Answering Top-k Queries Using Views

Gautam Das
University of Texas
gdas@cse.uta.edu

Dimitrios Gunopulos
University of California Riverside
dg@cs.ucr.edu

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

Dimitris Tsirogiannis
University of Toronto
dimitris@cs.toronto.edu

ABSTRACT

The problem of obtaining efficient answers to top-k queries has attracted a lot of research attention. Several algorithms and numerous variants of the top-k retrieval problem have been introduced in recent years. The general form of this problem requests the k highest ranked values from a relation, using monotone combining functions on (a subset of) its attributes.

In this paper we explore space performance tradeoffs related to this problem. In particular we study the problem of answering top-k queries using views. A view in this context is a materialized version of a previously posed query, requesting a number of highest ranked values according to some monotone combining function defined on a subset of the attributes of a relation. Several problems of interest arise in the presence of such views. We start by presenting a new algorithm capable of combining the information from a number of views to answer ad hoc top-k queries. We then address the problem of identifying the most promising (in terms of performance) views to use for query answering in the presence of a collection of views. We formalize both problems and present efficient algorithms for their solution. We also discuss several extensions of the basic problems in this setting.

We present the results of a thorough experimental study that deploys our techniques on real and synthetic data sets. Our results indicate that the techniques proposed herein comprise a robust solution to the problem of top-k query answering using views, gracefully exploring the space versus performance tradeoffs in the context of top-k query answering.

1. INTRODUCTION

Providing efficient answers to top-k (ranking) queries has been an active research topic. Such queries aim to retrieve quickly a number (k) of highest ranking tuples in the presence of monotone ranking functions defined on the attributes of underlying relations. Several algorithms have been proposed for a large number of variants of this basic problem, the most well-known being the *Threshold Algorithm* (TA) proposed by Fagin et.al., [19] and independently by Guntzer et. al., [11] and Nepal et. al., [21]. Adaptations of these algorithms in a relational context have been recently studied (see [20] and references therein).

VLDB '06, September 12-15, 2006, Seoul, Korea.

f3 = 3x1+10x2+5x3

R	tid	X1	X2	X3
	1	82	1	59
	2	53	19	83
	3	29	1	2
	4	80	22	90
	5	28	8	87
	6	12	55	82
	7	16	99	42
	8	18	42	67
	9	42	1	23
	10	23	21	88

f1 = 2x1+5x2

V1	tid	score
	7	527
	6	299
	4	270
	8	246
	2	201

f2 = x2+2x3

V2	tid	score
	6	219
	4	202
	10	197

Figure 1: Example Views

In relational query processing, the problem of answering queries using views has received significant attention due to its relevance to a variety of data management problems [13]. The problem here is to find efficient methods of answering a query using a set of previously defined materialized views over the database. One of the advantages of using views for query answering is the promise of increased performance. For example, views may be of small size, so if an answer can be obtained by processing data from one or more views the answer could be obtained much faster. With performance as the parameter of interest in mind, views offer a space/performance tradeoff to query answering. Views are materialized (incurring a space overhead) with the hope to gain on performance for some queries.

In this paper we focus on the problem of answering top-k queries using views. In this context a view is defined as a materialized form of a top-k query asked previously. In the presence of such views several problems of interest arise. Consider Figure 1. It presents a three attribute relation R and two views V_1 and V_2. Both views are defined on subsets of attributes of R. To simplify our example, assume that both views are defined using top-k queries expressed on the entire relation R. Thus, the views do not specify any (range) selection conditions on the attributes they aim to rank. The first view is defined as a result of a top-5 query using function f_1 and contains 5 tuples of R. The second is defined as the result of a top-3 query using function f_2 and contains 3 tuples of R. Given a top-2 query defined using function f_3, thus requesting the 2 highest ranking tuples of R according to f_3, we can always apply a standard top-k algorithm (e.g., TA [19], assuming that the right access paths are available as in [2]) using data from R and obtain an answer to the query. A natural question arises regarding the feasibility

of obtaining an answer to this query using views V_1 and V_2. For example one may be able to use data in conjunction from views V_1 and V_2, in order to obtain an answer to the query. Clearly we need to identify whether the use of the views can guarantee an answer to the top-k query. Moreover, even if this is the case, we would like to proceed using the views, only if an answer to the query can be obtained much faster using the views as opposed to utilizing data of R directly. Considering this problem in a more general setting, we would like to address a *view selection* problem, namely given a collection of views $\mathcal{V}$, we would like to identify efficiently the set $\mathcal{U} \subseteq \mathcal{V}$ of views to use in order to efficiently provide an answer to an incoming query.

In this paper, we place such problems into perspective formalizing them and we make the following contributions:

- We propose *LPTA*, an algorithm that given a set of views and a top-k query utilizes the set of views to produce an answer to the query. This algorithm is a nontrivial adaptation of the TA algorithm, requiring the solution of linear programming optimizations in each iteration.
- We present an analysis of the *view selection* problem, formalizing it in the case of two attribute relations for arbitrary data distributions and extend our analysis to the case of multi-attribute relations as well.
- Inspired by our analysis, we present a cost estimation framework that enables reasoning about the *cost* of answering a query given a set of views. We subsequently utilize this framework in order to identify efficiently a solution to the view selection problem. We subsequently extend this framework to other forms of top-k queries and views.
- We present the results of a detailed experimental evaluation utilizing both real and synthetic datasets highlighting the benefits of our overall proposal.

The rest of this paper is organized as follows. We discuss related work in Section 2. In Section 3 we present formal definitions of the problems considered in this paper. In Section 4 we discuss the *LPTA* algorithm. Section 5 is devoted to a conceptual analysis of the view selection problem, while Section 6 describes a practical solution that includes a cost estimation framework. We discuss further variations to our problem framework in Section 7. Section 8 presents the results of our experiments, while we conclude in Section 9.

2. RELATED WORK

Originally, top-k (ranking) queries have been proposed in a multimedia context [8, 6, 7], where the aim is to produce a number of highest ranking results from a set of ordered lists, according to monotone ranking functions defined on the elements of the lists. Each list consists of a tuple identifier and an attribute value and is arranged in non increasing order of that value. Each tuple identifier is assigned a *score* according to the ranking function computed on the attribute values of the associated list and the objective is to identify the k tuples with the highest scores.

The threshold algorithm (TA) constitutes the state of the art for top-k query answering [21, 11, 19]. The TA algorithm accesses list items in lock-step, traversing each list in a sequential fashion. For each tuple identifier encountered in the sequential access, it immediately probes (via random access) the remaining lists for their scores and thus the full score of the tuple identifier is immediately known to the algorithm. This algorithm has a deterministic stopping condition and once encountered it terminates with the correct top-k set. Adaptations of such an algorithm to work on top of relations have been studied as well (see references in [20]). In such a scenario, lists are essentially simulated via suitable access paths per relational attribute. Sequential ordered access is conducted via an iterator interface on the suitable access path. Random access is conducted via indexed random access on the tuple identifier. Notice that in this case only a single random access per tuple identifier is necessary to resolve all attributes of a tuple identifier. This is because the underlying data representation is a relation and a single random access on the tuple identifier provides access to all attribute values of the tuple.

Several variants of the basic ideas of the TA algorithm have been proposed in the literature. In one variant (TA-Sorted) [19, 11] lists are always accessed sequentially. No random accesses are performed and thus at any point the score of a tuple identifier is partially known. Variants of the basic top-k problem have been considered in a web context [3, 1], in a relational database context [9, 17] as well as on join scenarios [15, 18, 16]. Others considered nearest neighbor type of approaches for this problem [5, 4, 22].

Hristidis et. al., [14] studied the problem of supporting top-k queries on a relation R, utilizing a relational database, by storing multiple copies of R each ordered according to a different ranking function. The particular problem studied was highly restrictive, as it assumed that only one copy of a relation could be utilized to obtain an answer to a new query. Due to this assumption, the entire relation copy should be available, not some prefix of the ranking suitably restricted to a fixed number of tuples (as it would be the case if that was the result of a top-k query). This is required in order to guarantee that a query answer can always be extracted from the relation copy. The basic idea in the proposed *PREFER* algorithm is, given a ranking function supplied by the query and a ranking function according to which a relation copy R_C has been ordered, to obtain the maximal distance from the beginning of R_C that the answer to the top-k query should be located, using suitable upper bounds of the domain of each attribute in the relation instance. Then, by accessing R_C sequentially, evaluating the query ranking function on the tuples of R_C, the answer to the top-k query can be obtained. An additional restrictive assumption is that all attributes of R are always utilized for all top-k queries.

3. PRELIMINARIES

3.1 Top-k Queries and the Threshold Algorithm

Consider a single relation R with m numeric attributes $X_1, \ldots, X_m$, and n tuples $t_1, \ldots, t_n$. Let $Dom_i = [lb_i, ub_i]$ be the domain of the i-th attribute. We will refer to table R as a base table. Each tuple t may be viewed as a numeric vector $t = (t[1], t[2], \ldots, t[m])$. Each tuple is associated with a tuple-id (tid). In this paper we consider top-k *ranking queries*, which can be expressed in SQL-like syntax: SELECT TOP $[k]$ FROM R WHERE $Range_Q$ ORDER BY $Score_Q$. More abstractly, a ranking query may be expressed as a triple $Q = (Score_Q, k, Range_Q)$, where $Score_Q(t)$ is a function that assigns a numeric score to any tuple t (the function does not necessarily involve all attributes of the table), and $Range_Q(t)$ is a Boolean function that defines a selection condition for the tuples of R in the form of a conjunction of range restrictions on $Dom_i, i \in \{1, \ldots, m\}$. Each range restriction is of the form $l_i \leq X_i \leq u_i, i \in \{1, \ldots, m\}$ and the interval $[l_i, u_i] \subseteq Dom_i$. The semantics requires that the system retrieve the k tuples with the top scores satisfying the selection condition. Efficient algorithms for top-k retrieval have been the subject of much recent research, the best known being the *Threshold Algorithm* (TA) and its many

variants [19, 8]. This algorithm requires that the scoring function should be *monotonic*; in our case, this means that if we consider two tuples t and u such that $t[i] \leq u[i], 1 \leq i \leq m$, then $Score_Q(t) \leq Score_Q(u)$. In our paper we especially focus on *additive* scoring functions, i.e., where $Score_Q(t) = w_1 t[1] + w_2 t[2] + \ldots w_m t[m]$, where each w_i is a positive constant. Additive scoring functions are monotonic and have been widely adopted in the context of such algorithms. The TA algorithm requires that each attribute has an index mechanism that allows all tuple-ids to be accessible in *sorted order*. This implies that for each attribute X_i, all tuple-ids can be retrieved one by one in descending order of their X_i values. Given any tuple-id, the entire tuple can be efficiently retrieved - this access mechanism is called *random access*. Due to space limitations, we do not review the TA algorithm in more detail here but we refer the reader to the vast bibliography on the subject (e.g., see [19][2] and references therein).

3.2 Ranking Views

The main thrust of this paper is to investigate whether ranking queries can be answered more efficiently than the standard TA algorithm, if we are allowed to also leverage the presence of *materialized ranking views*. Informally, a materialized ranking view V is the materialized result of the tuples of a previously executed ranking query Q, ordered according to the scoring function $Score_Q$. More formally, for a previously executed query Q' = ($Score_{Q'}$, k', $Range_{Q'}$), the corresponding materialized ranking view $'$ is a set of k (tid, $score_Q(tid)$) pairs, ordered by decreasing values of $score_Q(tid)$. Henceforth, we refer to ranking queries and materialized ranking views simply as *queries* and *views* respectively. Given a relation R with m attributes, we assume availability of the views $V_{X_i}, 1 \leq i \leq m$. Each of these views is ordered according to the scoring function $\sum_{j=1}^{m} w_j t[j]$ in which $w_j = 0, j \neq i$ and 1 otherwise. We refer to the set of views V_{X_i} as *base views*. The standard TA algorithm can utilize such views directly. Each of these views serves as a "list" on which the algorithm has sequential access. One random access per tuple identifier encountered is required to fully resolve the score of an identifier however. This is because tuples are available in base relation R (on which random access is available using a tuple identifier). This is in contrast to the original TA algorithm description requiring $m - 1$ accesses per tuple identifier encountered using m lists.

The following problems constitute the main focus of this paper:

PROBLEM 3.1 (TOP-k QUERY ANSWER USING VIEWS). *Given a set $\mathcal{U}$ of views, and a query Q, obtain an answer to Q combining all the information conveyed by the views in $\mathcal{U}$.*

We propose an algorithm named *LPTA* to solve this problem. The second problem we address in this paper is the *view selection* problem.

PROBLEM 3.2 (VIEW SELECTION). *Given a collection of views $\mathcal{V} = \{V_1, \ldots V_r\}$ that includes the base views (thus $r \geq m$) and a query Q, determine the most efficient subset $\mathcal{U} \subseteq \mathcal{V}$ to execute Q on.*

Such a subset $\mathcal{U}$ will be provided as input to *LPTA*. Notice that since $\mathcal{V}$ contains the base views, we can always obtain an answer to an arbitrary ranking query by running TA. We seek to identify subset $\mathcal{U}$ that when utilized by *LPTA* can solve the problem faster (for suitably defined performance metrics). This problem is challenging because it is not always possible to obtain an answer to a query using a set of views, other than the base views. As a simple example consider a query asking for the, say, 10 highest ranking tuples; however each of the views specified to participate in the query answer contain fewer than 10 tuples each. This situation may become more complicated however. Depending on the ranking functions of the query and the views, even if the views contain more than 10 tuples, it is not always the case that the query answers can be obtained from views, other than the base views. Thus, the view selection problem should identify a set of views that can both provide an answer to the query and at the same time provide the answer faster than running TA on the base set of views, if possible.

We structure the rest of the paper as follows. We begin our presentation by solving an important variation of Problem 3.1, in which each view $V \in \mathcal{U}$ is of the form $V = (Score_V, n, *)$. Thus, each view V represents an ordering of *all* the database tuples according to $Score_V$. However, the query Q at hand may be of the form $Q = (Score_Q, k, *)$. We provide an algorithm for this problem in Section 4. Section 5 presents analytical reasoning for the view selection problem. In section 6 we present an algorithm for the solution of problem 3.2. Section 7 discusses our solution for views and queries of the form $V' = (Score_{V'}, k', *)$ and $Q = (Score_Q, k, *)$ respectively, as well as our solution for the most general case, i.e., where views and queries are of the form $V' = (Score_{V'}, k', Range_{V'})$ and $Q = (Score_Q, k, Range_Q)$ respectively.

4. LPTA: LINEAR PROGRAMMING ADAPTATION OF THE THRESHOLD ALGORITHM

In this section we shall describe Algorithm *LPTA* which is a linear programming adaptation of the classical TA algorithm to solve Problem 3.1 for the special case when views and queries are of the form $V' = (Score_{V'}, n, *)$ and $Q = (Score_Q, k, *)$ respectively. We shall first illustrate our adaptation using a simple example, and then follow it up with a formal description.

Consider a relation with attributes X_1, X_2 and X_3 as shown in Figure 1. Let views V_1 and V_2 have scoring functions f_1, f_2 respectively as shown in Figure 1 and consider a query $Q = (f_3, k, *)$. The algorithm initializes the top-k buffer to empty. It then starts retrieving the tids from the views V_1, V_2 in a lock-step fashion, in the order of decreasing score (w.r.t. the view's scoring functions). For each tid read, the algorithm retrieves the corresponding tuple by random access on R, computes its score according to the query's scoring function f_3, updates the top-k buffer to contain the top-k largest scores (according to the query's scoring function), and checks for the stopping condition as follows: After the dth iteration, let the last tuple read from view V_1 be (tid_d^1, s_d^1) and from view V_2 be (tid_d^2, s_d^2). Let the minimum score in the top-k buffer be $topk_{min}$. At this stage, the unseen tuples in the view have to satisfy the following inequalities (the domain of each attribute of R of Figure 1 is $[1, 100]$):

$$0 \leq X_1, X_2, X_3 \leq 100 \quad (1)$$

$$2X_1 + 5X_2 \leq s_d^1 \quad (2)$$

$$X2 + 2X3 \leq s_d^2 \quad (3)$$

This system of inequalities defines a convex region in three dimensional space. Let $unseen_{max}$ be the solution to the linear program where we maximize the function $f_3 = 3X_1 + 10X_2 + 5X_3$ subject to these inequalities. It is easy to see that $unseen_{max}$ represents the maximum possible score (with respect to the ranking query's scoring function) of any tuple not yet visited in the views. The algorithm terminates when the top-k buffer is full and $unseen_{max} \leq topk_{min}$. Considering the example of Figure 1 the

algorithm will proceed as follows; first retrieve tid 7 from V_1 and conduct a random access to R to retrieve the full tuple and tid 6 from V_2 accessing R again. The top-2 buffer contains the following pairs (tid_d^i, s_d^i) $\{(7, 1248), (6, 996)\}$. The solution to the linear program with $s_q^1 = 527$ and $s_d^2 = 219$ yields an $unseen_{max} = 1338 > topk_{max} = 1248$ and the algorithm conducts one more iteration. This time we access tid 6 from V_1 and tid 4 from V_2. The top-2 buffer remains unchanged and the linear program is solved one more time using $s_d^1 = 299$ and $s_d^2 = 202$. This time, $unseen_{max} = 953.5 < topk_{max} = 1248$ and the algorithm terminates. Thus, in total *LPTA* conducts two sequential and two random accesses per view. In contrast, the TA algorithm executed on R of Figure 1 will identify the correct top-2 results after 12 sorted and 12 random accesses in total. The performance advantage of *LPTA* is evident.

Intuitively, the algorithm will stop early if the scoring function of the views is "similar" to the scoring function of the query - this crucial issue is discussed in more detail in subsequent sections. Algorithm 1 gives the pseudo-code for *LPTA* generalized for multiple views. The algorithm starts by performing sequential and random accesses like TA to fill the top-k buffer ($topk$-Buffer). Tuples update the current top-k buffer as their score becomes available. At each step the solution to the linear program provides new information and the stopping condition is tested. The algorithm terminates when the stopping condition is satisfied. It is evident that *LPTA* is more general than the TA algorithm, in the sense that *LPTA* essentially becomes the TA when the set of views $\mathcal{U}$ provided is equal to the set of base views.

It terms of execution "cost", notice that both the *LPTA* and TA algorithms have sequential and random access patterns. These I/O operations play a significant role in determining the execution efficiency of the algorithms - in fact, as our experiments show, they overshadow the costs of CPU operations such as updating the top-k buffer, testing for stopping condition (even the solution of the linear programs), and so on. It is evident that in both algorithms sequential and random access patterns are highly correlated; every sequential access incurs a random access. For *LPTA* every tuple identifier encountered in a view via a sequential access prompts a random access to the base relation to resolve its attributes and compute the score according to the query. Similarly for TA every identifier encountered via sequential access prompts a random access to the base relation as well. As a result the determining factor for the performance advantage of an algorithm is the the distance from the beginning of the view (relation) each algorithm has to traverse (read sequentially) before coming into a halt with the correct answer, multiplied by the number of views participating in the process. In the case of Algorithm 1, if d is the number of lock-step iterations, the running cost is therefore $O(dr')$.

Algorithm 1 $LPTA(\mathcal{U}, Q)$

```
U = {V_1, ..., V_r'} // Set of views
Q = (Score_Q, k, *) // Query
topk-Buffer = {}
topk_min = 0
for d = 1 to n do
  for all views V_i (1 ≤ i ≤ r') in lock-step do
    Let (tid_d^i, s_d^i) be the d-th item in V_i
    // Update topk-Buffer
    Let t_d^i = RandomAccess(tid_d^i)
    if Score_Q(t_d^i) > topk_min then
      if (|topk-Buffer| = k) then
        Remove min score tuple from topk-Buffer
      end if
      Add (tid_d^i, Score_Q(t_d^i)) to topk-Buffer
      topk_min = min score of topk-Buffer
    end if
    //Check stopping condition by solving LP
    Let Unseen = convex region defined by
     lb_j ≤ X_j ≤ ub_j ∀ 1 ≤ j ≤ m
     Score_{V_j} ≤ s_d^j ∀ 1 ≤ j ≤ r'
    Compute Unseen_max = max_{t∈Unseen}{Score_Q(t)}
    if (|topk-Buffer| = k) and (unseen_max ≤ topk_min) then
      Return topk-Buffer
    end if
  end for
end for
```

5. THE VIEW SELECTION PROBLEM: A CONCEPTUAL DISCUSSION

This section is devoted to a conceptual discussion of the view selection problem. We first graphically illustrate various view selection scenarios of the problem for two attribute relations (in two dimensions). We then present formal results that hold for multi-attribute relations (for any dimension). Without loss of generality assume that the domain of each attribute of an m-attribute relation is normalized to $[0, 1]$. We refer to an m-attribute relation as an m-dimensional database.

5.1 View Selection in Two Dimensions

Consider Figure 2. The unit square $OPRT$ contains all the points of a two dimensional database. Assume that two views V_1 and V_2 have been materialized (in addition to the base views, i.e., $\mathcal{V} = \{V_1, V_2, V_X, V_Y\}$). In the figure, these two views are represented by vectors denoting the respective directions of increasing score - thus the views themselves are essentially materialized lists of tids, sorted according to their projections on these vectors. The direction of increasing score of a query Q's scoring function is also represented by a vector. Note that for this example, both view vectors are to the same side (clockwise) of the query vector.

Let M be the tuple with the k-th largest score in the database. Let AB be the line segment perpendicular to the query vector that passes through M and intersects the unit square. Thus, the top-k tuples are contained in the triangle ABR, while the remaining tuples are contained in the polygon $ABPOT$.

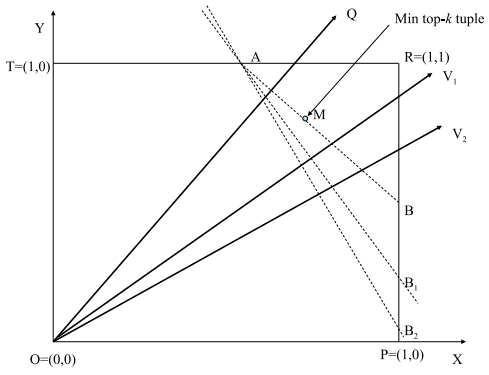

Figure 2: Selecting a view from same side of query

Now let us assume that only one of the views, V_1, is available for

use by the *LPTA* algorithm in answering the query. The execution of *LPTA* will perform sorted accesses on this view, which may be visualized as *sweeping* a line perpendicular to the vector V_1 from infinity towards the origin, and the order in which the data tuples are encountered by this sweepline is the same as their order in the materialized view.

Let us discuss when the stopping condition is reached. Notice that when the sweepline encounters M, it cannot stop because at this stage there is no guarantee that the stopping condition has been reached. In fact, the stopping condition is reached when the sweepline crosses position AB_1. This is because, at this position the convex polygon AB_1POT encloses all the tuples that have not yet been encountered in the view, and by solving a linear program, we see that the maximum score of any tuple in this polygon (according to $Q's$ scoring function) is the score of point A, which is no more than the score of M. I.e., $\forall t \in AB_1POT, Score_Q(t) \leq Score_Q(M)$. The number of sorted accesses performed by this algorithm is equal to the number of tuples inside the triangle AB_1R (denoted as $NumTuples(AB_1R)$).

Now, let us compare this to the execution of *LPTA* when only the view V_2 is available for use. In this case, the algorithm stops when the sweepline crosses position AB_2. The number of sorted accesses performed by this algorithm equals $NumTuples(AB_2R)$. Clearly, this is a slower execution compared to using V_1. More generally, if several views in two dimensions are available, and all their vectors are to one side of the query vector, then it is optimal for *LPTA* to use the vector that is closest to the query vector.

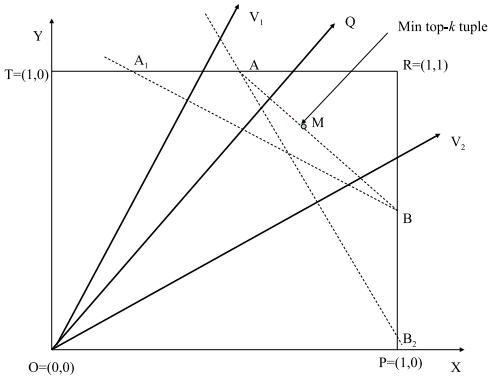

Figure 3: Selecting a view from either side of query

Next, let us consider the case when the view vectors are on either side of the query vector. This is illustrated in Figure 3. Suppose we can only use one of V_1 or V_2 for our execution. The respective stopping conditions are illustrated in the figure by the lines A_1B and AB_2 respectively, representing positions of the respective sweeplines when the stopping condition is reached. It is easy to see that V_1 (resp. V_2) should be preferred over V_2 (resp. V_1) if $NumTuples(A_1BR)$ is smaller (resp. larger) than $NumTuples$ (AB_2R).

Next, let us analyze whether two views are better than one. Is it more efficient to run *LPTA* on both V_1 and V_2, rather than just running on only one of V_1 or V_2? To understand this question, consider Figure 4. Since the algorithm is performing sorted accesses on both views in lock-step, the stopping condition is reached when the sweeplines respectively cross positions $A'_1B'_1$ and $A'_2B'_2$, such that

1. The intersection of $A'_1B'_1$ and $A'_2B'_2$ is a point S on the line AB
2. $NumTuples(A'_1B'_1R) = NumTuples(A'_2B'_2R)$ (since the algorithm sweeps each view in lock-step)

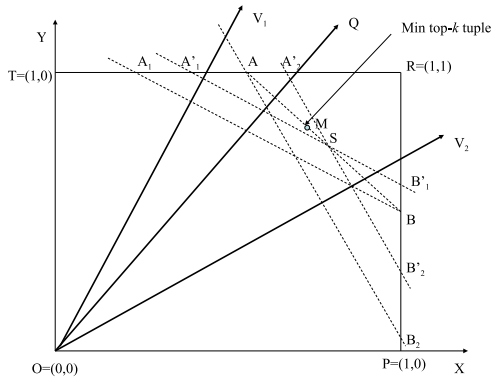

Figure 4: Selecting views from both sides of query

Note that at the time the algorithm stops, the position of each sweepline is *before* the respective stopping positions if only one view had been used. However, the total number of sorted accesses performed by the algorithm is equal to $NumTuples(A'_1B'_1R) + NumTuples(A'_2B'_2R) = 2NumTuples(A'_1B'_1R)$. In general, this may or may not be better than if we used just one view. Intuitively, if one of the view vectors, say V_1, is very close in orientation to the query vector, it may be best to just use V_1 only. Thus we conclude that if the minimum of $NumTuples(A_1BR)$, $NumTuples(AB_2R)$, and $2NumTuples(A'_1B'_1R)$ is $NumTuples$ (A_1BR) then use V_1; else if it is equal to $NumTuples(AB_2R)$ then use V_2; else use both V_1, V_2.

The following theorem summarizes our discussion of the two dimensional case:

THEOREM 1. *Let $\mathcal{V} = \{V_1, \ldots, V_r\}$ be a set of views for a two dimensional dataset, and let Q be a query. Let V_a and V_c be the closest view vectors to the query vector in anticlockwise and clockwise order respectively. Then the optimal execution of* LPTA *requires the use of a subset of the views from $\{V_a, V_c\}$.*

5.2 View Selection in Higher Dimensions

Theorem 1 can be extended to higher dimensions as follows:

THEOREM 2. *Let $\mathcal{V} = \{V_1, \ldots, V_r\}$ be a set of views for a m-dimensional dataset, and let Q be a query. Then the optimal execution of* LPTA *requires the use of a subset of the views $\mathcal{U} \subseteq \mathcal{V}$ such that $|\mathcal{U}| \leq m$.*

Proof: We shall prove this theorem by contraction. Assume that the views are in general positions, i.e., that no two view vectors are parallel. Assume that the optimal execution of *LPTA* for query Q requires the use of a set of views $\mathcal{U}'$ such that $|\mathcal{U}'| = m' > m$. Let W be a hyperplane perpendicular to the query vector such that it passes through the tuple M with the smallest top-k score. Let us visualize the execution of *LPTA* as the sweep of m' hyperplanes - each perpendicular to a corresponding view vector - from infinity in lock-step towards the origin. At any point, the intersection of the lower halves of these hyperplanes as well as the unit hypercube in

the first quadrant, defines a convex region. The stopping condition is reached when this convex region drops just below the hyperplane W. Let the number of lock-step iterations be d and let S be the maximal point of this convex region that lies on W at that instant.

Now, since the point S is a m-dimensional point, it is defined by the intersection of exactly m hyperplanes. Some of these hyperplanes may correspond to hyperplanes perpendicular to some of the view vectors, while others may correspond to hyperplanes that define the surface faces of the unit cube. Clearly, there are at least $m' - m$ hyperplanes corresponding to views that do not intersect with S. It is now easy to see that if we rerun *LPTA* with these latter views eliminated from $\mathcal{U}'$, the algorithm will stop after exactly the same number of iterations (d), and on stopping, the maximal point of the convex region that lies on W will be the same point S as the earlier run. This execution is cheaper, requiring at most md sorted accesses as compared to $m'd$ sorted accesses in the earlier run. □

However, Theorems 1 and 2 notwithstanding, we note that the decision of which m (or fewer) views to use has to be taken *before* the query is actually executed. Even in the two dimensional case, this involves deciding which of the two views V_a, V_c to use (or whether to use both). To do so, we need to estimate the score of the tuple M and the number of tuples contained in certain triangles (since this is equivalent to the number of sorted accesses performed). A naive way of estimating and comparing the number of tuples likely to be encountered is to simply compare the areas of the respective triangles. But such an approach assumes that the data follows an uniform distribution within the triangles, which is often quite unrealistic.

Instead, in our approach for view selection, we not only utilize the above conceptual conclusions, but *also* leverage knowledge of the actual data distribution. We discuss our proposed view selection algorithms in the next section.

6. VIEW SELECTION ALGORITHMS

In this section we develop algorithms for Problem 3.2. In particular, we develop a view selection procedure $SelectViews(Q, \mathcal{V})$, which, given a set of views $\mathcal{V}$ and a ranking query Q, determines a subset of views $\mathcal{U}$ on which the *LPTA* algorithm should be run. Towards the end of the section we also describe two simpler view selection procedures: $SelectViewsUniform(Q, \mathcal{V})$ and $SelectViewsByAngles(Q, \mathcal{V})$. These procedures are considerably simpler than $SelectViews(Q, \mathcal{V})$, but only work well for fairly restrictive datasets.

Since the objective is to return $\mathcal{U}$ such that *LPTA* can be executed on this subset, $\mathcal{U}$ has to encompass two properties, namely (a) it should be able to produce an answer to the top-k query and (b) the subset $\mathcal{U}$ identified should be the one that is able to provide an answer to a ranking query most efficiently among all possible subset choices of $\mathcal{V}$.

Notice that the set $\mathcal{V}$ available to Problem 3.2 contains the set of base views. As a result there is always a subset of views that can be utilized to provide an answer to any query. To determine the best set of views to supply *LPTA* with, we formulate an estimation problem. We describe a methodology to estimate the cost of actually running *LPTA* on a set of views. Since *LPTA* defaults to TA when the set of views in $\mathcal{U}$ is the base views, this methodology will estimate the cost of running TA on the base views as well. We will use this methodology to identify the set $\mathcal{U}$ with the smallest cost.

The notion of cost adopted in this estimation is the total number of sequential and random data accesses conducted. Both *LPTA* and TA have sequential and random data access components that are highly correlated. Every sequential access in a view (in the case of *LPTA*) or base view (in the case of TA) prompts a random access to the base relation in order to retrieve the complete list of values in the corresponding tuple and compute the score. As a result, the number of sequential accesses is a precise indicator of their sequential and random access I/O behavior. We develop a cost model that estimates the number of elements that *LPTA* (TA) will sequentially accesses provided with a set of views.

The rest of this section is organized as follows. First, we discuss a cost estimation procedure $EstimateCost(Q,\mathcal{U})$ which, given a query Q and any subset of views $\mathcal{U} \subseteq \mathcal{V}$, returns an estimate of the cost of running *LPTA* on exactly this set of views. Second, we show how this procedure is used by the view selection procedure $SelectViews(Q, \mathcal{V})$ to search amongst the subsets of $\mathcal{V}$ for the subset $\mathcal{U}$ that minimizes $EstimateCost(Q,\mathcal{U})$. Finally, we conclude the section with two simpler view selection procedures: $SelectViewsUniform(Q, \mathcal{V})$ and $SelectViewsByAngles(Q, \mathcal{V})$.

6.1 The Procedure $EstimateCost(Q,\mathcal{U})$

In this subsection we present a comprehensive estimation procedure that takes into account multi-attribute views (higher dimensions) as well as non-uniform data distributions.

6.1.1 Approximating Score Distributions

$EstimateCost(Q,\mathcal{U})$ is an estimation procedure. Thus, to be able to estimate the execution cost, we have to assume that some type of compact distributional model of the data is available. Since real data is rarely uniformly distributed along each attribute, we make use of standard database statistics such as *histograms* computed to represent the distribution of the data along each attribute of the base table R of size n (tuples). For this paper, we assume *equi-depth* histograms, thus if H_i is an equi-depth histogram with b buckets representing the distribution of points along the X_i attribute, then each bucket represents n/b data points.[1] Similarly we make use of histograms of the score distribution for each view. Such histograms can be computed when the view is materialized or derived on demand.

Let the scoring function of Q be $Score_Q(t) = w_1t[1]+w_2t[2]+ \ldots w_mt[m]$. Clearly each tuple's score lies in the interval $[0, \sum_i^m w_i]$. In our estimation procedure, we will need to compute a histogram H_Q that represents the distribution of the scores of all tuples of the database according to this scoring function. Since computing the scores of all tuples in the database is prohibitive, we discuss a much more efficient way of approximating H_Q by *convolutions* of the histograms H_i that represent the marginal distributions along each attribute. The convolution procedure, which we discuss next, assumes that the attributes are *independent*.

We define the convolution of two probability density functions (pdfs) as follows.

DEFINITION 1. **Convolution of two distributions:** *Let $f(X)$, $g(Y)$ be the pdfs of two independent random variables X, Y respectively. Then $e(Z)$, the pdf of the random variable $Z = X + Y$, is known as the convolution of $f(X)$ and $g(Y)$, and can be expressed as $e(Z) = \int_0^Z f(Y)g(Z - Y)dY$*

In our case, pdfs are represented by histograms. Consider a query Q with a simple scoring function $Z = X_i + X_j$ where X_i, X_j are two attributes. The histogram H_Q representing the distribution of Z can be computed by the convolution of the two b-bucket histograms H_i, H_j by replacing the integral above with a cartesian product of histograms as follows. Assume that the bucket

[1]While other types of histograms are of course possible, the specific type of histogram to be used is orthogonal to the methods of this paper, and we use equi-depth histograms mainly for the ease of exposition.

Algorithm 2 $EstimateCost(Q,\mathcal{U})$

$\mathcal{U} = \{V_1, \ldots, V_{r'}\}$ // Set of views
$Q = (Score_Q, k, *)$ // Query
Compute $H_Q, H_{V_1}, H_{V_2}, \ldots, H_{V_{r'}}$ // via convolutions
Estimate $topk_{min}$ from H_Q
for d = 1 to b // b buckets in each histogram **do**
  for all $H_{V_i} (1 \leq i \leq r')$ in lock-step **do**
    Let s_d^i be lower boundary of current bucket in H_{V_i}
    //Check stopping condition by solving LP
    Compute $Unseen_{max}$ = max of $Score_Q$ subject to
     $lb_j \leq X_j \leq ub_j \; \forall \, 1 \leq j \leq m$
     $Score_{V_j} \leq s_d^j \; \forall \, 1 \leq j \leq r'$
    if $(unseen_{max} \leq topk_{min})$ **then**
      // Perform logarithmic search within last buckets
      Let $s_d^i(n')$ denote interpolated score of n'th tuple in current bucket of H_{V_i} $(1 \leq n' \leq n/b)$
      Let $Unseen_{max}$ = max of $Score_Q$ subject to
       $lb_j \leq X_j \leq ub_j \; \forall \, 1 \leq j \leq m$
       $Score_{V_j} \leq s_d^j(n') \; \forall \, 1 \leq j \leq r'$
      Compute (via logarithmic search) smallest n' s.t. $unseen_{max} \leq topk_{min}$
      Return $((d-1)n/b + n')r'$
    end if
  end for
end for

Algorithm 3 $SelectViews(Q, \mathcal{V})$

$\mathcal{V} = \{V_1, \ldots, V_r\}$ // Set of views
$Q = (Score_Q, k, *)$ // Query
$\mathcal{U} = \{\}$
$MinCost = MinCurCost = \infty$
for $i = 1$ to m **do**
  for $V \in \mathcal{V} - \mathcal{U}$ **do**
    if $(EstimateCost(Q,\mathcal{U} \cup \{V\}) < MinCurCost)$ **then**
      $MinV = V$
      $MinCurCost = EstimateCost(Q,\mathcal{U} \cup \{V\})$
    end if
  end for
  if $(MinCurCost < MinCost)$ **then**
    $\mathcal{U} = \mathcal{U} \cup \{MinV\}$
    $MinCost = MinCurCost$
  else
    Return $\mathcal{U}$
  end if
  Return $\mathcal{U}$
end for

boundaries of H_i and H_j are the same: $[0 = h_0, h_1, \ldots, h_b = 1]$ (if not, we can create two equivalent histograms with $2b$ buckets and the same bucket boundaries). Consider the cartesian product $C_{i,j} = H_i \times H_j$ where $C_{i,j}[p,q] = H_i[p] \cdot H_j[q]$ (where $H_i[p]$ is the relative count associated with bucket p).

We can approximate the pdf of $X_i + X_j$ with a histogram with $2b$ buckets and boundaries $g_0 = 0, g_1 = h_1, \ldots, g_b = 1, g_{b+1} = 1+h_1, \ldots, g_{2b} = 2$. To compute the histogram we have to compute the probability $Prob(g_k < A + B \leq g_{k+1})$ for the buckets of the new histogram, which may be derived as $\sum_{h_l + h_m = g_{k+1}} C_{i,j}[l][m]$. This histogram can subsequently be approximated by a b bucket histogram by merging neighboring pairs of buckets. This procedure gives an $O(b^2)$ algorithm for computing the convolution of the two pdfs.

Note that this basic technique can also be utilized to derive score value distribution histograms on demand for each view using attribute value histograms of base relation attributes. This approach can be extended to computing (in $O(mb^2)$ time) histograms of more general linear combination scoring functions of the general form $Z = \sum_i^m w_i X_i$. We omit further details from this version of the paper.

6.1.2 *Simulating* LPTA *on Histograms*

Algorithm 2 describes the pseudo code for $EstimateCost(Q,\mathcal{U})$. Since we obviously cannot afford to execute *LPTA* directly on the views in $\mathcal{U}$, we perform the cost estimation by first computing histograms representing the distribution of scores along each view in $\mathcal{U}$ according to the scoring functions of the respective views (in case these histograms have not already been computed when the views were created), and then "walking down" these histograms bucket by bucket in lock-step until the stopping condition is reached. Even though we are simulating *LPTA*, the estimation time is much faster because a single iteration of this simulation corresponds to n/b iterations if directly executed on the views.

There are two complications with this simulation that need to be discussed. First, we need to pre-estimate $topk_{min}$. This is necessary because we do not have access to actual tuples or their tids, as we are accessing histograms which only contain aggregated information. The value of $topk_{min}$ is estimated from H_Q by determining the bucket which contains the kth highest tuple. Since the kth tuple is very likely to be inside a bucket, we use linear interpolation within the bucket to estimate $topk_{min}$.

Second, if the final number of buckets visited along each view's histogram is d, a very simple estimation of the number of sorted accesses is nr'/b, where r' is the number of views used in our estimation. However, since n/b can be large, this estimate can be rather crude. We refine this estimate by computing the smallest number of tuples, n', that need to be scanned from each of the last buckets visited, so that the stopping condition is reached - using linear interpolation for the scores of tuples within each bucket. Since the size of each bucket is potentially large (n/b), we cannot afford to do a linear search to determine n'. Instead, we perform a "logarithmic" search, which proceeds by repeatedly doubling n', checking the stopping condition at each iteration, and then performing a final binary search between the last pairs of values of n'.

Thus, the algorithm finally returns the estimated number of sorted accesses as $((d-1)n/b+n')r'$. The running time of this algorithm is $O((d-1) + \log n')$ lock-step iterations.

6.2 The Procedure $SelectViews(Q, \mathcal{V})$

Once we have developed a cost estimation procedure, our next task is to determine the subset of views of a base relation R with m attributes with least cost. According to Theorem 2, an obvious algorithm to determine the best subset of views $\mathcal{U}$ is to estimate the cost of all possible $\binom{r}{p}$ subsets of $\mathcal{V}$ (where $p \leq m$) and select the subset of views with the smallest cost (we refer to this approach as $Excustive$). While this approach is feasible for databases with few attributes, for large values of m this can often be prohibitive. For such cases we follow a simple greedy heuristic to select the subset set of views with the cheapest cost. The pseudo code is described in Algorithm 3.

6.3 Simpler View Selection Procedures

In this subsection we present two procedures for selecting views that are considerably simpler than $SelectViews(Q, \mathcal{V})$ presented above. However, these procedures are only effective for datasets with fairly restrictive data distributions.

6.3.1 The Procedure $SelectViewsSpherical(Q, \mathcal{V})$

Consider the special case of a dataset that is uniformly distributed in the unit m-dimensional sphere (i.e., there is no preferred direction in the dataset, nor is there any skew in the data density). Let $\mathcal{V}$ be a set of views over this dataset, and let Q be a query. Assume that the score function of each view has been *normalized*, such that if we take a point t that intersects a view vector V and the surface of the unit sphere, then $Score_V(t) = 1$. Now let us imagine that we execute *LPTA* on the entire set of $\mathcal{V}$ views. The execution may be visualized as sweeping the corresponding perpendicular hyperplanes from just outside the unit sphere towards the origin. After d iterations, let $t_d^1, \ldots, t_d^r$ be the last tuples read from each view, and let $Unseen_d$ be the convex region defined by the intersection of the lower halfspaces below each view's hyperplane. From the symmetry of the spherical distribution, we can make the following assumptions:

1. The distance of each t_d^i to the origin is the same (equivalently, $Score_{V_1}(t_d^1) = Score_{V_2}(t_d^2) = \ldots Score_{V_r}(t_d^r)$). Moreover, as d increases, these distances (scores) decrease by the same amount.

2. As d increases, the convex region $Unseen_d$ does not change in *shape*; it just gets smaller in scale.

These assumptions suggest a very simple algorithm to select a set of views $\mathcal{U}$. We fix any valid score value, say s. We then solve the linear program where we maximize $Score_Q$ subject to the inequalities $Score_{V_j} \leq s \; \forall \; 1 \leq j \leq r'$. Let S be the vertex of the convex region at which $Score_Q(S)$ is maximized. Using arguments similar to Theorem 2, the set $\mathcal{U}$ is defined as those views that intersect with S.

Clearly this procedure is much simpler than the more general $SelectViews()$ procedure described earlier, as it only has to solve a linear program once (and does not have to simulate *LPTA* on histograms). However, it is also clear that this procedure will only work well for very restrictive data distributions.

6.3.2 The Procedure $SelectViewsByAngles(Q, \mathcal{V})$

We finally present an extremely simple heuristic for selecting views. The procedure $SelectViewsByAngles(Q, \mathcal{V})$ simply sorts the view vectors by increasing angle with the query vector, and returns the top-m views as the set $\mathcal{U}$, the intuition being that views that have similar orientations with the query vector are likely to contribute towards early stoppage of *LPTA*. However, this procedure suffers from two disadvantages: (a) it too makes an implicit assumption of uniform data distribution, and (b) several of the selected views may render one another "redundant" as they may have very similar orientations.

7. MORE GENERAL QUERIES AND VIEWS

In this section we extend our methods to work for more general queries and views. We first consider the case where views may only be restricted to materializing the top-k tuples. We then extend to the case where both views and queries my be associated with range selection conditions.

7.1 Views that Only Materialize their Top-k Tuples

In this subsection we consider the case where materialized views only contain their top-k tuples. The view selection problem may be posed as follows: Given a set of views $\mathcal{V} = \{V_1, \ldots, V_r\}$ that contains the set of base views where each non-base view $V_i = (Score_{V_i}, k_i, *)$, and a query $Q = (Score_Q, k, *)$, determine the subset of views $\mathcal{U}$ with least cost for answering the query. The new complication stems from the fact that certain combinations of views now *may not* be able to answer the query on hand - simply because they may not have enough tuples materialized.

Our approach to solving this problem is to modify the procedure $SelectViews()$ and its associated subroutine $EstimateCost()$ of the previous section as follows. First, for each view V_i, we prepare its histogram H_i using convolutions as before, but then *truncate* the histogram by removing a portion of its tail such that the truncated histogram represents the distribution of the top-k_i tuples of the view. Next, when the *LPTA* algorithm is mimicked in the $EstimateCost()$ procedure, if any of the view histograms are exhausted before the $topk_{min}$ score has been reached, the procedure simply returns a cost of infinity. This way, the greedy $SelectViews()$ algorithm will make sure not to return this particular combination of views. Note that since base views are present in $\mathcal{V}$, and each of these views cover all tuples in the database, the procedure $SelectViews()$ is guaranteed to always return with a feasible combination of views for answering Q.

7.2 Accommodating Range Conditions

In this subsection we consider the case where views and queries are also associated with range selection conditions $Range_V$ and $Range_Q$ respectively. Each selection condition is a conjunctive condition of the form $[l_1, u_1] \& [l_2, u_2] \& \ldots \& [l_m, u_m]$ with the usual semantics - a tuple is selected if all its attribute values fall within the respective intervals of the range condition. The view selection problem may be posed as follows: Given a set of views $\mathcal{V} = \{V_1, \ldots, V_r\}$ where each view $V_i = (Score_{V_i}, k_i, Range_{V_i})$, and a query $Q = (Score_Q, k, Range_Q)$, determine the subset of views $\mathcal{U}$ with least cost for answering the query. As with views that materialize only their top-k tuples, views with range conditions complicate the view selection problem because now certain combinations of views may not have enough tuples to be able to answer the query at hand.

We propose the following simple but effective approach to solving this problem. We first select a subset $\mathcal{W} \subseteq \mathcal{V}$ of views, such that *each view* in $\mathcal{W}$ "covers" the query's range. I.e, $V \in \mathcal{W}$ if and only if $Range_Q \subseteq Range_V$. The subset W can be selected very efficiently as each each interval of the view's range condition can be easily checked to see if it contains the corresponding interval of the query's range condition (several indexing schemes may be utilized for this step [12]).

Once $\mathcal{W}$ has been selected, we run $SelectViews()$ on $\mathcal{W}$ instead of $\mathcal{U}$. The algorithm for $SelectViews()$ needs to be modified, especially the part that computes score histograms for the query as well as the views in $\mathcal{W}$ (unless of course the score histogram of the view has already been constructed at view creation time). To compute the score histograms of views, we first truncate each attribute's histogram H_i such that the truncated histogram represents the distribution of X_i values only within the corresponding interval defined in the view's range condition. These truncated histograms are then convoluted to generate the histogram corresponding to the view's score distribution.[2] Using a similar procedure, we also compute a histogram that describes the query's score distribution. The procedure $SelectViews()$ can be run without any further modifications.

We note that in principle, it is not necessary for each view's range

[2]Note that in the earlier case of views that materialize only their top-k tuples, we first convoluted the attribute histograms and then truncated the resultant score histogram. In contrast, here we first do the truncations and then perform the convolutions.