples requested. Our technique is robust for a wide range of $k$ values. Even when queries ask for the top-1000 tuples, the execution time is less than 25% of the time of a sequential scan. We can also see that the percentage
of restarts increases with $k$. This can be explained as follows. For “expensive” restarts, our Dynamic strategy will choose a high value of $\alpha$, which in turn will make restarts rare, thus minimizing execution times. Conversely, if restarts are inexpensive, our Dynamic strategy will choose a lower value of $\alpha$: Although restarts will then be more likely, they will contribute less to the total execution cost. In our specific case, when $k$ increases from 50 to 1,000, the $dNR$ distance produced by the NoRestarts strategy in the Verify/Restart step in Section 3 remains almost unchanged, since there are not many new buckets needed to guarantee the higher values of $k$ (this effect is related to the granularity of the histogram’s buckets). On the other hand, the execution time and number of tuples retrieved for the cases that do not need restarts do increase, therefore restarts become relatively less expensive. Our Dynamic strategy ultimately chooses lower values of $\alpha$, which results in higher percentages of restarts but in generally lower execution times.

### 6.7 Effect of Index Configurations

Figure 23 shows how our technique performs in the absence of multiattribute indexes. In this experiment we constructed a one-column unclustered B$^+$-tree index for each attribute mentioned in the query (Section 5.4). The resulting performance is 1.5 to 8 times worse than that for multiattribute indexes (the percentage of retrieved tuples remains the same). However, even when only unidimensional indexes are available, the total execution time was found to be below 60% of that of a single sequential scan over the relation in all our experiments.

### 6.8 Comparison with Sampling-Based Techniques

Recently, reference [Chen and Ling 2002] modified our strategies in [Chaudhuri and Gravano 1999] to use sampling rather than multidimensional histograms to evaluate top-$k$ queries over relational databases. For each incoming query, a range selection query that is expected to cover most of the top-$k$ tuples is constructed and evaluated. However, instead of using multidimensional histograms to define the range selection query, reference [Chen and Ling 2002] uses sampling. In particular, a uniform sample of the data set is kept in memory, and is used to define the boundaries of the corresponding range selection query. The query model that is used is slightly different from ours, with no restarts. In effect, when using sampling
it is not possible to guarantee that at least $k$ tuples will be retrieved. Therefore, the result of the selection query in [Chen and Ling 2002] serves as an approximate answer to the original top-$k$ query, and the experimental evaluation focuses on the precision and recall of the query mapping strategies.

In this section, we compare our Dynamic technique against Para, an adaptive sampling-based technique proposed in [Chen and Ling 2002]. In particular, for each experiment we generate a uniform sample of the data set and tune Para as described in [Chen and Ling 2002] so that it results in 100% recall (corresponding to the exact answer). As explained before, the Para technique can return fewer than $k$ tuples in some situations, even when tuned for 100% recall. In those rare cases, we perform a sequential scan over the data to retrieve the remaining tuples, since this is the only way to guarantee correct results in our query model.

Figure 24 shows the execution times of the Dynamic and the Para techniques for different data sets and both Uniform and Biased workloads. For a fair comparison, the sample for Para uses the same amount of memory as the histograms for Dynamic do, which results in twice as many sample tuples in Para compared to the number of buckets in the histograms. We can see in Figure 24 that the resulting execution times are comparable for both techniques. In fact, there is at most a 5% difference in execution times between Dynamic and Para. This is not surprising, since both techniques are based on the same underlying approach, differing only in the specific model used to approximate the data distribution (i.e., sampling and multidimensional histograms). These results should not be taken as definitive since both techniques can be improved. On one hand, our Dynamic techniques can take advantage of more accurate multidimensional histograms (e.g., [Bruno et al. 2001; Gunopulos et al. 2000]). On the other hand, more accurate sampling techniques (e.g., stratified or weighted sampling) can be used to enhance the accuracy of Para. It is our belief that techniques based on novel multidimensional histograms would be more accurate than sampling-based techniques when the number of dimensions in the data sets is fairly moderate (as it is the case for traditional selectivity estimation of range queries). On the other hand, we believe that sampling-based techniques should scale better than histogram-based techniques for high data dimensionality, as reported experimentally in [Chen and Ling 2002].
Top-k Selection Queries over Relational Databases

Fig. 24. Execution time for histogram- and sampling-based techniques.

6.9 Evaluation Conclusions

In previous sections we evaluated our Dynamic strategy for different data distributions, workloads, histogram and index configurations, and query parameters. We compared Dynamic with the static mapping strategies of Section 3 and also with techniques that require a sequential scan over the relation. In summary, our Dynamic technique was found to be more efficient than the histogram-based alternatives and it always resulted in lower execution times than a single sequential scan over the relation. In particular, when we used Equi-Depth histograms and multiattribute indexes, the Dynamic technique took around 4% to 22% of the time of a sequential scan for Biased workloads, and around 10% to 40% of the time of a sequential scan for Uniform workloads. Although performance degrades when only single attribute indexes are available, execution times are still below two thirds of that of a sequential scan in those cases. Finally, the number of tuples retrieved by our Dynamic strategy was found to be in all cases less than 6% of the relation size.

In the experiments of Section 6.7, we used the existing index structures in SQL-Server 7.0, i.e., single- and multi-column B+-trees. It is important to note that, whenever new access methods (e.g., R-trees) are incorporated to a RDBMS and fully integrated to its query optimizer, our techniques will take advantage of these new index structures without any changes in the proposed framework. For a given range query, the optimizer is responsible for choosing the query plan and index configuration that would result in the most efficient execution. Therefore, the support of new index structures in a RDBMS can only improve the execution time of our techniques. Of course, if the added access method natively supports nearest-neighbor searches, then our technique is no longer necessary.

7. RELATED WORK

Motro [Motro 1988] 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.

Carey and Kossman [Carey and Kossmann 1997; 1998] 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 [Carey

and Kossmann 1997; 1998]. Hence, these strategies require a preprocessing step to

compute the distance function itself involving one sequential scan of all the data.

Donjerkovic and Ramakrishnan [Donjerkovic and Ramakrishnan 1999] 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 [Don-

jerkovic and Ramakrishnan 1999] involves a single attribute, while the core of our

contribution is dealing with multiattribute conditions without assuming indepen-
dence among the attributes, for which we exploit multidimensional histograms.

Recently, Chen and Ling [Chen and Ling 2002] modified our strategies in [Chaud-
huri and Gravano 1999] for evaluating top-k queries over relational databases. In-

stead of using multidimensional histograms to define the range selection query that

is expected to cover most of the top-k tuples, the authors use sampling. The query

model that is used is slightly different from ours, with no restarts. Instead, the

result of the selection query serves as an approximate answer to the original top-k

query. Therefore, the experimental evaluation focuses on the precision and recall of

the proposed query mapping strategies. (See Section 6.8.) Also recently, Hristidis

et al. [Hristidis et al. 2001] presented PREFER, a prototype that uses multiple ma-

terialized views to efficiently answer preference queries, which return the k tuples

that maximize a given linear function over the relation’s attributes.

Multidimensional density estimation is an active research field. The main tech-
niques comprise sampling [Olken and Rotem 1990], wavelets [Matias et al. 1998],
fractal dimension concepts [Faloutsos and Kamel 1997; Belussi and Faloutsos 1995],
and multidimensional histograms. Multidimensional histogram construction shares
some intriguing features with multidimensional access method construction. Mul-
tidimensional access methods [Gaede and Günther 1998] support efficient search
operations in spatial databases. They partition the data domain into buckets, and
assign to each bucket some information about the tuples it covers (usually the set
of rids). There is a connection between access methods and histogram techniques
regarding the different ways in which they partition the data domain. For instance,
the partitioning strategy used in STGrid histograms [Aboulnaga and Chaudhuri
1999] is similar to that of the Grid File technique [Nievergelt et al. 1984]. MHist
histograms [Poosala and Ioannidis 1997] share the hierarchical or recursive parti-

a similar partitioning scheme to hB-tree’s holey bricks [Lomet and Salzberg 1990]. Finally, GenHist histograms use overlapping buckets, just as R-trees [Guttman 1984] do. A natural question arises then. Why not use existing access methods directly to model density distributions of data sets? This idea is explored for example in the CONTROL project [Avnur et al. 1998], which uses the shell of R-trees to provide online visualization of large data sets, by traversing the R-tree breadth-first and approximating the underlying data distribution with the aggregated information at each level. In spite of these connections between histograms and access methods, we believe that there are fundamental differences between the two. The main goal of multidimensional access methods is to allow efficient access to each “bucket” using only a few disk accesses, so the main objective is to distribute tuples evenly among buckets and maintain a high fraction of bucket utilization to prevent long searches. On the other hand, histogram techniques need to form buckets enclosing areas of uniform tuple density whenever possible, so that the techniques that assume uniformity inside buckets work as expected.

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., [Korn et al. 1996]) 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 [Seidl and Kriegel 1998] further refines this idea.) This approach is conceptually similar to that in this paper (and also in [Chaudhuri and Gravano 1996]), where we first find a suitable distance \(D\), 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.” Algorithms and data structures specifically designed to answer top-\(k\) queries are expected to result in more efficient executions than our techniques. Integrating these algorithms into today’s complex and performance-sensitive RDBMSs is challenging even with the support for extensibility available in modern database servers. In contrast, our strategies can be easily implemented as a thin layer on top of an existing RDBMS, and can benefit from the inclusion into the RDBMS of new and more accurate histograms (since they can be plugged in without changes to our framework), and additional access methods, as explained in Section 6.9.

8. CONCLUSIONS

In this paper, we studied the problem of mapping a multiattribute top-\(k\) selection 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 algorithms exploit histograms and are able to cope with a variety of scoring functions. 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 mapping top-\(k\) queries to multiattribute range queries that are
“tuned” to a given workload reduces the probability of restarts while ensuring that the required top-k matches are returned. Our techniques are robust, and perform significantly better than existing strategies requiring at least one sequential scan of the data sets.

A key property of our techniques is that they can be easily implemented on top of existing RDBMSs with minimal changes to the existing interfaces. Therefore, our techniques can automatically benefit from advanced multidimensional access methods as they become integrated into RDBMSs, since the underlying query optimizers will exploit such access methods for the execution of the selection queries that we produce. Furthermore, as discussed in Section 6.9, our techniques could be adapted to benefit from multidimensional access methods as a complement to (or even replacement of) the statistical information provided by multidimensional histograms. A deeper analysis of these ideas is subject of intriguing future work.

REFERENCES


Received June 2001; revised March 2002; accepted April 2002