Efficient Top-k Query Answering using Cached Views

Min Xie
Dept. of Computer Science,
Univ. of British Columbia
minxie@cs.ubc.ca

Laks V.S. Lakshmanan
Dept. of Computer Science,
Univ. of British Columbia
laks@cs.ubc.ca

Peter T. Wood
Dept. of CS and Inf. Syst.,
Birkbeck, U. of London
ptw@dcs.bbk.ac.uk

ABSTRACT

Top-k query processing has recently received a significant amount of attention due to its wide application in information retrieval, multimedia search and recommendation generation. In this work, we consider the problem of how to efficiently answer a top-k query by using previously cached query results. While there has been some previous work on this problem, existing algorithms suffer from either limited scope or lack of scalability. In this paper, we propose two novel algorithms for handling this problem. The first algorithm LPTA$^+$ provides significantly improved efficiency compared to the state-of-the-art LPTA algorithm [26] by reducing the number of expensive linear programming problems that need to be solved. The second algorithm we propose leverages a standard space partition-based index structure in order to avoid many of the drawbacks of LPTA-based algorithms, thereby further improving the efficiency of query processing. Through extensive experiments on various datasets, we demonstrate that our algorithms significantly outperform the state of the art.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: System—*query processing*

General Terms

Algorithms, Performance

Keywords

Top-k Query Processing, Top-k Query Answering using Views

1. INTRODUCTION

Top-k query processing has recently received a significant amount of attention due to its wide application in information retrieval, multimedia search and recommendation generation [32]. In many of these applications, one assigns to each result tuple t a *value* or *score* using a score function, which indicates the desirability or preference of t; and instead of returning to the user all result tuples, the number of which might be overwhelming, one returns only the k most preferred tuples under the given score function, where k is a small positive integer.

EDBT/ICDT '13, March 18–22, 2013, Genoa, Italy.

R

tid	A	B	C
t_1	0.3	0.6	0.4
t_2	0.4	0.5	0.6
t_3	0.3	0.7	0.3
t_4	0.5	0.3	0.5
t_5	0.2	0.8	0.8
t_6	0.6	0.5	0.7

(a)

V_1 (f_1=0.1A+0.9B, k_1=3)

tid	rank	score
t_5	1	0.74
t_3	2	0.66
t_1	3	0.57

V_2 (f_2=0.1A+0.5B+0.4C, k_2=3)

tid	rank	score
t_5	1	0.74
t_6	2	0.59
t_2	3	0.53

(b)

Figure 1: (a) A relation R with three attributes A, B, C; (b), two cached views V_1, V_2 which contain top-3 tuples according to the the two score functions $f_1(t) = 0.1t[A] + 0.9t[B]$, $f_2(t) = 0.1[A] + 0.5[B] + 0.4[C]$ respectively.

For the applications under consideration, typically a simple linear score function is used to aggregate the attribute values of a tuple into a score, due to its intuitiveness [22, 30, 26, 43, 25, 45]. Figure 1 (a) shows an example relation R which contains 6 tuples over attributes A, B and C. Consider a query Q_1 which asks for the top-3 tuples with the highest values under the score function $f_1(t) = 0.1t[A] + 0.9t[B]$. The result is shown as (a cached view) V_1 in Figure 1 (b).

While various efficient algorithms have been proposed for processing top-k queries [27, 32], one significant limitation is that they cannot take advantage of the cached results of previous queries. E.g., consider the previous example query Q_1 whose result V_1 is shown in Figure 1 (b). Suppose a (possibly different) user subsequently asks the top-1 query Q' with the score function $f'(t) = 0.2t[A] + 0.8t[B]$. Then, as we will see in later sections, we can use the previous cached result V_1 to determine, *without accessing the original database R*, that t_5 is the top-1 answer for Q'.

Leveraging cached query results to scale up query answering has recently become increasingly popular for most large scale websites. For example, the popular Memcached [4] caching system has reportedly been adopted by many large scale websites such as Wikipedia [12], Flickr [2], Twitter [9] and Youtube [14]. The application of cached query results or materialized views for speeding up query answering in relational databases, the so-called query answering using views

(QAV) problem, has been extensively studied (see [28] for an excellent survey). This problem has been shown to have applications in data integration [36], query optimization [38], and data warehouse design [42].

For top-k query processing, recently there have been some initial efforts at using materialized query results for speeding up query answering. In the PREFER system, Hristidis et al. [30] consider the problem of how to select one best materialized view for answering a query. Their setting is quite restrictive, as it cannot exploit multiple materialized views, and it also makes a strong assumption that all attributes of the underlying base table are always utilized for all top-k queries. Overcoming these limitations, Das et al. [26] propose a novel algorithm, called LPTA, which is able to utilize multiple materialized views for answering a top-k query. Ryeng et al. [41] extend these techniques to answer top-k queries in a distributed setting.

Though LPTA overcomes many of the limitations of PREFER, unfortunately it still suffers from several significant limitations. Firstly, the core techniques proposed in [26] rely on the assumption that either (1) each top-k view is a *complete ranking of all tuples in the database*, or (2) that the *base views*, which are *complete rankings of all tuples in the database according to the values of each attribute, are available*. These assumptions may often be unrealistic in practice.

Consider the example of finding top-k movies. There are several popular websites which provide top-k lists of movies based on different criteria. For example, Metacritics [5] provides a ranked list of (up to 5639) movies based on *Metascore* [6], which is aggregated from critics and publications like the New York Times (NYT) and the San Francisco Chronicle; IMDB [3] provides a top-250 list of movies based on votes from their users; and RottenTomatoes (RT) [8] provides a top-100 list of movies based on the *Tomatometer* score, which is calculated based on critics. Here, the top-k lists on Metacritics, IMDB, and RT can be regarded as materialized views. Because of the huge number of movies available, it is impractical to obtain the complete ranking of all movies from each of the sources, and for the same reason, we cannot assume base views corresponding to the complete ranking of all movies on each of the individual scores, e.g, NYT score, are available. Consider the query that asks for top-k movies according to an aggregation of NYT score, IMDB score, and Tomatometer score. Since the only information we have access to is the top-k movies from the Metacritics, IMDB, and RT, *the technique proposed in [26] cannot be used to answer this query*. Similar examples can also be found in other domains: finding the top-k universities based on university ranking lists from U.S. News [11] and The Times [13]; or finding the top-k cars based on automobile ranking lists from U.S. News [10] and Auto123 [1].

The second issue with the LPTA algorithm proposed in [26] is that it uses linear programming (LP) as a sub-procedure to calculate the upper bound on the maximum value achievable by a candidate result tuple, and the LPTA algorithm needs to call this sub-procedure iteratively. It has been demonstrated in [26] that for low dimensionality scenarios (e.g., 2 or 3), the cost of this LP overhead is reasonable. However, we will show in our experiments that for scenarios with higher dimensionality, which we note is very common, this iterative invocation of the LP sub-procedure may incur a high computational overhead.

Finally, for both PREFER [30] and LPTA [26], a potentially costly view selection operation is necessary. For example, the view selection algorithm in [26] requires the simulation of the top-k query process over the histograms of each attribute, and the processing cost is linear with respect to the number of views. This cost can be prohibitive given a large pool of cached query (view) results. Furthermore, (histograms over) base views are often not available in practice, restricting the applicability of this view selection procedure.

In this paper, we propose two novel algorithms for the problem of top-k query answering using cached views. Our first algorithm LPTA$^+$ is an extension of LPTA as proposed in [26]. In LPTA$^+$, we make a novel observation on the characteristics of LPTA, and by taking advantage of the fact that our views are cached in memory, we can usually avoid the iterative calling of the LP sub-procedure, thus greatly improving the efficiency over the LPTA algorithm. LPTA$^+$ can be useful for scenarios with a small number of views and low dimensionality. For the case where the number of cached views is large and the dimensionality is high, we further propose an index structure called the *inverted view index* (*IV-Index*), which stores the contents of all cached views in a central data structure in memory, and can be leveraged to answer a new top-k query efficiently *without any need for view selection*.

Specifically, we make the following contributions: (1) We consider the general problem of top-k query answering using views, where base views are not available, and the cached views include only the top-k tuples which need not cover the whole view (Section 2). (2) For scenarios where we are not allowed to maintain additional data structures, we extend LPTA and propose a new algorithm, LPTA$^+$, which can significantly improve performance over LPTA (Section 3). (3) We further propose a novel index-based algorithm, IV-Search, which leverages standard space-partitioning index structures, and can be much faster than LPTA/LPTA$^+$ in most situations. We consider two different strategies for the IV-Search algorithm, and discuss additional optimization techniques (Section 4). (4) We present a detailed set of experiments showing that the performance of our proposed algorithms can be orders of magnitude better than the state-of-the-art algorithms (Section 5). Related work is discussed in Section 6 and we conclude the paper in Section 7.

2. PROBLEM SETTING

Given a schema $\mathbf{R}$ with m numeric attributes $A_1, \ldots, A_m$, we denote a relation instance of $\mathbf{R}$ by R. In practice, $\mathbf{R}$ may have other non-numeric attributes as well, but we are concerned only with the numeric attributes. Every tuple $t \in R$ is an m-dimensional vector $t = (t[1], \ldots, t[m])$, where $t[i]$ denotes the value of t on attribute A_i, $i = 1, \ldots, m$. Similar to previous work on top-k query processing, we assume that attribute values are normalized in the range of $[0, 1]$, and that each tuple t also has a unique tuple id.

Similar to [26], we define a top-k query Q over $\mathbf{R}$ as a pair (f, k), where k is the number of tuples required, and $f : [0, 1]^m \rightarrow [0, 1]$ is a linear function which maps the m attribute values of a tuple t to a preference score, i.e., $f(t) = w_1 t[1] + \cdots + w_m t[m]$, where $w_i \in [0, 1]$, and $\sum_i w_i = 1$. Note that since every w_i is non-negative, the function f is clearly *monotone*, i.e., for two tuples t_1 and t_2, if $t_1[i] \geq t_2[i]$, $i = 1, \ldots, m$, then $f(t_1) \geq f(t_2)$.

Given a relation R and a query $Q = (f, k)$, without loss of

generality, assume that $k \leq |R|$ and that larger f values are preferred. Then the semantics of top-k query processing is to find k tuples in R which have the largest values according to the query score function f. We can formally define the answer to a top-k query as follows.

DEFINITION 1. **Top-k Query Answer**: *Let $Q = (f, k)$ be a top-k query over relational schema $\mathbf{R}$. Given a relation R over $\mathbf{R}$, the answer of Q on R, $Q(R)$, is a list of tuples from R such that $|Q(R)| = k$, and $\forall t \in Q(R)$ and $\forall t' \in R \backslash Q(R)$, $f(t) \geq f(t')$. Finally, tuples of higher rank in $Q(R)$ have a higher score according to the score function f.*

A *top-k cached view*, or a top-k view for brevity, is defined similarly to a top-k query, except the results of a top-k view are cached in memory. For each tuple t in a cached view, we assume all attribute values $t[i]$, $i = 1, \ldots, m$, will also be cached in memory, and thus can be efficiently accessed at query time. We allow random access by id on the cached tuples. Given a view $V_i = (f_i, k_i)$, without any ambiguity, we reuse V_i also to denote the list of (k_i) tuples materialized along with their ranks and scores w.r.t. f_i.

We use $V_i[j]$ to denote the tuple $t \in V_i$, which has the jth highest score w.r.t. f_i, with ties broken using tuple id, i.e., a tuple with a smaller tuple id will be ranked higher. Similarly for a given relation R, we denote the jth tuple in R following the order defined by a score function f as $R_f[j]$.

Let $\mathcal{V} = \{V_1, \ldots, V_p\}$ be a set of views, where $V_i = (f_i, k_i)$ is a top-k_i view, and let $Q = (f, k)$ be a top-k query. Inspired by the notion of *certain answers* when answering a non-ranking query using views [15], we say a relation R is *score consistent* with the set $\mathcal{V}$ of views, if for any view $V_i = (f_i, k_i) \in \mathcal{V}$, the jth tuple $R_{f_i}[j]$ in R w.r.t. f_i has the same score as the jth tuple $V_i[j]$ in V_i, i.e., $f_i(R_{f_i}[j]) = f_i(V_i[j])$, for $j = 1, \ldots, k_i$. Note that we do *not* require $R_{f_i}[j]$ to have the same tuple id as $V_i[j]$, since the score of a tuple is determined solely by its attribute values and not by its tuple id (Definition 1).

Given a set of views $\mathcal{V} = \{V_1, \ldots, V_p\}$, a score consistent relation R is the counterpart of a possible world under the closed world assumption (see [15]). Accordingly, we define a tuple $t \in V_i$, $i = 1, \ldots, p$, to be a *certain answer* to Q if, for any relation R which is score consistent with $\mathcal{V}$, $f(t) \geq f(R_f[k])$, i.e., the score of tuple t is no worse than the score of the kth tuple in R under the query score function f. Motivated by the previously mentioned applications where we need to efficiently answer a query using merely top-k views, we consider the following problem.

DEFINITION 2. **Top-k QAV (kQAV)**: *Given a set $\mathcal{V} = \{V_1, \ldots, V_p\}$ of top-k_i views, $i = 1, \ldots, p$ and a top-k query $Q = (f, k)$, find all certain answers of Q, denoted $Q(\mathcal{V})$, up to a maximum of k answers.*

Notice that we have no access to the complete ranking of tuples in the views nor access to the base views. Similar to query answering using views in a non-ranking setting [15], given only the view set $\mathcal{V}$, we need to find the certain answers. The number of certain answers may be less than, equal to, or more than k. Since $Q = (f, k)$, we restrict the output to a maximum of k certain answers, where any ties at rank k are broken arbitrarily.

As an example, consider the set of views $\mathcal{V} = \{V_1, V_2\}$ as shown in Figure 1 (b) and assume the base relation R is no longer available. Assume we are given the query $Q = (f, 1)$, where $f = 0.1A + 0.8B + 0.1C$. Using the techniques proposed in Section 3, we can determine that $\{t_5\}$ is the set of certain answers to Q. Intuitively, this is because after accessing the second tuple in V_1 and V_2, we can derive that for all unseen tuples, the maximum possible value w.r.t. Q is 0.6425 which is smaller than the current best tuple t_5 for which $f(t_5) = 0.74$. And if $Q = (f, 4)$, we can only find 3 certain answers to Q, which are t_5, t_3 and t_1. This is because after accessing all three tuples in V_1 and V_2, the maximum possible value w.r.t. Q is 0.56 for all unseen tuples, and only t_5, t_3, t_1 have projected values larger than or equal to 0.56.

2.1 System Overview

Motivated by the applications discussed in the introduction, we consider the following top-k query evaluation framework as illustrated in Figure 2. For a top-k query $Q = (f, k)$ submitted to the query processing system, the query executor will consult the cached top-k views to find the maximum set of certain answers to Q. In this work, we will focus on the kQAV problem where the goal is to efficiently find certain answers using only the given top-k views.

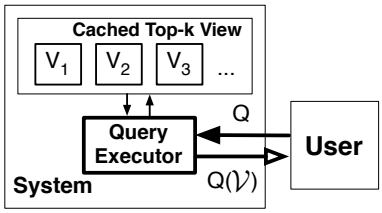


Figure 2: System overview.

In the following sections, we will first adapt and improve the LPTA algorithm as originally proposed in [26] for addressing the kQAV problem as defined above, where neither the complete ranking of tuples nor the base views are available. We will then discuss how a standard space partition-based index can be used to further optimize the performance of the algorithm.

3. LPTA-BASED KQAV PROCESSING

In this section, we first discuss LPTA, the state-of-the-art algorithm proposed in [26] for answering a top-k query using a set of views.[1] We shall see in Section 3.1 that LPTA has several limitations. We first review LPTA and discuss how it can be adapted to produce certain answers when cached views are not complete rankings of tuples and no base views are available. In Section 3.2, we propose a new algorithm LPTA$^+$ which overcomes the limitations of LPTA.

3.1 Algorithm LPTA

In [26], Das et al. first studied the problem of answering a top-k query using multiple views. Similar to the TA algorithm [27], the authors of [26] assume that the underlying database can be randomly accessed to retrieve tuple attribute information using tuple ids, and that each view stores a list of tuple ids along with the scores. They focus on the scenario where either each view is a complete ranking of all tuples in R, or the base views, which are complete rankings of all tuples in R according to the values of each

[1]Recall that they assume that views are complete rankings of tuples or that base views are available.

attribute, are available. Thus a top-k query can always be answered exactly and completely. We next briefly review the LPTA algorithm presented in [26].

Consider the score function f of each query/view also as a vector $\vec{f}$ from the origin O, representing the direction of increasing value. Given the assumption on the score function, the vector defined by any possible score function considered will reside in the first quadrant. For now, we will assume that for every cached view $V_i = (f_i, k_i)$, it is the case that $k_i = |R|$ and the tuples in V_i are sorted based on f_i, or their projected values on $\vec{f_i}$. In Figure 3 (a), we show an example of the relation R from Example 1 when projected on the first two dimensions A and B. Given a query $Q = (f, k)$, we can rank the tuples by projecting them onto $\vec{f}$, as shown in the figure.

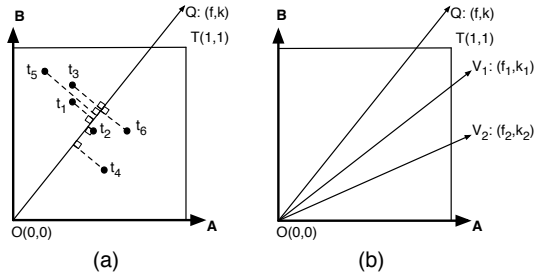


Figure 3: Example of LPTA.

Recall that we have a set $\mathcal{V}$ of p views, and assume that a set $\mathcal{U} \subseteq \mathcal{V}$ of r views has been selected in order to answer the query (we discuss the view selection problem below). In order to answer a top-k query $Q = (f, k)$, the LPTA algorithm accesses tuples sequentially from the r views. For each tuple t accessed, the algorithm performs a random access to the database in order to retrieve the attribute value information of t. The current candidate top-k results can be easily maintained from the accessed tuples. However, it is more challenging to find the maximum value τ that can be achieved by any unseen tuple, which is critical for the stopping criterion of the LPTA algorithm. Let the last tuple accessed in each view $V_i = (f_i, k_i) \in \mathcal{U}$ be denoted by $\bar{t}_i$, $i = 1, \ldots, r$. As observed in [26], τ can be calculated by solving the following linear programming (LP) problem:

$$\begin{aligned} \max_{t} \quad & \tau = f(t) \\ \text{subject to:} \quad & f_i(t) \leq f_i(\bar{t}_i), i = 1, \ldots, r \\ & 0 \leq t[i] \leq 1, i = 1, \ldots, m \end{aligned} \tag{1}$$

The "LP solver" is clearly more complex and time consuming than other components in the LPTA algorithm, so instead of invoking this solver every time a new tuple is accessed from a view V_i, LPTA accesses tuples from the r views in a lock-step fashion, i.e., the LP solver will be called once for every r tuples accessed.

The pseudocode for LPTA is given in Algorithm 1. We initialize a priority queue X based on the score function f of Q (line 1) and the threshold value τ (line 2). The algorithm then iteratively accesses tuples from the r views in a lock-step fashion (lines 4–5). For each set of r tuples accessed, the algorithm finds the value of τ by solving the LP problem (Formula 1) (line 8). If the kth tuple $X[k]$ in the priority queue has value no less than τ, the algorithm can stop.

As we will demonstrate in Section 3.2, while the cost of iteratively calling the LP solver is reasonable when the dimensionality for the given input relation R is low, the cost increases significantly as the dimensionality grows. *We will discuss in Section 3.2 how this increased cost can be avoided by leveraging innate characteristics of the kQAV problem.*

Algorithm 1: LPTA($\mathcal{U} = \{V_1, \ldots, V_r\}$, $Q = (f, k)$)

```
1 X ← an empty priority queue;
2 τ ← ∞;
3 repeat
4 |   {t̄_1, ..., t̄_r} ← getNextTuple(U);
5 |   retrieveTupleInfo({t̄_1, ..., t̄_r});
6 |   X.insert({t̄_1, ..., t̄_r});
7 |   X.keepTop(k);
8 |   Find τ by solving the LP problem in Formula (1);
9 until noNewTuple(U) or (|X| = k and f(X[k]) ≥ τ);
```

Another problem that remains to be addressed in using LPTA is how to choose the r views from a potentially large pool of cached views, so that query processing cost can be minimized. As shown in [26], we need no more than m views for processing a query on an m-dimensional relation R (so $r \leq m$), and this view selection process is critical for the performance of the LPTA algorithm. In [26], the authors first observe that for the 2-dimensional case, we can prune views by considering the angle between view score function vectors and the query score function vector. Given a query score function f and two view score functions f_1, f_2, if $\vec{f_1}$ and $\vec{f_2}$ are to the same side of $\vec{f}$, then we only need to select the view which has the smaller angle to $\vec{f}$ for answering the query, while the other view can be pruned. For example, consider the two cached views V_1 and V_2 along with query Q in Figure 3 (b). Because $\vec{f_1}$ has the smaller angle to $\vec{f}$, Q can be answered using V_1, while V_2 can be pruned.

However, this pruning technique may not be very useful for high dimensional scenarios. As has been shown in recent work [41], the pruning of views in the general case may involve solving an LP problem whose number of constraints is proportional to the total number of views. This is clearly not practical when the number of views is large, but this is precisely the situation that arises when we want to answer a query using previously cached results. Thus, in [26], the authors adopt a greedy strategy for selecting views.

The view selection algorithm ViewSelect in [26] can be described as follows. Let $\mathcal{U}$ be the current set of views selected. ViewSelect will select the next view to be added to $\mathcal{U}$ by using function EstimateCost to simulate the actual top-k query Q on the *histograms* [33] of the views in $\mathcal{U}$ and those of the remaining views. If there is no view which can improve the cost of the current set of views, the algorithm stops and returns the current set of views selected.

Since each call of the EstimateCost sub-procedure again involves solving LP problems against the histograms of the corresponding cached views, the computational cost for view selection turns out to be very high. In Section 3.2 we will first improve LPTA by removing many of the calls to the LP solver. Then in Section 3.3, we will show how we could use an LPTA-based algorithm for handling the general kQAV problem with top-k views.

3.2 Algorithm LPTA$^+$

The original LPTA algorithm relies heavily on repeatedly

invoking the LP solver for both view selection and query processing, since the number of times the LP solver will be invoked is proportional to the number of calls to the LPTA algorithm (on both views and histograms) multiplied by the number of tuples accessed from the views/histograms. This is especially problematic when the dimensionality is high, since the cost of LP solver increases significantly as dimensionality grows.

To test this intuition, we conducted a preliminary experiment to measure the relative contribution of the LP solver and other operations to the overall cost. For a randomly generated dataset, where each attribute value of a tuple is chosen randomly from a uniform distribution, Figure 4 shows how query processing cost increases as dimensionality increases. The results were obtained by selecting from a pool of 100 randomly generated views, and by averaging the time of processing 100 randomly generated top-10 queries, with all views cached in memory. As can be seen from the figure, the processing cost of the LP solver dominates the cost of other operations in the LPTA algorithm. As the dimensionality grows, the cost of the LP solver increases quickly while the cost of other operations remains essentially constant.

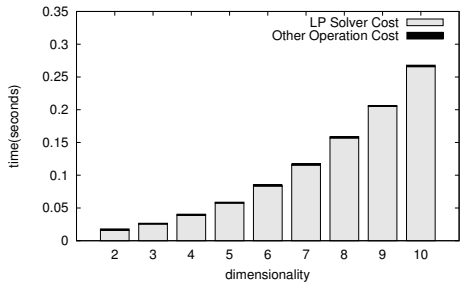


Figure 4: Query processing cost of LPTA as the dimensionality increases.

An important question is whether all these invocations of the LP solver are actually necessary. We will soon see that, by taking advantage of the fact that the views are cached in memory and so can be accessed sequentially with very small overhead, it will be sufficient to solve the LP problem just a few times for most executions of the LPTA algorithm.

To see this, we need to first to discuss how an LP solver works. We assume in this paper that the LP solver is based on the SIMPLEX algorithm [24], which is the most widely used LP algorithm. The general SIMPLEX algorithm usually works in two phases. The goal of the first phase is to find one *feasible solution*[2] to the original problem, while the goal of the second phase is to find the optimal solution to the original problem. Because the formulation of our problem as represented by Formula (1) is in *standard maximization form* [24] (i.e., there are no constraints of the form $w_1t[1] + \cdots + w_mt[m] \geq \theta$ except the non-negative variable constraints), the first phase of finding a feasible solution is essentially trivial. Thus we need to concentrate on the second phase of the SIMPLEX algorithm.

We call each non-zero variable in a feasible solution a *basic solution variable* or BSV. In order to obtain the optimal solution in the second phase, we use the *pivoting* technique, which essentially replaces one BSV by a variable which is not currently a BSV, in the hope that the target value τ can be increased.

[2]A feasible solution to an LP problem is a solution which satisfies all the constraints.

Now recall from the LPTA algorithm in Section 3.1 that for every r tuples read, we need to solve a new LP problem. An interesting characteristic of this process is that, for every LP problem formulated, the only change is in the *Right Hand Side* (RHS) of Formula 1, specifically $f_i(\overline{t_i})$; other parts of the constraints remain the same.

This characteristic motivates us to consider the following improvement to the LPTA algorithm. As before, we start by solving the LP problem once for the first set of r tuples accessed, deriving the BSVs for the optimal solution in the process. Then, when new tuples are accessed, we can reuse the previously derived BSVs, and check whether they lead to the optimal solution. If they do, then we have obtained the optimal solution for the new LP problem without exploring different possible BSVs using pivoting, which can be very costly [24]. The check above can be done more efficiently than pivoting. We note that this technique is different from previous work on *Incremental Linear Programming* [17], where the focus is on the more general problem of adding/removing/updating constraints.

The intuition behind the above optimization can also be illustrated using geometric properties. Consider the 2-dimensional example in Figure 5. Let t_1 and t_2 be the last two tuples accessed from V_1 and V_2 respectively. The optimal solution for the LP problem in Formula (1) can be obtained at vertex c of the convex polytope $Oacb$ in Figure 5 (a). Since the values of c on dimensions A and B are both positive, we know that A and B are the BSVs of the optimal solution. After we have accessed two new tuples t_3, t_4 from V_1, V_2, we need to shift the two edges ac and bc of the convex polytope down and left to $a'c'$ and $b'c'$, as shown in Figure 5 (b). Given the fact that the score functions of the cached views are all monotone, it is very likely that, for the new convex polytope $Oa'c'b'$, the optimal solution will be at the vertex c', which again has positive A and B values, and thus corresponds to the same BSVs. This shows that the optimal solution corresponding to the new tuples can be obtained by choosing the same set of BSVs in the LP problem, i.e., we do not need to repeat the pivoting steps to find the optimal BSVs.

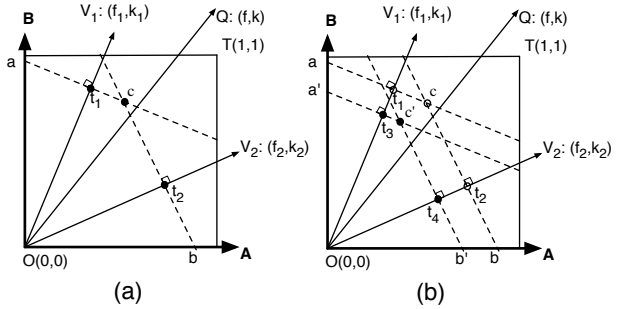


Figure 5: Example of LPTA$^+$.

The pseudocode of the new LPTA$^+$ algorithm is shown in Algorithm 2. Compared with LPTA, the difference lies in how the τ value is calculated (lines 8–15). For the first set of tuples accessed, we run the LP solver and derive the corresponding optimal BSVs B and τ (lines 8–9). After that, in each iteration we check whether re-pivoting is needed by using the function *isValidOptimal* to verify whether the exist-

ing BSVs lead to a new optimal solution (lines 11–12); if they do, we derive τ directly, otherwise we solve the LP problem again and derive the new B and τ. Function *isValidOptimal* basically pushes variables in B directly into the BSVs of the SIMPLEX algorithm, and checks whether it forms a valid solution considering the new RHS vector. The overhead of this operation is small and can clearly avoid many unnecessary pivoting steps in the SIMPLEX algorithm.

Algorithm 2: LPTA$^+$($\mathcal{U} = \{V_1, \ldots, V_r\}$, $Q = (f, k)$)

```
1  X ← an empty priority queue;
2  τ ← ∞, B ← nil;
3  repeat
4      {t̄_1, ..., t̄_r} ← getNextTuple(U);
5      retrieveTupleInfo({t̄_1, ..., t̄_r});
6      X.insert({t̄_1, ..., t̄_r});
7      X.keepTop(k);
8      if B is nil then
9          Compute the optimal BSVs B and τ using an
           LP solver;
10     else
11         derive new RHS vector b using {t̄_1, ..., t̄_r};
12         if isValidOptimal(U, B, b) then
13             derive the new τ directly;
14         else
15             Compute the optimal BSVs B and τ using
               an LP solver;
16 until noNewTuple(U) or (|X| = k and f(X[k]) ≥ τ);
```

Since LPTA$^+$ improves only the efficiency of calculating τ, we know that both LPTA and LPTA$^+$ will examine the same number of tuples from $\mathcal{U}$. As we will demonstrate in the experiments, the reuse of BSVs in LPTA$^+$ usually has a very small cost, and thus by avoiding many unnecessary pivoting steps, LPTA$^+$ can be much more efficient than LPTA in practice.

3.3 Handling the General kQAV Problem

Although LPTA$^+$ can improve the efficiency of LPTA, we still need to extend it to handle the general kQAV problem, where we have only top-k views rather than complete rankings of tuples, and no base views.

Our first observation is that, given a fixed set of views $\mathcal{U} = \{V_1, \ldots, V_r\}$, we can find all the certain tuples from $\mathcal{U}$ by using the LPTA$^+$ algorithm with the following simple modifications: (1) if the algorithm stops before all tuples in $\mathcal{U}$ are exhausted, we have already found a set of top-k certain answers for the query, since every possible unseen tuple will have a value no better than the current top-k results; (2) if we have exhausted all tuples in $\mathcal{U}$, let τ be the threshold value derived from the last tuple of each view; if we remove from the candidate top-k queue all tuples which have value smaller than τ, then the remaining tuples in the queue are guaranteed to be certain answers. Similar to the first case, the pruning of the tuples in the candidate top-k queue here is sound because τ indicates the maximum value that can be achieved by an unseen tuple, say t. Every tuple t' which is pruned has a value less than τ, so there exists a possible relation instance R which is score consistent with $\mathcal{U}$, and at the same time contains an unlimited supply of tuples that have the same attribute values as t. Thus t' cannot become a top-k result for this R since it will be dominated by t.

Now one question is whether, given a set of cached views, we can find a minimal subset of views which can give us the *maximum* set of certain answers to the query $Q = (f, k)$ (up to a total of k). Unfortunately as discussed in [26], an obvious algorithm to determine the best subset of views has a high complexity since we need to enumerate all possible combinations of r views. Instead, following the heuristics proposed in [26], we propose the modification to the LPTA/LPTA$^+$ algorithm described below. This modification guarantees that we will find the maximum set of certain answers to Q and that its complexity is linear in the number of views, but it does not guarantee that the number of views used is minimal.

Consider the second case above (we do not need any changes to the first case since that already finds a set of top-k certain answers). Instead of pruning tuples which have a value less than τ, we keep these candidate tuples and iteratively consider each of the remaining views. For each view $V' \in \mathcal{V} - \mathcal{U}$, we investigate all tuples in V' one by one, replacing existing candidate tuples with them whenever they have higher value with respect to Q; meanwhile, we try to refine the threshold value τ by considering the last tuple accessed in V'. During this process, if we have k candidate answers which have value larger than or equal to τ, we know we have found the top-k certain answers; otherwise, if all views have been exhausted, we can get the maximum set of certain answers by pruning from the candidate queue those which have value less than τ.

It is straightforward to see that the above heuristic, when used in conjunction with LPTA instead, gives us a procedure for finding all certain answers to Q (up to a maximum of k). *Thus, LPTA can be used to find certain answers even when base views or complete tuple rankings are not available.*

4. IV-INDEX BASED TOP-K QAV

Though the LPTA$^+$ algorithm proposed in Section 3 improves the efficiency of the original LPTA algorithm by avoiding unnecessary pivoting operations, the algorithm still needs to invoke the LP solver multiple times, during both view selection and query processing. When the underlying relation has high dimensionality, the cost of LP solver calls can be considerable. This motivates the quest for an even more efficient algorithm for finding the certain answers.

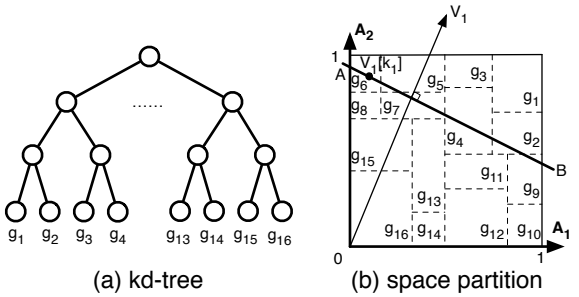
To this end, we propose a simple index structure, called the *Inverted View Index* (IV-Index). Using this index greatly reduces the number of invocations of the LP solver, allowing all certain answers in $Q(\mathcal{V})$ to be returned quickly.

4.1 Inverted View Index

Given the set $\mathcal{V}$ of cached views, we first collect all tuples in these views into an *Inverted View Index* (IV-Index) $\mathcal{I} = (\mathcal{T}, \mathcal{H}_V, \mathcal{H}_t)$. The components of the index are as follows: $\mathcal{H}_t$ is a lookup table which returns the attribute value information for a tuple given its id; $\mathcal{H}_V$ is a lookup table which returns the definition of a view, and $\mathcal{T}$ is a high-dimensional data structure. In this work, we utilize a kd-tree as the underlying high-dimensional data structure as it has been shown to have the most balanced performance compared with other high-dimensional indexing structures [19]. However, we note that the techniques we propose can be easily adapted to utilize *quad-trees* or other indexing struc-

tures.

Each node g in the kd-tree $\mathcal{T}$ represents an m-dimensional region, with the root node g_{root} of $\mathcal{T}$ representing the entire region from $(0, \ldots, 0)$ to $(1, \ldots, 1)$. The kd-tree is built as follows. Starting from the root node, we recursively partition the region associated with the current node g into two parts based on a selected dimension and a splitting hyperplane. These two sub-regions are represented by two nodes which will become the children of g in $\mathcal{T}$. Once this recursive process has completed, the disjoint regions represented by the leaf nodes of $\mathcal{T}$ form a partitioning of the whole m-dimensional space. An example of a kd-tree along with the partitioning is shown in Figure 6.



(a) kd-tree (b) space partition

Figure 6: Example of (a) a kd-tree, and (b) the corresponding partition of 2-dimensional space.

For a node g, without ambiguity, we also use g to denote the region associated with the node. To facilitate query processing, we associate each leaf node g of $\mathcal{T}$ with a set T_g of tuple ids (tids), corresponding to tuples in the cached views that belong to g. Given a node (region) g, let the value range of g on each of the m dimensions be $[g_l^1, g_u^1]$, ..., $[g_l^m, g_u^m]$, and let $t_g^{\vdash} = (g_l^1, \ldots, g_l^m)$ and $t_g^{\dashv} = (g_u^1, \ldots, g_u^m)$. Then for any monotone function f, it is clear that the maximum (minimum) value that can be achieved by any tuple in g is $f(t_g^{\dashv})$, (resp., $f(t_g^{\vdash})$).

Since the set of top-k views cached in the memory may not cover the complete set of tuples in the database, it is clear that we may only have "partial" knowledge about regions associated with some leaf nodes in $\mathcal{T}$. Let R be *any* relation that is score consistent with $\mathcal{V}$. Given a region g, let R_g denote the set of tuples in R whose values fall inside g. Then we say that a region g is *complete*, or $\kappa(g) = true$, if $T_g = R_g$ for every score-consistent relation R; otherwise we say that g is *partial*, or $\kappa(g) = false$. This is a semantic property and it is expensive to check it directly. A sufficient condition for checking the completeness of a region g is given in the following lemma.

LEMMA 1. *A region g is complete if there exists a top-k cached view $V_i = (f_i, k_i)$ in $\mathcal{V}$ for which $f_i(V_i[k_i]) < f_i(t_g^{\vdash})$.*

PROOF. If $f_i(V_i[k_i]) < f_i(t_g^{\vdash})$, then clearly for any score-consistent relation R, $\forall t \in R_g$, $f_i(V_i[k_i]) < f_i(t)$. So according to the definition of top-k cached view, all tuples in R_g must belong to V; hence $R_g = T_g$. □

A 2-dimensional example of Lemma 1 is shown in Figure 6 (b). This example shows a vector $\vec{f_1}$ corresponding to a view $V_1 = (f_1, k_1)$ along with the k_1'th tuple $V_1[k_1]$ from the view. If we draw a line AB through $V_1[k_1]$ which is perpendicular to $\vec{f_1}$, we can observe that $t_{g_1}^{\vdash}$ and $t_{g_3}^{\vdash}$ are above AB; thus $f_1(V_1[k_1]) < f_1(t_{g_1}^{\vdash})$, $f_1(V_1[k_1]) < f_1(t_{g_3}^{\vdash})$, and g_1, g_3 are complete. On the other hand, $f_1(V_1[k_1]) > f_1(t_{g_2}^{\vdash})$, so if the only top-$k$ cached view we have is V_1, we are not able to determine whether g_2 is complete or not. This is because we do not have enough information about the part of g_2 which is below AB. If R contains no tuple which falls inside this region, g_2 is complete; however, if R does contain tuples which fall inside this region, g_2 is partial.

We note that it is not possible to derive a necessary and sufficient condition for checking the completeness of a region given only the top-k cached views. This is because we will have to consult the original database R to check whether the regions which cannot be decided using Lemma 1, e.g., g_2 in above example, are complete or not. Obviously this process can be expensive, and more importantly, it is against the purpose of our kQAV framework which is to answer queries using only top-k cached views. So we will simply label regions whose completeness cannot be decided by Lemma 1 as partial. Alternative weaker sufficient conditions for completeness checking are left as future work.

Consider a partial leaf node g in $\mathcal{T}$ for a top-k cached view $V_1 = (f_1, k_1)$. If $f_1(t_g^{\vdash}) \leq f_1(V_1[k_1]) \leq f_1(t_g^{\dashv})$ (i.e., the hyperplane which crosses $V_1[k_1]$ and is perpendicular to $\vec{f_1}$ intersects with g), we will store a pair $p = (V_1, V_1[k_1].id)$ in a *cross view* set P_g associated with g. In p, the first entry is a pointer to the definition of V_1, while the second entry is the tuple id of $V_1[k_1]$. If no such views exist, i.e., the view is complete, $P_g = \emptyset$. Consider the example of Figure 6 (b), and suppose that V_1 is the only top-k cached view. Then $(V_1, V_1[k].id)$ is in P_{g_2} as well as in P_{g_4},P_{g_5},P_{g_6},P_{g_7} and P_{g_9}.

4.2 IV-Search Algorithm

Given an IV-Index $\mathcal{I}$, a top-k query $Q = (f, k)$ can be answered by traversing the corresponding kd-tree of $\mathcal{I}$ using a strategy such as *best-first search* [40].

The pseudocode of our first algorithm, called IVS-Eager, is given in Algorithm 3. The algorithm traverses the kd-tree $\mathcal{T}$ by visiting first those nodes which have larger maximum value with respect to Q (lines 3–17), as indicated by $f(t_g^{\dashv})$, since these nodes may have good potential to contain tuples which have high value with respect to Q. If the current node g is a leaf node, then we extract all tuples within g and check whether they can become new candidate top-k results (lines 9–11). In addition, if a leaf node g is partial, we need to collect information from P_g, which defines the region of the unseen tuples which cannot be covered by the top-k cached views, and solve a linear programming problem to find the maximum value that can be achieved by any unseen tuples in g (lines 12–13). Finally, if the current node g has its maximum value $f(t_g^{\dashv})$ less than or equal to $f(X_r[k])$, which is the value of the kth candidate tuple in X_r, the algorithm can stop, since according to the best-first search strategy, any unseen nodes cannot contain a tuple which is better than $X_r[k]$ (line 14).

The correctness of IVS-Eager follows from the best-first search strategy, since every unseen tuple will have value smaller than $X_r[k]$ with respect to Q. In addition, the updating of the threshold value τ ensures that every tuple returned is a certain answer.

One inefficiency in IVS-Eager is that, for every partial leaf node encountered, we need to invoke an LP solver to update the threshold value τ. This can be expensive for the following two reasons: first, as shown in the example of Figure 6

Algorithm 3: IVS-Eager($\mathcal{I}$=($\mathcal{T}, \mathcal{H}_V, \mathcal{H}_t$),$Q = (f, k)$)

```
1  X_n ← an empty priority queue for kd-tree nodes;
2  X_r ← an empty priority queue for candidate results;
3  X_n.enqueue(g_root, f(t⁺_{g_root}));
4  τ ← ∞;
5  while ¬X_n.isEmpty() do
6      g ← X_n.dequeue();
7      τ ← min(τ, f(t⁺_g));
8      if isLeaf(g) then
9          foreach t ∈ g do
10             X_r.enqueue(t, f(t));
11         X_r.keepTop(k);
12         if ¬κ(g) then
13             τ ← min(τ, LPSolve(P_g, Q));
14         if |X_r| = k ∧ f(X_r[k]) ≥ f(t⁺_g) then break;
15     else
16         foreach g_c ∈ children(g) do
17             X_n.enqueue(g_c, f(t⁺_{g_c}));
18 return {t | t ∈ X_r ∧ f(t) ≥ τ};
```

Algorithm 4: IVS-Lazy($\mathcal{I} = (\mathcal{T}, \mathcal{H}_V, \mathcal{H}_t)$, $Q = (f, k)$)

```
1  X_n ← an empty priority queue for kd-tree nodes;
2  X_r ← an empty priority queue for candidate results;
3  C_n ← an empty cache for partial leaf nodes;
4  X_n.enqueue(g_root, f(t⁺_{g_root}));
5  τ ← ∞;
6  while ¬X_n.isEmpty() do
7      g ← X_n.dequeue();
8      τ ← min(τ, f(t⁺_g));
9      if isLeaf(g) then
10         foreach t ∈ g do
11             X_r.enqueue(t, f(t));
12         X_r.keepTop(k);
13         if ¬κ(g) then C_n.add(P_g);
14         if |X_r| = k ∧ f(X_r[k]) ≥ f(t⁺_g) then break;
15     else
16         foreach g_c ∈ children(g) do
17             X_n.enqueue(g_c, f(t⁺_{g_c}));
18 P ← consolidateCrossViewSets(C_n);
19 τ ← min(τ, LPSolve(P, Q));
20 return {t | t ∈ X_r ∧ f(t) ≥ τ};
```

(b), each top-k cached view might be stored in the cross view set of many nodes, so there might be duplicated computation if we solve the LP problem for every node individually; second, when the dimensionality is high, the number of such partial nodes will be large. In the following, we propose another algorithm, called IVS-Lazy, which needs to solve only one (potentially larger) LP problem.

Algorithm 4 lists the pseudocode of IVS-Lazy. The difference with IVS-Eager is that whenever a partial leaf node g is encountered in IVS-Lazy, we store the cross view set of g in a cache C_n (line 13) rather than immediately solve the LP problem and update the threshold τ as is done in IVS-Eager. After we have exhausted all nodes in the kd-tree, we collect all the view information in C_n and solve a single LP problem (lines 18–19).

4.2.1 Further Optimization via View Pruning

As can be observed from Algorithms IVS-Eager and IVS-Lazy, a critical operation in both algorithms is to collect constraints from the cross view set(s), and solve the LP problem given the query and constraints. Since the complexity of an LP problem may increase considerably with respect to the number of constraints, pruning constraints which are not useful can be very important for the overall query performance.

Let $Q = (f, k)$ be the query to be processed, and assume that we have accessed more than k tuples, i.e., $|X_r| \geq k$, using each of the two IV-Index based search algorithms. Now consider the point at which we solve the LP problem, i.e., line 13 in IVS-Eager and line 19 in IVS-Lazy. Let $t_{min} = X_r[k]$ be the current kth tuple in X_r, and let $V = (f', k')$ be a view from the corresponding cross view set. According to the definition, for any tuple $t \notin V$, we have $f'(t) \leq f'(V[k'])$, so the maximum value that can be achieved by any such tuple can be calculated using the following LP problem:

$$\begin{aligned} \max_{t} \quad & \phi = f(t) \\ \text{subject to:} \quad & f'(t) \leq f'(V[k']) \\ & 0 \leq t[i] \leq 1, i = 1, \ldots, m \end{aligned} \tag{2}$$

Let $f(t) = w_1 t[1] + \cdots + w_m t[m]$, and $f'(t) = w'_1 t[1] + \cdots + w'_m t[m]$. A careful inspection of the above LP formulation will reveal that it is exactly the *Fractional Knapsack Problem* (or *Continuous Knapsack Problem*) [35]. In this problem, we are given a set of items $o_1, \ldots, o_m$, where each item o_i, $1 \leq i \leq m$, has weight w'_i and value w_i, and we are asked to pack them into a knapsack with maximum weight $f'(V[k'])$ such that the total value is maximized, while allowing fractions of an item to be put into the knapsack.

It is well known that a greedy algorithm which accesses items ordered by *utility* (value divided by weight) finds the optimal solution for the fractional knapsack problem, in linear time. Thus we utilize the following Algorithm FKP to find the maximum value ϕ which can be achieved by an unseen tuple with respect to a view V and a query Q. If this value ϕ is less than $f(t_{min})$ (the value of the current kth tuple in X_r), we can safely prune V from consideration in both IVS-Eager and IVS-Lazy when checking cross view sets.

4.3 Discussion

Since we usually prefer the cached views to reflect the most recent and popular queries, and the memory consumption of the index structure needs to be bounded, a mechanism for cache replacement is necessary. There is much previous work on good strategies for cache/buffer replacement [34, 23], so in this work we will assume that a cache replacement strategy has been specified. Instead, we will only discuss how the basic operations of inserting and deleting a view might be implemented using the IV-Index.

To handle view insertion and deletion, we could associate with each *tuple* t cached in the memory a count $c(t)$, indi-

Algorithm 5: FKP($Q = (f, k)$, $V = (f', k')$)

```
1 l ← {(i, u_i ← w_i/w'_i) | 1 ≤ i ≤ m};
2 Sort tuples in l based on utility;
3 φ ← 0, B ← f'(V[k']);
4 for (i, u_i) ∈ l do
5     if w'_i ≥ B then
6         φ ← φ + u_i B;
7         break;
8     else φ ← φ + w_i, B ← B − w'_i;
9 return φ;
```

cating how many views contain t. In addition, we could associate with each *node* g a count $c(g)$, specifying how many views cover g, or make g a complete node, according to Lemma 1. First consider inserting a new top-k cached view $V = (f, k)$. For each tuple $t \in V$, we set $c(t) = c(t) + 1$ and insert t into the kd-tree if necessary. To change the completeness status of nodes affected by V in the kd-tree, we could use a best-first strategy to find each node g for which $f(t_g^{\dashv}) > f(V[k])$, and set $c(g) = c(g) + 1$. Similarly, when deleting a top-k cached view $V = (f, k)$, we could use a best-first strategy to find each node g for which $f(t_g^{\dashv}) > f(V[k])$, and set $c(g) = c(g) - 1$. In addition, we find each cached tuple $t \in V$ for which $f(t) > f(V[k])$, set $c(t) = c(t) - 1$ and remove it from cache when $c(t) = 0$.

5. EMPIRICAL RESULTS

In this section, we study the performance of various algorithms for the kQAV problem based on one real dataset of NBA statistics and four synthetic datasets. The goals of our experiments are to study: (i) the performance of the LPTA-based algorithms, and by how much LPTA$^+$ improves the state-of-the-art LPTA algorithm; (ii) the relative performance of the lazy and eager versions of the IV-Index-based algorithm, and to what extent they outperform LPTA+; (iii) the effectiveness of the pruning process proposed in Section 4.2.1. We implemented all the algorithms in Python, and all experiments were run on a Linux machine with a 4 Core Intel Xeon CPU, OpenSuSE 12.1, and Python 2.7.2.

The NBA dataset is collected from the Basketball Statistics website [7], which contains the career statistics information of NBA players until 2009. The NBA dataset has 3705 tuples and we selected 10 attributes to be used in our experiments. The synthetic datasets are generated by adapting the benchmark generator proposed in [20]. The *uniform* (UNI) dataset and the *powerlaw* (PWR) dataset are generated by considering each attribute independently. For UNI, attribute values are sampled from a uniform distribution, and for PWR, attribute values are sampled from a power law distribution with $\alpha = 2.5$ and normalized into the range $[0, 1]$. In the *correlated* (COR) synthetic dataset, values from different attributes are correlated with each other, while in the *anti-correlated* (ANT) synthetic dataset, values from different attributes are anti-correlated with each other. Each synthetic dataset is over 10 attributes and has 100000 tuples.

Weights for the score functions in all views are generated randomly, and all views are cached in memory. Similar to previous work on LPTA-based algorithms [26], the size of the histograms used for estimation is set to be roughly 1% of the size of the corresponding dataset. For the IV-Index-based approach, we set the number of tuples in the leaf nodes of the kd-tree to be less than or equal to 50. Alternative configurations for the kd-tree were also tested with similar results and so are omitted here for lack of space. Finally, the query score functions are also generated randomly, and all results reported here are based on an average of the results from processing 100 queries.

5.1 LPTA-based Algorithms

In Figure 7, we compare the performance of LPTA and LPTA$^+$ for queries which ask for the top-100 tuples using a set of 100 views. Figure 7 (a–e) considers the setting in which each view contains 1000 tuples. We can see that, for all five datasets, LPTA$^+$ is much faster than LPTA in most cases. Similar results are obtained for the setting in which each view contains 100 tuples (Figure 7 (f–j)). However, we note that for this setting, query processing time is longer because now the views contain fewer tuples, so we need to check more additional views in order to guarantee that we will find all the certain answers in $Q(\mathcal{V})$ w.r.t. the query Q.

In Figure 8 (a), we compare the performance of LPTA and LPTA$^+$ when varying the number of views in the cache pool. Here we fix the number of dimensions at 5, and consider queries where k is randomly selected from 10 to 100. As can be seen from this figure, the performance of both algorithms degenerates as the number of views increases. However, LPTA$^+$ is still twice as fast as LPTA in most settings. This result was obtained using the RND dataset. Very similar results were obtained for the other datasets and for different dimensionality settings, and are thus omitted.

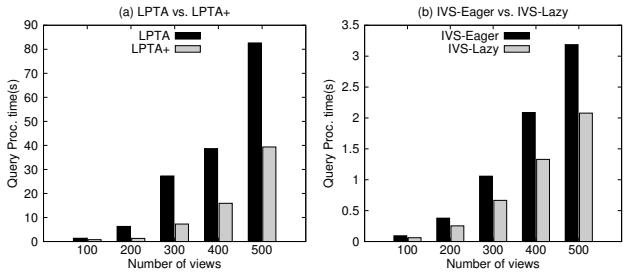


Figure 8: When varying the number of views on the RND dataset, the performance comparison between: (a) LPTA and LPTA$^+$; (b) IVS-Eager and IVS-Lazy.

In Figure 9 (a), we compare the performance of LPTA and LPTA$^+$ when varying the value k in each query from 10 to 100, given 100 views and by fixing the the dimensionality at 5. Similar to the previous results, the performance of both algorithms degenerates as k increases, but LPTA$^+$ is still faster than LPTA for all settings. The results obtained for datasets other than RND are very similar. We discuss Figures 8(b) and 9(b) below.

Although LPTA$^+$ can greatly outperform LPTA, it can be observed that the query processing cost for LPTA$^+$ is still high.

5.2 IV-Index-based Algorithms

Figure 10 shows the experimental results of the IVS-Eager and IVS-Lazy algorithms under the same settings as in Figure 7. Compared with the results of LPTA-based algorithms in Figure 7, we can readily see that the IV-Index-based ap-

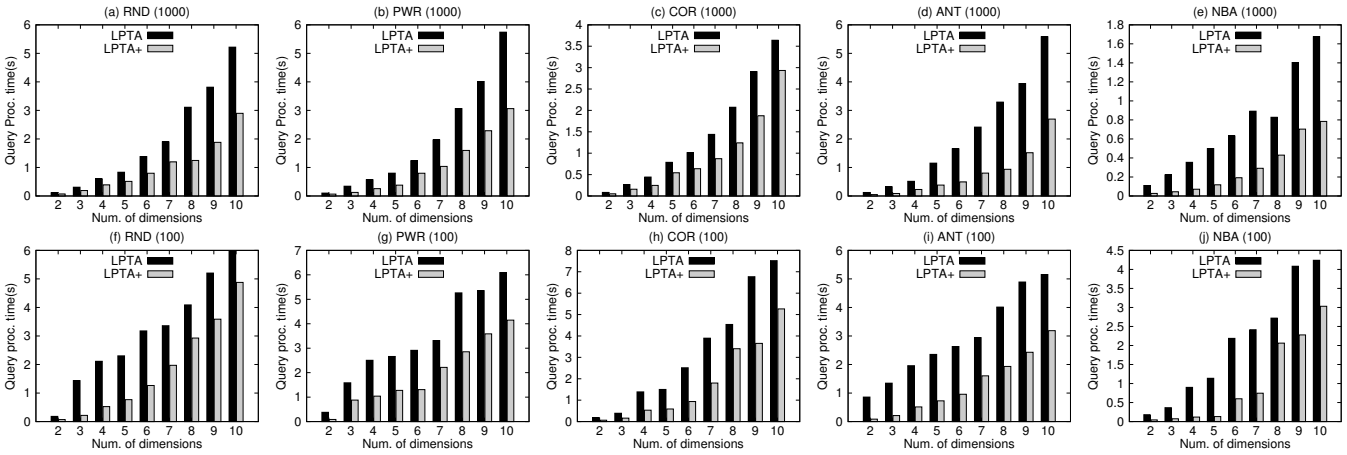


Figure 7: LPTA vs. LPTA$^+$: (a–e) results on 5 datasets with each view containing 1000 tuples; (f–j) results on 5 datasets with each view containing 100 tuples.

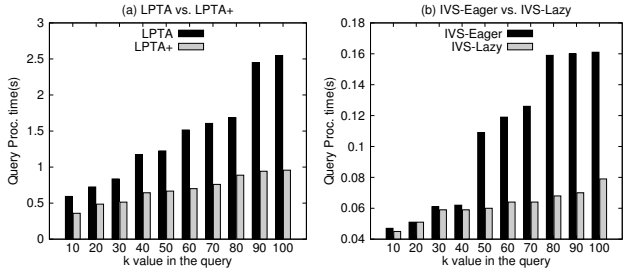


Figure 9: When varying the value k of a query on the RND dataset, the performance comparison between: (a) LPTA and LPTA$^+$; (b) IVS-Eager and IVS-Lazy.

proaches are orders of magnitude faster than the LPTA-based approaches under all circumstances. From Figure 10 (a–j), we can also observe that, in most cases, IVS-Lazy is much faster than IVS-Eager, since it saves many calls to the LP solver. The only exception is for low-dimensional cases where both algorithms have a very small query processing cost. The advantage of IVS-Lazy especially applies for the high-dimensional cases where more nodes in the kd-tree are partial. Similar to the results of the LPTA-based algorithms, both IVS-Eager and IVS-Lazy are much faster on views which contain more tuples, simply because they need to check fewer partial nodes in the kd-tree.

When we vary the number of views and when we vary the number k in each query, as can be observed from Figure 8 (b) and Figure 9 (b), the performance of IV-Index-based algorithms are orders of magnitude faster than the LPTA-based algorithms. The running time of both IVS-Eager and IVS-Lazy increases as the number of views increases, or as k increases, as with all algorithms. However, IVS-Lazy has consistently better performance than IVS-Eager.

5.3 Effectiveness of Pruning

Finally, in Figure 11, we show the effectiveness of the pruning techniques proposed in Section 4.2.1. In this experiment, we fix the number of tuples in each view to be 100, and for each query $Q = (f, k)$, k is a random number within $[10, 100]$; for other settings of these parameters, the results obtained are very similar. As can be seen from the figure, the pruning technique can improve the performance of both IV-Search algorithms. Notice that for various dimensionality settings, the overall performance of IVS-Lazy/Pruning is consistently the best on all five datasets.

6. RELATED WORK

For general top-k query processing, the most popular approach is the *Threshold Algorithm* (TA) / *No Random Access Algorithm* (NRA) as proposed by Fagin et al. in [27]. While TA and NRA differ in whether random access to the database is allowed, this family of algorithms usually share a similar query processing framework which accesses tuples from the database in a certain order, while maintaining an upperbound on the maximum value that can be achieved by the tuples that have not yet been accessed. If the current top-k result has a value no less than the best value achievable by any unseen tuple, the algorithm can stop. Recently, various improvements to the original algorithms such as the *Best Position Algorithm* [16] have been proposed, while variations of top-k queries such as *Rank Join* [31] and *Continuous Top-k Queries* [45] have been studied. Finally, Li et al. study how top-k algorithms might be implemented in a relational database [39]. An excellent survey on top-k query processing can be found in [32].

Hristidis et al. [30] first considered the problem of using views to speed up top-k query processing. They focused on finding one best view which can be used for answering a query. As mentioned in [26], their setting is quite restrictive as it cannot exploit multiple views, and it also assumes that all attributes of the underlying base table are always utilized for all top-k queries. Das et al. [26] propose a novel algorithm, called LPTA, which overcomes the limitations of [30] by utilizing multiple views for answering a top-k query.

It can be verified that the kQAV problem defined here is a generalization of the kQAV problem as considered in [26]. This is because the core techniques proposed in [26] rely on the assumption that either each top-k view $V_i = (f_i, k_i) \in \mathcal{V}$ is a complete ranking of all tuples in R, i.e., $k_i = |R|$; or the *base views*, which are complete rankings of all tuples in R according to the values of each attribute, are available. We make no such assumptions in our setting. That said, we can

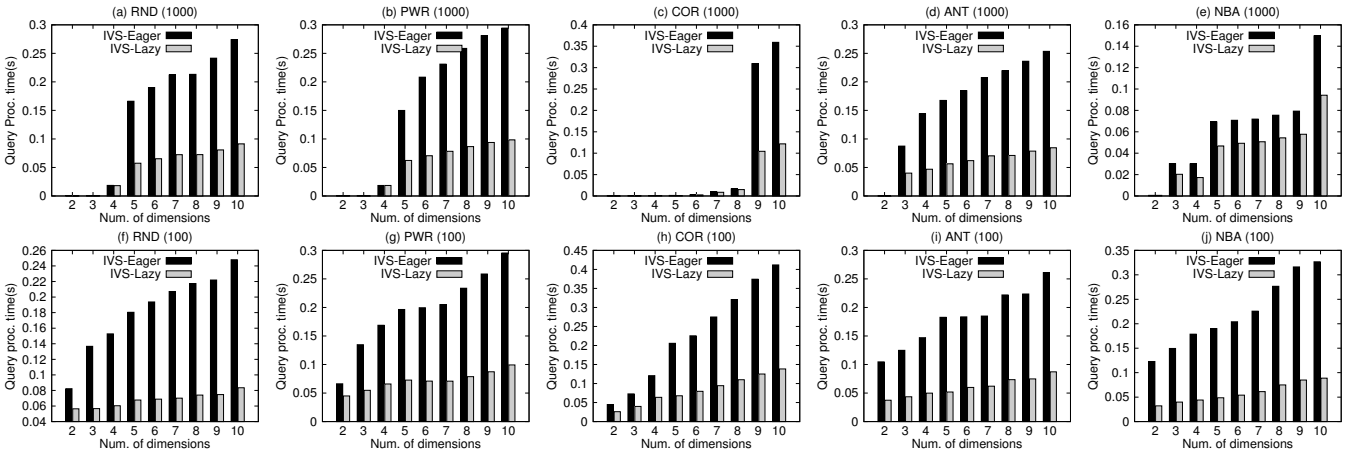


Figure 10: IVS-Eager vs. IVS-Lazy: (a–e) results on 5 datasets with each view containing 1000 tuples; (f–j) results on 5 datasets with each view containing 100 tuples.

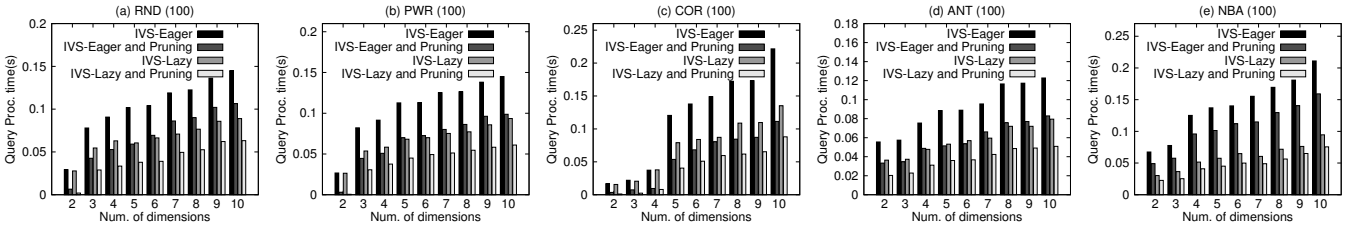


Figure 11: Pruning effectiveness test of IV-Search algorithms based on the five datasets.

easily adapt our algorithms to work in settings where we do have base views available or all views are complete rankings of all tuples.

In [18], the authors consider the problem of whether a top-k query can be answered exactly using a set of top-k views, which resembles the classical query containment problem in databases [21]. However, this work does not address the general kQAV problem, i.e., return a maximum set of certain answers, in case $\mathcal{V}$ cannot answer a query Q exactly. Ryeng et al. [41] extend the techniques proposed in [26] and [18] to answer top-k queries in a distributed setting. They assume that access to the original database is available through the network interface, thus exact top-k answers can always been found by forming a "remainder" query which can be utilized to fetch tuples not available in the views. We note that the focus of our work is on efficient algorithms for finding answers to the kQAV problem, where the original database is not accessible. Should it be accessible, we can adapt the techniques proposed in [41] to find the additional answers in the case where our algorithms cannot find enough certain tuples from the cached views.

In addition to leveraging views, an alternative way of optimizing top-k query processing is through a *Layered Index* [22, 44, 29, 37]. These approaches try to organize tuples in the database into an layered index structure. We can quickly obtain the answers to a top-k query by accessing just the first few layers of the index. First, we note that our proposed IV-Index is significantly different from the layered index, since it is based on a standard space partitioning index such as kd-tree. Furthermore, these layered indexes all assume access to the original database is available, so are difficult to adapt to scenarios where we have no access to the database.

7. CONCLUSION

In this paper, motivated by many real applications, we considered the problem of top-k query answering using cached views, where each view is a top-k view for some k. To address this problem, we first considered the state-of-the-art LPTA algorithm as proposed in [26]. The performance of LPTA suffers because of iterative calls to a linear programming sub-procedure, which can be especially problematic when the number of views is large or if the dimensionality of the dataset is high. By observing an interesting characteristic of the LPTA framework, we proposed LPTA$^+$ which has greatly improved efficiency over LPTA. We adapted both algorithms so they work in our kQAV setting, where views are not complete tuple rankings and base views are not available. Furthermore, we proposed an index structure, called IV-Index, which stores the contents of all cached views in a central data structure in memory, and can be leveraged to answer a new top-k query much more efficiently compared with LPTA and LPTA$^+$. Using comprehensive experiments, we showed LPTA$^+$ substantially improves the performance of LPTA while the algorithms based on IV-Index outperform both these algorithms by a significant margin. As future work, it is important to study how to further optimize the IV-Index based approach, e.g., by investigating alternative weaker sufficient conditions for completeness checking for nodes. It is interesting to consider how the proposed algorithms could be adapted for more complex top-k querying frameworks such as Rank Join.

8. ACKNOWLEDGMENTS

This research was supported in part by a grant from NSERC (Canada). The first author's research was also supported by the Four Year Doctoral Fellowship from the University of British Columbia.

9. REFERENCES

[1] Auto123 consumer car ratings. http://www.auto123.com/en/car-reviews/consumer-ratings/.
[2] Flickr. http://www.flickr.com.
[3] IMDB. http://imdb.com.
[4] Memcached. http://memcached.org.
[5] Metacritics. http://www.metacritic.com.
[6] Metascore. http://www.metacritic.com/about-metascores.
[7] Nba basketball statistics. http://www.databasebasketball.com.
[8] Rottentomatoes. http://www.rottentomatoes.com.
[9] Twitter. http://twitter.com.
[10] U.s. news best cars. http://usnews.rankingsandreviews.com/cars-trucks/rankings/cars/.
[11] U.s. news best collage rankings. http://www.usnews.com/rankings.
[12] Wikipedia. http://www.wikipedia.org.
[13] World university rankings. http://www.timeshighereducation.co.uk/world-university-rankings/.
[14] Youtube. http://www.youtube.com.
[15] S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In *PODS*, pages 254–263, 1998.
[16] R. Akbarinia, E. Pacitti, and P. Valduriez. Best position algorithms for top-k queries. In *VLDB*, pages 495–506, 2007.
[17] G. J. Badros, A. Borning, and P. J. Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, 2001.
[18] E. Baikousi and P. Vassiliadis. View usability and safety for the answering of top-k queries via materialized views. In *DOLAP*, pages 97–104, 2009.
[19] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.
[20] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.
[21] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.
[22] Y.-C. Chang, L. D. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: Indexing for linear optimization queries. In *SIGMOD*, pages 391–402, 2000.
[23] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *VLDB*, pages 127–141, 1985.
[24] G. Dantzig. *Linear Programming and Extensions*. Princeton University, 1998.
[25] G. Das, D. Gunopulos, N. Koudas, and N. Sarkas. Ad-hoc top-k query answering for data streams. In *VLDB*, pages 183–194, 2007.
[26] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *VLDB*, pages 451–462, 2006.
[27] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.
[28] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.
[29] J.-S. Heo, J. Cho, and K.-Y. Whang. The hybrid-layer index: A synergic approach to answering top-k queries in arbitrary subspaces. In *ICDE*, pages 445–448, 2010.
[30] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD*, pages 259–270, 2001.
[31] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765, 2003.
[32] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.
[33] Y. E. Ioannidis. The history of histograms. In *VLDB*, pages 19–30, 2003.
[34] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, pages 439–450, 1994.
[35] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.
[36] C. T. Kwok and D. S. Weld. Planning to gather information. In *AAAI/IAAI, Vol. 1*, pages 32–39, 1996.
[37] J. Lee, H. Cho, and S. won Hwang. Efficient dual-resolution layer indexing for top-k queries. In *ICDE*, pages 1084–1095, 2012.
[38] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, pages 95–104, 1995.
[39] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. RankSQL: Query algebra and optimization for relational top-k queries. In *SIGMOD*, pages 131–142, 2005.
[40] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, third edition, 2010.
[41] N. H. Ryeng, A. Vlachou, C. Doulkeridis, and K. Nørvåg. Efficient distributed top-k query processing with caching. In *DASFAA*, pages 280–295, 2011.
[42] D. Theodoratos and T. K. Sellis. Data warehouse configuration. In *VLDB*, pages 126–135, 1997.
[43] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava. Ranked join indices. In *ICDE*, pages 277–288, 2003.
[44] D. Xin, C. Chen, and J. Han. Towards robust indexing for ranked queries. In *VLDB*, pages 235–246, 2006.
[45] A. Yu, P. K. Agarwal, and J. Yang. Processing a large number of continuous preference top-k queries. In *SIGMOD*, pages 397–408, 2012.