PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries

Vagelis Hristidis
Dept. of Computer Science and Engineering
University of California, San Diego
La Jolla, CA92093
vagelis@cs.ucsd.edu

Nick Koudas
AT&T Labs-Research
koudas@research.att.com

Yannis Papakonstantinou
Dept. of Computer Science and Engineering
University of California, San Diego
La Jolla, CA92093
yannis@cs.ucsd.edu

ABSTRACT

Users often need to optimize the selection of objects by appropriately weighting the importance of multiple object attributes. Such optimization problems appear often in operations' research and applied mathematics as well as everyday life; e.g., a buyer may select a home as a weighted function of a number of attributes like its distance from office, its price, its area, etc.

We capture such queries in our definition of preference queries that use a weight function over a relation's attributes to derive a score for each tuple. Database systems cannot efficiently produce the top results of a preference query because they need to evaluate the weight function over all tuples of the relation. PREFER answers preference queries efficiently by using materialized views that have been preprocessed and stored.

We first show how the result of a preference query can be produced in a pipelined fashion using a materialized view. Then we show that excellent performance can be delivered given a reasonable number of materialized views and we provide an algorithm that selects a number of views to precompute and materialize given space constraints.

We have implemented the algorithms proposed in this paper in a prototype system called PREFER, which operates on top of a commercial database management system. We present the results of a performance comparison, comparing our algorithms with prior approaches using synthetic datasets. Our results indicate that the proposed algorithms are superior in performance compared to other approaches, both in preprocessing (preparation of materialized views) as well as execution time.

1. INTRODUCTION

Users and applications often need to optimize the selection of entities by ranking them according to the importance (weight) of multiple entity attributes (parameters). Such optimization problems appear often in operations' research and applied mathematics as well as everyday life. However, only few current applications provide multiparametric ranked queries (for example buying a used car in the autotrader section of www.personallogic.com Web site, where one can express desired weights for multiple parameters of a car) and for all such Web applications we are aware of, multiparametric ranked queries are evaluated on small data sets only. Unfortunately, database technology cannot provide acceptable response time and throughput when such queries are evaluated on large data sets. The reason is that the conventional evaluation techniques for such queries require the retrieval and ordering of the entire dataset, with the obvious negative consequences on the time to deliver the first result tuples, which, indeed, are typically the only ones a user is interested in. PREFER is a layer on top of commercial relational databases and allows the efficient evaluation of multiparametric ranked queries.

For example consider a database containing houses available for sale. The properties have attributes such as price, number of bedrooms, age, square feet, etc. For a user, the price of a property and the square feet area may be the most important issues, equally weighted in the final choice of a property, and the property's age may also be an important issue, but of lesser weight. The vast majority of e-commerce systems available for such applications do not help users in answering such queries, as they commonly order according to a single attribute. Manual examination of the query results has to take place subsequently. In our running example, the user will have to order the properties according to, say, price and then manually examine the square feet area and the property's age. One may have to inspect a lot of houses until the best combination of important attributes is found, since the cheap houses will most probably be old and small.

As yet another example, consider a user querying the *Zagat*[1] online database containing restaurant information in New York City. If one is interested in a pricey restaurant and wishes to achieve a balance between Zagat's rating and the distance to the restaurant, one has to "explore" the database issuing selection queries repeatedly. The Zagat database is also available on hand held devices such as the PalmPilot. Issuing multiple queries or browsing through a large result

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACM SIGMOD '2001 Santa Barbara, California USA
Copyright 2001 ACM 1-58113-332-4/01/05 ...$5.00.

[1] `www.zagat.com`

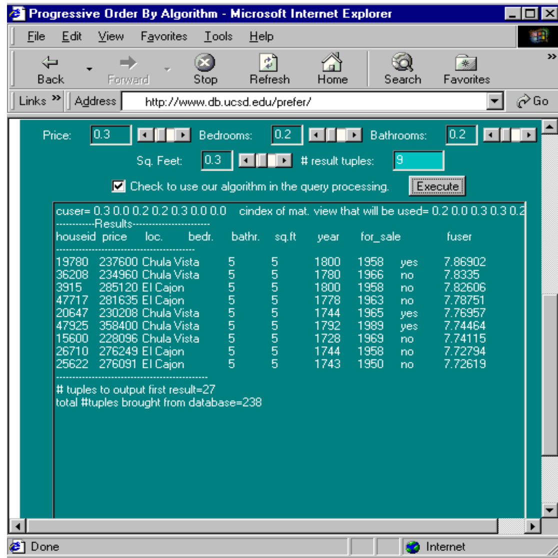


Figure 1: Preference Queries

collection on a hand held device is an even bigger inconvenience especially if connectivity is supported through a wireless link.

In the above examples each user has a *preference* about the importance (or weight) of the attributes associated with the entities (houses and restaurants) searched. In this paper, without loss of generality, we focus on queries over a single relation $R(A_1, A_2, \ldots, A_n)$. The user provides a preference $a_1, a_2, \ldots, a_n$ assigned to each attribute $A_1, A_2, \ldots, A_n$ and a query returns the tuples of R ordered according to the weighted *preference function* $a_1A_1 + a_2A_2 + \ldots + a_nA_n$. We refer to such queries as *preference queries*. The functionality of preference queries is exposed to the user by interfaces such as the one of Figure 1. For each attribute, the interface provides a slider bar that the user adjusts along with the attribute value specified in the selection. The position of the slider bar expresses the attribute preference a_i that the user assigns to the specific attribute A_i. One can also specify the number of tuples desired in the query answer. Once the first set of tuples is returned the user has the ability to receive the next bunch of tuples, again ordered by weighted preference.

Unfortunately database systems do not support the efficient evaluation of preference queries, where "efficiency" primarily refers to response time and throughput. The application has to retrieve the whole database, apply the preference function to each tuple, and sort accordingly. Hence, in the common case that the user wants to retrieve just a few tuples, the application will have to retrieve the potentially thousands or millions tuples of the database; such an approach imposes a prohibitive penalty on the response time and throughput of the overall system as an entire relation has to be ordered to return just a few tuples.

The PREFER system provides excellent response time for such queries, by using pre-materialized preference queries, which we will refer to as views. A preference view is a relational view that is ordered according to a preference function. PREFER works as follows: Given a query and a view it computes the smallest prefix of the view that one has to read in order to find the top tuple according to the query. The intuition is that when the query's preference function is "similar" to the view's preference function the required prefix is small. PREFER's performance scales gracefully as more views are materialized and the chances that every query will find a "similar" view increase. Indeed, PREFER can provide guarantees on the maximum score of the tuples of the view prefix and consequently soft guarantees on the size of the view prefix that has to be accessed, by materializing a sufficient number of views. In this paper, we formalize preference queries and make the following contributions:

- We present an algorithm that computes the Top-N results of a query by using the minimal (given that we only retrieve the first tuple of the view) prefix of a view.
- We specify the set of queries for which a view can provide a soft guarantee about the number of tuples examined in order to provide the top-N tuples of the query.
- The performance of PREFER scales with the number of views that PREFER materializes. We experimentally show that we can provide guaranteed performance to all queries by using a reasonable number of views (between 10-100 in our experiments).
- We present an approximation algorithm that selects the "best views" when there is a limitation on the number of views (and disk space) we can use. We show experimentally that 10-20 views can provide excellent performance guarantees for most of the possible queries.
- We present a detailed experimental evaluation comparing our proposed algorithms with current state of the art and show that our approach provides good scalability both in terms of data set size as well as the number of attributes.
- We have developed our algorithms in a prototype system called PREFER [2] on top of a commercial database management system, demonstrating the practical utility of our overall approach.

This paper is organized as follows: Section 2 reviews related work. Section 3 presents definitions. In Section 4 we present algorithms that derive the answer of a preference query given the result of another preference query that has been already computed and materialized. Section 5 presents algorithms to optimize watermark values (watermark will be defined below) and subsequently provide guarantees on the amount of work needed to answer preference queries. In section 6 we present results from a prototype implementation of the proposed techniques comparing them with state of the art analyzing the performance and the implication of various parameters. Finally section 7 concludes the paper and discusses related problems for further study.

[2]PREFER is available on the web, at www.db.ucsd.edu/PREFER.

2. RELATED WORK

Personalization and customization of software components (e.g., `myexcite.com`) can be thought of as simple expressions of preferences. Agrawal and Wimmers in their pioneering work [2] put the notion on preferences into perspective and introduce a framework for their expression and combination. Their work is fairly conceptually layering the theoretical foundations to a preference framework. Our work, essentially deals with the algorithmic issues associated with the implementation of specific features of this framework. We adopt terminology in alignment with the framework of Agrawal and Wimmers [2]

Combining and ranking different models was used in the context of multi-media systems by Fagin [7, 9, 8]. Our work is related to that of Fagin since we are also concerned with the efficient computation of the extreme values of functions. Our optimization objectives and techniques are fundamentally different from that of Fagin however.

A significant number of works has been published the last five years on answering queries using views. The earlier works focused on conjunctive queries and views (e.g., [1]) and subsequent works extended into more powerful queries, views, and view set descriptions [6, 15, 13]. Rewriting aggregate queries using views has also been addressed [5, 4]. The nature of those algorithms is logic-based rather than quantitative, as is the case with our algorithms for using a view to answer a query, since the nature of the queries is very different.

The work closest to the one presented herein, is the work by Chang et. al., [3]. In this work an indexing technique, called the *Onion Technique* was introduced to facilitate the answer of linear optimization queries. Such queries are similar to preference selection queries since they retrieve tuples maximizing a linear function defined over the attributes of the tuples of a relation R. The basic observation of this technique is that the points of interest lie in the convex hull of the tuple space. Thus, the Onion technique in a preprocessing step computes the convex hull of the tuple space, storing all points of the first hull in a file and proceeds iteratively computing the convex hulls of the remaining points; it stops when all points in the tuple space have been placed to one of the convex hull files. Query processing is performed by evaluating the query and scanning each of these files, starting from the one storing the exterior convex full (since it is guaranteed to contain the first result), stopping when all desired results have been produced.

The onion technique suffers from two major drawbacks. Computing convex hulls is a computationally intensive task with complexity $O(n^{\frac{d}{2}})$, where n is the number of tuples in R and d is the number of attributes, making the technique impractical for large relations with more than two attributes. Moreover the technique is very sensitive in performance to the granularity of the attribute domains. If an attribute has very small domain, it is likely that all tuples lie in the same convex hull, thus a linear scan of the entire data set is required to produce the results. The performance of the technique is highly dependable on the characteristics of the dataset and no guarantees in performance can be provided. We evaluate the performance of this technique in Section 6

3. NOTATION AND DEFINITIONS

This section defines queries, views, and other relevant notation in the context of PREFER. Let R be a relation with k attributes $(A_1, \ldots, A_k)$ and let $[m_i, M_i]$ be the domain of attribute A_i, $1 \leq i \leq k, m_i, M_i \in \mathcal{R}^+$. The notation $A_i(t)$ refers to the value of attribute A_i in the tuple t.

Every query q consists of a *preference function* $f_q(.)$ and a single relation R. The preference function $f_q(t)$, $\prod_{i=1}^{k}[m_i, M_i] \rightarrow \mathcal{R}^+$ defines a numeric *score* for each tuple $t \in R$. The output of the query q is the *query result sequence* $R_q = [t_q^1, t_q^2, \ldots, t_q^n]$ of the tuples of R such that $f_q(t_q^1) \geq f_q(t_q^2) \geq \ldots \geq f_q(t_q^n)$. Note that we use the notation t_q^i to denote the tuple in the i-th position in the result sequence of q. Views are identical to queries; we use the term *view* when we refer to a query whose result has been materialized in advance in the system and we use the term *ranked query* (or query) when we refer to a query that the user submitted and the system has to reply to.

In this paper we focus on queries (and views) that use linear preference functions of the form $f(t) = \sum_{j=1}^{k} v_j A_j(t)$, because they provide an excellent tradeoff between ability to specify the order using multiple parameters and, at the same time, can be very efficiently pipelined using the techniques we present in this paper. The vector $\vec{v} = (v_1, \ldots, v_k)$ is called the *preference vector* of the query (view) and each coordinate of the vector is called *attribute preference.* We use $f_v(.)$ to indicate that f_v is a preference function with preference vector $\vec{v}$. Moreover we denote as R_v a ranked view which is ranked according to f_v. Without loss of generality, we assume that attribute preferences are normalized in $[0, 1]$ and that $\sum_{j=1}^{k} v_j = 1$. This assumption is not restrictive, as whatever the range of attribute preferences would be, they can always be normalized instantly by the system. Moreover, we choose to adopt such a normalization since we believe it is in agreement with the notion of preference. The total preference of a user is 1 and the preference on individual attributes is expressed as an additive term towards the total preference.

4. HOW TO PIPELINE A RANKED QUERY USING A RANKED VIEW

The algorithm presented next uses a view sequence R_v, which ranks the tuples of a relation R according to a preference vector $\vec{v}$, in order to efficiently pipeline the output sequence R_q of a user query q, which ranks the tuples of the relation R according to the user's preference vector $\vec{q}$. The key to the algorithm is the computation of a prefix R_v^1 of R_v that is sufficient to assure that the first tuple t_q^1 of the sequence R_q is in R_v^1. Once the first tuple of R_q has been retrieved the algorithm proceeds to compute the prefix R_v^2, to deliver the second tuple of R_q, and so on, leading to an efficient pipelined production of the query result.

The algorithm is presented in three steps. First we define the *first watermark* point, whose definition involves only $f_q(.)$, $f_v(.)$ and t_v^1 and provides a bound on the view preference score $f_v(t_q^1)$ of the top result t_q^1 of the query.[3] Then Section 4.1 provides the algorithm that pipelines the query output, given an "oracle" that provides watermark points. The algorithm is applicable to any function for which one can

[3]The first watermark provides the *tightest* prefix of R_v given knowledge of t_v^1 only. One can produce tighter prefixes by using more tuples from R_v but this comes at the cost of increased watermark point computation and retrieval of more tuples of R_v.

construct such an "oracle". Section 4.2 provides the computation of the watermark in the case of queries and views specified by linear functions. Finally Section 4.3 presents an example of using this algorithm.

DEFINITION 1 (FIRST WATERMARK). *Consider*

- *the view v consisting of the function f_v applied on the relation R, and*
- *the query q consisting of the function f_q applied also on the relation R*

The first watermark of the user query q in the view R_v is the maximum value $T^1_{v,q} \in \mathcal{R}^+$ with the property:

$$\forall t \in R, f_v(t) < T^1_{v,q} \Rightarrow f_q(t) < f_q(t^1_v) \tag{1}$$

The definition leads to an efficient computation of the watermark (see Section 4.2) since it involves only tuple t^1_v. According to the definition, if a tuple t in the view R_v is below the first watermark $T^1_{v,q}$ (that is, $f_v(t) < T^1_{v,q}$) then t cannot be the top result t^1_q of the query, since at least t^1_v is higher in the query result (according to the property $f_q(t) < f_q(t^1_v)$). This implies that $f_v(t^1_q) \geq T^1_{v,q}$. Hence, in order to find t^1_q one has to scan R_v from the start and retrieve the prefix $[t^1_v, t^2_v, \ldots, t^{w-1}_v, t^w_v)$, where t^w_v is the first tuple in R_v with $f_v(t^w_v) < T^1_{v,q}$, i.e., t^{w-1}_v is the last tuple of R_v that is above the watermark. The top query tuple t^1_q is the tuple $t^j_v, 1 \leq j \leq w-1$ that maximizes $f_q(t^j_v)$. Furthermore, the prefix $[t^1_v, t^2_v, \ldots, t^{w-1}_v]$ allows us to potentially locate a few more (besides t^1_q) of the top tuples of the query result, as the following theorem shows:

THEOREM 1. *Let $[t^1_q, t^2_q, \ldots, t^{w-1}_q]$ be the ranked order, according to q, of the tuples $[t^1_v, t^2_v, \ldots, t^{w-1}_v]$ that are above the first watermark. Let s be the index of t^1_v in this order, i.e., $t^1_v \equiv t^s_q$. Then $t^1_q, \ldots, t^s_q$ are the tuples with the highest rank in the answer of q.*

Proof: Clearly $f_q(t^1_q) \geq \ldots \geq f_q(t^{w-1}_q)$. Moreover due to the watermark property (Equation 1) $\forall t, f_v(t) < T^1_{v,q} \Rightarrow f_q(t) \leq f_q(t^s_q)$. The theorem follows, since $f_q(t^s_q) \leq f_q(t^{s-1}_q) \leq \ldots \leq f_q(t^1_q)$. □

The theorem guarantees that the top-s tuples, according to $f_q(.)$, in the prefix $[t^1_v, t^2_v, \ldots, t^{w-1}_v]$ are also the top-s tuples in the answer of q. That is, it is impossible for a tuple below the watermark to be one of the top-s tuples.

4.1 The Core of the Pipelining Algorithm

The algorithm *PipelineResults* in Figure 2 inputs R_v and computes in a pipelined fashion the N tuples with the highest score according to q. The algorithm assumes the existence of a function *DetermineWatermark()* (see Section 4.2) to efficiently compute the watermark value in R_v. Let s be the number of tuples output after computing the first watermark. If $s \geq N$ then our objective has been achieved. Otherwise we output the sequence of the top-s tuples and we mark those tuples as *processed* in R_v. Then we repeat the process and determine a new watermark value, to derive a new sequence of tuples with the highest scores according to q from the *unprocessed* tuples in R_v. In each iteration we locate the first tuple in R_v which is not marked as processed. Let this tuple be t^{top}_v. This is the tuple with the top score according to v among the unprocessed tuples of R_v. We repeat the watermarking process using t^{top}_v. A new sequence of tuples having the highest score according to q among the remaining tuples will be determined and output.

```
Algorithm PipelineResults(R_v,q,v, N){
Let top = 1
while (less than N tuples in the output) {
 Let T^top_{v,q} = DetermineWatermark(t^top_v)
 Scan R_v and determine the first tuple t_w
    with f_v(t_w) < T^top_{v,q}
 For all tuples t ∈ [t^1_v, t^{w-1}_v] compute and sort by f_q(t)
 Let s be the index of t^top_v in the sorted order
 Output the tuples t^1_q ... t^s_q and mark them in R_v
    as processed
 Find the top unprocessed tuple t^i_v in R_v
 Let top = i
 }
}
```

Figure 2: Algorithm to output the first N tuples according to q

4.2 Determining the Watermark

We will now use Equation 1 to determine the watermark value $T^1_{v,q}$ in the case of linear functions f_q and f_v. We assume that view R_v is ordered by decreasing values of the score of f_v. Thus we will determine the tuple t' that maximizes $f_v(t')$ while satisfying $f_q(t') < f_q(t^1_v)$. Since we know the values of t^1_v, $\vec{q} \equiv (q_1, \ldots, q_k)$ and $\vec{v} \equiv (v_1, \ldots, v_k)$, we need to come up with bounds for the values of $t \equiv (A_1(t), \ldots A_k(t))$ using the known parameters to maximize $f_v(t')$ while satisfying the inequality of Equation 1 for all $t \in R$. We will subsequently use these bounds to derive the watermark. Let us express $f_q(t) = \sum_{i=1}^k q_i A_i(t)$ as a function of $f_v(t) = \sum_{i=1}^k v_i A_i(t)$. Thus,

$$f_q(t) = \sum_{i=1}^k q_i A_i(t) = f_v(t) + \sum_{i=1}^k (q_i - v_i) A_i(t) \tag{2}$$

By substituting Equation 2 into Equation 1 we get

$$\forall t \in R, f_v(t) \leq T^1_{v,q} \Rightarrow f_v(t) + \sum_{i=1}^k (q_i - v_i) A_i(t) \leq f_q(t^1_v) \tag{3}$$

Consider that the highest possible $f_v(t)$ is achieved for t'. It is:

$$f_v(t') + \sum_{i=1}^k (q_i - v_i) A_i(t') \leq f_q(t^1_v) \tag{4}$$

We will treat Equation 4 as equality; since the left side of Equation 4 is linear on $f_v(t')$, the corresponding inequality is trivially satisfied. Since out objective is to determine the maximum $f_v(t')$ value that satisfies Equation 4, which is linear in $f_v(t')$, we will determine bounds for each attribute $A_i(t')$ in a way that the left part of Equation 4 is maximized. We determine the bounds for each attribute $A_i(t')$, by the following case analysis. Recall also that each attribute A_i has domain $[m_i, M_i]$.

$$A_i(t') = \begin{cases} min(\frac{f_v(t') - \sum_{j<>i}^{k} v_j m_j}{v_i}, M_i) & q_i > v_i <> 0 \\ M_i & q_i > v_i = 0 \\ 0 & q_i = v_i \\ max(\frac{f_v(t') - \sum_{j<>i}^{k} v_j M_j}{v_i}, m_i) & q_i < v_i \end{cases} \quad (7)$$

Figure 3: Bounds for A_i

- $(q_i - v_i) > 0$ and $v_i <> 0$: In this case we have

$$A_i(t') = \frac{f_v(t') - \sum_{j<>i}^{k} v_j A_j(t')}{v_i} \leq \frac{f_v(t') - \sum_{j<>i}^{k} v_j m_j}{v_i} \quad (5)$$

  We set $U_i = \frac{f_v(t') - \sum_{j<>i}^{k} v_j m_j}{v_i}$. Since $A_i(t') \leq M_i$, we have that $A_i(t') = min(U_i, M_i)$.

- $(q_i - v_i) > 0$ and $v_i = 0$: then $A_i(t') = M_i$
- $(q_i - v_i) = 0$: we can ignore this term
- $(q_i - v_i) < 0$ and $v_i <> 0$: In this case we have that:

$$A_i(t') = \frac{f_v(t') - \sum_{j<>i}^{k} v_i A_j(t')}{v_i} \geq \frac{f_v(t') - \sum_{j<>i}^{k} v_j M_j}{v_i} \quad (6)$$

  We set $L_i = \frac{f_v(t') - \sum_{j<>i}^{k} v_j M_j}{v_i}$. Since $A_i(t') \geq m_i$, we have that $A_i(t') = max(L_i, m_i)$.

Figure 3 summarizes the results of our analysis for each attribute value $A_i(t')$. Notice that we use the notation $A_i(t')$ to denote the bound for the value of attribute A_i. Also notice that when $(q_i - v_i) > 0$ we determine an upper bound for the value of $A_i(t')$ whereas when $(q_i - v_i) < 0$ we determine a lower bound. The main difficulty in solving Equation 4 directly, lies on the existence of min and max terms, with two operands each, in the expressions derived for the attribute bounds (Figure 3). Each min (equivalently max) term however, is linear on $f_v(t')$ thus it is easy to determine for which range of $f_v(t')$ values, each operand of min (equivalently max) applies, by determining the $f_v(t')$ value that makes both operands equal. Assume the expression for attribute bound $A_i(t')$ contains a min or a max term. Let e_i be the value for $f_v(t')$ that makes both operands of min or max equal. As $f_v(t')$ varies, we now know exactly which operand in each min or max term we should use to determine a bound on the attribute value. Since both U_i and L_i terms are linear on $f_v(t')$, we observe weather $f_v(t')$ lies on the left or right or e_i. There are at most k attribute bound expressions and thus $1 \leq i \leq k$. Possible values of $f_v(t')$ range between $\sum_{i=1}^{k} v_i m_i$ and $\sum_{i=1}^{k} v_i M_i$. If we order the e_i's, we essentially derive a partitioning of the range of possible values of $f_v(t')$ in $k+1$ intervals, $I_i, 1 \leq i \leq k+1$. For each value of $f_v(t')$ in these intervals the expressions used to compute each attribute bound are fixed and do not involve min or max.

We construct a table E having $k+1$ columns, denoting the value intervals for $f_v(t')$ and k rows, denoting the expressions for each attribute bound. For each entry $E(i,j), 1 \leq i \leq k, 1 \leq j \leq k+1$ in this table we record the exact expression that we will use to determine the bound for attribute A_i. If an attribute bound expression is not a function of $f_v(t')$ we can just record the value in the suitable entry as a constant. Once the table is populated, for each value of $f_v(t')$ we know the attribute bound formulas that comprise the left hand side of Equation 4. Thus we have $k+1$ possible expressions for the left side of Equation 4. Each expression, $E_j, 1 \leq j \leq k+1$ is produced by:

$$E_j = f_v(t') + \sum_{i=1}^{k} (q_i - v_i) E(i,j) \quad (8)$$

THEOREM 2. *Setting $E_j = f_q(t_v^1), 1 \leq j \leq k+1$ and solving for $f_v(t')$ determines the watermark value.*

Proof: For each j two possibilities exist: (a) the $f_v(t')$ value computed does not fall in the j-th interval. In this case, the expression for E_j cannot yield $f_q(t_v^1)$ since E_j produces an upper bound for $f_q(t)$ by construction, (b) $f_v(t')$ falls in the j-th range. Since $E_j = f_q(t_v^1)$ is a linear function and has a unique solution in range j, $f_v(t')$ is the watermark $T_{v,q}^1$. Note that the range of possible values for $f_v(t')$ is the same with the range of possible values for E_j, thus j will always be identified. □

Algorithm *DetermineWatermark* is shown in Figure 4. The algorithm assumes that table E has been computed in a preprocessing step. The algorithm uses $O(k^2)$ space and determines the watermark solving k equations in the worst case.

```
Algorithm DetermineWatermark(tuple t_v^1) {
    for j from k+1 downto 1 {
        Solve E_j = f_q(t_v^1) and determine watermark
        if watermark ∈ I_j return watermark
    }
}
```

Figure 4: Algorithm *DetermineWatermark*

4.3 An Example

Let us present an example of the algorithm's operation. Assume q is a query with $\vec{q} = (0.1, 0.6, 0.3)$ and R_v a view with $\vec{v} = (0.2, 0.4, 0.4)$. Let $m_1 = m_2 = m_3 = 5$ and $M_1 = M_2 = M_3 = 20$. The sequence R_v is shown in Figure 5. To populate table E we use the equations of Figure 3 to calculate the bounds for each attribute A_i. Thus: $A_1(t) = max(\frac{f_v(t')-16}{0.2}, 5)$, $A_2(t) = min(\frac{f_v(t')-3}{0.4}, 20)$, $A_3(t) = max(\frac{f_v(t')-12}{0.4}, 5)$.

Next we calculate $e_i's$ that make the terms in min or max expressions equal.

$$e_1 = 17, \; e_2 = 11, \; e_3 = 14 \quad (9)$$

We are now ready to fill table E. The table is presented in Figure 6. Recall that t_v^1 is the first tuple of R_v. Now we solve Equation 4 for each of the 4 intervals starting with the last one. In interval I_4, solving Equation 4 results in $f_v(t') = 8.8$ which is not in I_4 and it is rejected. In I_3 we

tupleID	A1	A2	A3	$f_v(t)$	$f_q(t)$
1	10	17	20	16.8	17.2
2	20	20	11	16.4	17.3
3	17	18	12	15.4	16.1
4	15	10	8	10.2	9.9
5	5	10	12	9.8	10.1
6	15	10	5	9	9
7	12	5	5	6.4	5.7

Figure 5: View R_v and scores of each tuple based on f_v and f_q

$T^1_{v,q}$	5..11	11..14	14..17	17..20
A_1	5	5	5	$\frac{f_v(t')-16}{0.2}$
A_2	$\frac{f_v(t')-3}{0.4}$	20	20	20
A_3	5	5	$\frac{f_v(t')-12}{0.4}$	$\frac{f_v(t')-12}{0.4}$

Figure 6: Table E

get $f_v(t') = 14.26$, which is valid. To output the first tuple for f_q we scan R_v up to the first tuple with score greater than or equal to $f_v(t') = 14.26$. This is tuple t_v^3 with score 15.4. So the minimum prefix of R_v that we have to consider in order to get the first result for query q consists of all tuples $t \in [t_v^1, t_v^3]$. We order these three tuples by f_q and output t_v^2 and t_v^1. Now in order to get further results we locate the first unprocessed(not yet output) tuple in R_v, which is t_v^3 and use it instead of t_v^1 in Equation 4. The algorithm continues like this. If we repeat the above steps, we get the following results. $f_v(t') = 13.1$, so the prefix now becomes just t_v^3, which we output. Next we use t_v^4 in Equation 4 and get $f_v(t') = 8.26$, so the prefix is $[t_v^4, t_v^6]$. We sort these tuples and output t_v^5 and t_v^4. Next we use t_v^6 in Equation 4 and get $f_v(t') = 7.66$, so our fourth prefix is just t_v^6, which we output. Finally output t_v^7, which is the last unprocessed tuple in R_v.

5. VIEW SELECTION

PREFER materializes in advance multiple views in order to provide short response time to client queries. In its simplest version the view selection module (see Figure 8) inputs from the user the relation R and the size l of the maximum view prefix that the PipelineResults Algorithm may have to retrieve in order to deliver the first result of an arbitrary preference query on R. The view selection module outputs and materializes a set of view sequences $\mathcal{V}$ such that for every query q there is at least one identifiable view $R_v \in \mathcal{V}$ that "covers" q, i.e., when R_v is used to answer q at most l tuples of R_v are needed to deliver the first tuple of q. In Section 6 we show experimentally that the number of views needed to cover the whole space of possible queries is in the order of 10 to 100 in typical cases. However, if space limitations require that we build at most n views, a modified view selection algorithm is used in order to cover the maximum amount of queries with n views; since the problem of finding such a maximum coverage, as we will show, is NP-hard, PREFER uses a greedy algorithm that provides an approximate solution. The details and the properties of the view selection algorithm are described in Section 5.2. Note that, in a similar fashion, PREFER can select views that guarantee the retrieval of the first m query results by retrieving at most l tuples. We describe the generalization to top-m tuples in Section *5.1.1*.

We present next the key definitions of "coverage" of a query by a view. Section 5.1 provides algorithms that decide coverage and compute (precisely and approximately) the space covered by a view. Section 5.2 uses the coverage algorithms in a view selection algorithm that either (i) produces a set of views that covers the space of all possible queries (referred to as *query space*), or (ii) produces the best approximate set of n views that cover as much query space as possible.

DEFINITION 2. *The ranked materialized view R_v covers the query q for its top m results using l tuples, if the PipelineResults Algorithm generates the top-m result tuples of q by using at most the top-l tuples of R_v. We will say that q is covered by R_v using l tuples to indicate that the first result tuple of q requires at most l tuples of R_v to be retrieved.*

We will often also say R_v covers q when the number l of tuples needed is obvious from the context.

DEFINITION 3. *The space $S^l_{R_v} \subseteq [0,1]^k$ covered by the view sequence R_v using l tuples is the set of all query preference vectors $\vec{q}$ such that the first result of q can be derived using only the top-l tuples of R_v.*

5.1 Deciding Coverage and Computing the Space Covered by A View

We describe next two key algorithms of the view selection module:

1. the *view cover decision* algorithm is given a sequence R_v, a number l, and a query q and decides in $O(1)$ time whether q is covered by R_v using l tuples.[4] Notice that the algorithm uses only the l-th tuple of R_v.

2. the *view cover* algorithm inputs a view sequence R_v and a number l and returns the k-dimensional space $S^l_{R_v}$.

For both algorithms the key point is the following: Since we want to guarantee that at most l tuples from R_v will be read whenever a query q uses R_v we have to place the first watermark at t_v^l. By the watermark properties and a mathematical manipulation similar to the one of page 4.2 we derive the inequality

$$f_v(t_v^l) + \sum_{i=1}^{k} (q_i - v_i) A_i(t_v^l) \leq f_q(t_v^1) \qquad (10)$$

In Equation 10 the only unknowns are the components of the vector $(q_1, \ldots, q_k)$, for which $\sum_{i=1}^{k} q_i = 1$. Hence the view cover decision algorithm requires that we simply plug the vector $(q_1, \ldots, q_k)$ in Equation 10. The view cover problem requires solving Equation 10, which is a linear function. Its solution $S^l_{R_v}$ is in general a convex polytope [12]. In general computing the exact solution of Equation 10 is not an easy computational task. We reduce the computational costs involved by computing an optimistic approximation of the solution however. More specifically, we compute the minimum

[4]Obviously the PipelineResults Algorithm could be used as the view cover decision algorithm but its complexity is $O(l)$.

and maximum values of each q_i that tightly bound the solution polytope, deriving an axis-aligned *Minimum Bounding Hyperrectangle* (MBH). Determining the $MBH^l_{R_v}$ of the solution of Equation 10 can be performed very efficiently; it consists of determining the solutions to the following k constraint optimization problems:

$$min\ q_i\ s.t \begin{cases} f_v(t^l_v) + \sum_{i=1}^k (q_i - v_i)A_i(t) \leq f_q(t^1_v) \\ \sum_{i=1}^k q_i = 1 \\ q_i \geq 0 \end{cases} \quad (11)$$

$$max\ q_i\ s.t \begin{cases} f_v(t^l_v) + \sum_{i=1}^k (q_i - v_i)A_i(t) \leq f_q(t^1_v) \\ \sum_{i=1}^k q_i = 1 \\ q_i \geq 0 \end{cases} \quad (12)$$

Each constraint optimization problem is linear and can be solved in polynomial time using standard off the shelf optimization methods such as Simplex. Simplex is a widely used method, requiring $O(k^2)$ space to derive a solution. Even for very large k it usually reaches the solution in a few iterations.

5.1.1 Guarantees For Multiple Results

Providing guarantees for multiple results from R_v can take place in a similar fashion. One can repeat the above process for the second desired watermark position. If the corresponding convex polytopes intersect, all the queries falling inside the intersection, satisfy both guarantees. Let $\ell_i, 1 \leq i \leq N$ be the positions of watermark $T^i_{v,q}$ we wish to guarantee. Repeating the procedure above for each ℓ_i, will provide a sequence of minimum bounding hyperrectangles MBH_i. If $\bigcap_{i=1}^N MBH_i$ is not null, then every query falling in, satisfies all guarantees. If $\bigcap_{i=1}^N MBH_i$ is null, then we are certain that such a guarantee cannot be provided by R_v for any query. Since MBH is an optimistic approximation of the solution convex polytope, if a pair of MBH's does not intersect, then the corresponding convex solutions don't intersect either. However, it is possible to have an non null intersection of all the MBHs but a null intersection of the corresponding convex solutions. This introduces an error which we evaluate in Section 6.

5.2 Selecting Views To Materialize

The simplest version of the view selection algorithm covers every possible query with at least one view R_v. That is, the view selection algorithm generates a set of views $\mathcal{V}$ such that the union of the query spaces covered by the views covers the whole space $[0,1]^k$, i.e., $\cup_{R_v \in \mathcal{V}} S^l_{R_v} = [0,1]^k$. In practice, the algorithm considers a discretization of the $[0,1]^k$ space by using a user-provided discretization parameter d. Then the space has $\{\#(x_1,\ldots,x_k)|x_i = r_i d, r_i \in Z, x_i \in [0,1], \sum_{i=1}^k x_i = 1\}$ points and the view selection algorithm keeps introducing views until no point is left uncovered. The $O(1)$ view cover decision algorithm is used to check whether a given view R_v covers a query q.

In reality space constraints may exist and only a finite number of views, C can be actually materialized. Thus, the choice of a "good" set of ranked views to materialize is an important issue. This gives rise to the following constraint optimization problem.

PROBLEM 1. **(View Selection Under Space Constraint)** *Given a set of views $R^1_v, \ldots, R^s_v$ that covers the space $[0,1]^k$ select C views that maximize the number of points in $[0,1]^k$ covered.*

```
Algorithm ViewSelection(){
while (not all preference vectors in [0,1]^k covered)
{  Randomly pick v ∈ [0,1]^k and add it to the list
   of views, L
}
GREEDY ← 0
for l = 1 to C {
 select v ∈ L that covers the maximum uncovered
   vectors in [0,1]^k
 GREEDY ← GREEDY ⋃ MBH_l
 }
}
```

Figure 7: Ranked View Selection Under Space Constraint

Problem 1 is an instance of the *maximum coverage problem* [11], as the following reduction shows: The space of all possible preference vectors, $[0,1]^k$, can be considered as the reference set. Each of the views is a "subset" of $[0,1]^k$ containing a number of preference vectors. We wish to select C "subsets" to maximize the number of elements of the reference set that are covered. The maximum coverage problem is NP-Hard as set cover can be easily reduced to it. However, it can be approximated efficiently as the following theorem shows:

THEOREM 3 (GREEDY APPROXIMATION). *The Greedy Heuristic is an $1 - \frac{1}{e}$ approximation for maximum coverage.*

Proof: See [11].

The Greedy heuristic works iteratively by picking the next view from the collection $R^1_v, \ldots, R^s_v$ that covers the maximum number of uncovered elements of $[0,1]^k$. Figure 7 summarizes our approach.

5.3 Selecting A Ranked View for a Preference Query

Query processing, once C views have been materialized proceeds as follows. The MBH's of the views are stored in a data structure supporting "point in hyperrectangle" queries, such as an R-tree [10, 14]. At query time, we use the data structure to identify the MBH and subsequently the ranked views whose MBH's contain the point. An extra check is performed (using the view cover decision algorithm) to find the ranked views that actually cover the given query. The reason for the extra check is that the $MBH^l_{R_v}$ is only an approximation of the convex polytope $S^l_{R_v}$ and it is possible that $q \in MBH^l_{R_v}$ but $q \notin S^l_{R_v}$. This is a side effect of the approximation of the exact solution. On the average, the ratio of the polytope's volume to the MBH's volume is close to the ratio of the volume of a k-dimensional sphere of diameter d over a k-dimensional cube of side d. We evaluate the effectiveness of the approximation in Section 6 and we find that on the average 3% of the queries (for the range of parameters of our experimentation) fall in the MBH but not in the convex polytope.

When the overall number of views that we materialized is bounded, it is likely that not all points of $[0,1]^k$ are covered. Thus it is possible to generate preference vectors that are

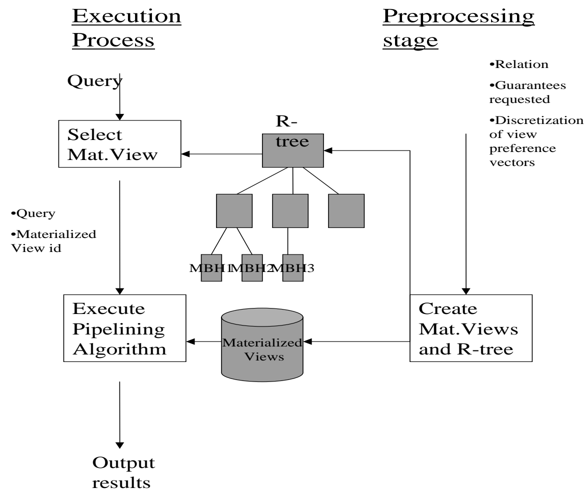


Figure 8: System Architecture

not covered by any of the stored views. For such queries, we cannot provide performance guarantees based on our construction. We execute them by choosing the ranked view with an MBH nearest to the preference vector, as a heuristic.

5.4 PREFER System Architecture

The overall system architecture is shown in Figure 8. Using algorithm $ViewSelection$ we select a number of views and we materialize them. A relational DBMS is used for storing the views. The user interacts with PREFER through an applet which connects to the database through JDBC. The MBH's of the views are stored in an R-tree and given a preference vector, we identify using the R-tree the MBH that encloses it. The MBH points to a materialized view, which we subsequently use to apply our algorithms to identify and retrieve the results and ship it to the user.

A preference query can be trivially answered using a DBMS, by evaluating the preference function on each database tuple and sorting the tuples by their score. We allow this option in PREFER and one can observe in real time the performance benefits of our approach.

6. EXPERIMENTAL RESULTS

To evaluate PREFER's algorithms for the efficient execution of preference selection queries, we carried a detailed performance evaluation. First we measured the running time of our algorithms during their preprocessing step, where the materialized view selection is performed. Then we evaluate query performance as different parameters vary. We define *query performance* as the fraction of queries that satisfy the user-provided guarantee on the size of the view prefix that PREFER has to retrieve from the view in order to retrieve a user-provided number of top query results. We present a comparison of our algorithms with other proposed state-of-the-art solutions and finally compare with the time required by a commercial database management system to complete the same task.

The experiments use two synthetic datasets; the relation attributes of the first dataset are independent while the attributes in the second dataset are correlated. The database consists of a relation *houses* with six attributes, namely:

attributes	*Top-1 tuple*	
	Discretization 0.1	*Discretization 0.05*
3	25min	88min
4	60min	370min
5	190min	1800min

attributes	*Top-10 tuples*	
	Discretization 0.1	*Discretization 0.05*
3	30min	93min
4	65min	380min
5	210min	2000min

Figure 9: View Selection Algorithm Running Time

HOUSEID, PRICE, BEDROOMS, BATHROOMS, SQ_FT and YEAR. We performed experiments that used three, four or five of the attributes (HOUSEID is not a preference attribute). The cardinality of the five preference attributes is 1000000 , 10 , 8 , 3500 and 50 respectively for the random dataset and 1 , 500000 , 5 , 5 , 1500 and 50 respectively for the correlated dataset. PRICE, BEDROOMS and SQ.FT were used for experiments involving three attributes; BATHROOMS was added as the fourth attribute and YEAR as the fifth. For the random dataset, the attribute values are chosen with a uniform distribution over their domain. In the correlated dataset, we used correlation patterns that we discovered in real datasets containing house information (we did not use these datasets because they were relatively small in size). The correlation coefficient between BEDROOMS and the rest of the attributes, is between 0.35 and 0.73, and the correlation of the other attribute pairs is at similar levels.

We use a discretization of 0.1 for the domain from which we draw view and query preference vectors (0 through 1, in increments of 0.1), except for when the experiment involves only three attributes in which case we use a granularity of 0.05 in order to have a sizeable number of possible preference vectors and stress the view selection algorithm. The computing environment consisted of a dual Pentium II with 512MB RAM running Windows NT Workstation 4.0, where all experiments were executed, and a PII 256MB RAM Windows NT Server 4.0, where the datasets were stored in an Oracle DBMS. PREFER is implemented in Java. The two computers were connected through a LAN.

The preprocessing phase of our algorithms, essentially the solution to the optimization problems of Equation 12, was carried out using the simplex method. We used a widely available implementation of the Simplex method as a black box. Such methods are well studied in the literature and highly optimized for performance.

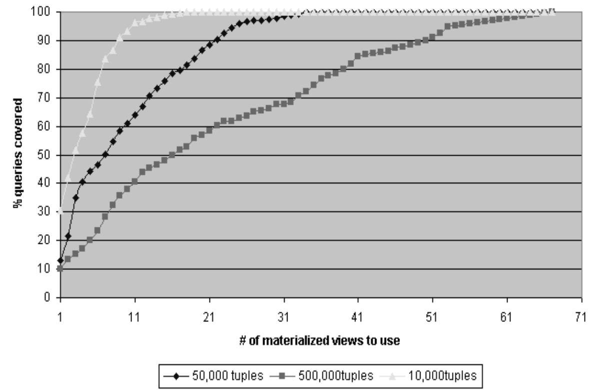
View Selection Running Time. Our first experiment assesses the running time that the view selection algorithm takes to cover the space of all queries. Figure 9 presents the running time of the algorithm for various parameters of interest, namely number of attributes in the underlying dataset, discretization of the domain of preference vectors and number of result tuples (1 or 10) that we require guarantees for, on a 50K tuple database. The guarantee provided in this case is 500 tuples (size of view prefix). The times in the figure include the time to find the 500-th ranked tuple

(located with a single pass over the underlying dataset) of as many views as were required, plus the time to solve the view cover decision problem, as described earlier. The running time increases with the number of attributes in the dataset, as the preference vector space increases in size; more effort is required to cover the entire space. It also increases with the granularity of the preference vectors as the space becomes denser in candidate query points that the algorithm has to cover. Finally, the running time increases with the number of result tuples we wish to provide guarantees for, as the algorithm has to solve the view cover decision problem for each result tuple we wish to have a guarantee for.

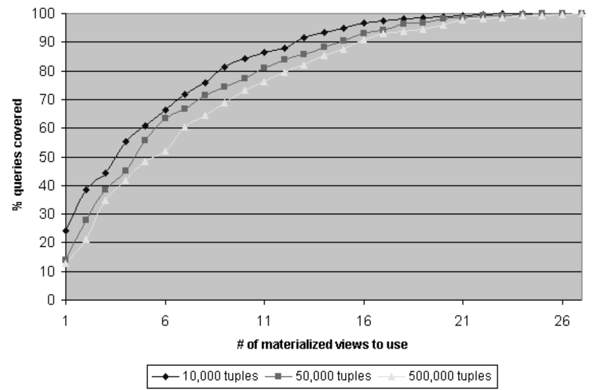
Query Performance as function of the Dataset size. Figure 10 presents the results of an experiment assessing the query performance of PREFER with respect to the dataset size. In this experiment we used datasets with four attributes. We target a guarantee that the first result of a random query is identified by retrieving at most 500 tuples from the database. We vary the number of views allowed to be materialized and we measure the fraction of the queries that satisfy the guarantee we wish to provide. The fraction of the queries is measured by exhaustively executing all possible queries (whose vectors' components fall on the 0.1 discretization) on the views that have been materialized and counting the number of them that satisfy the guarantee. We observe that the view selection algorithm scales gracefully with the dataset size. For the case of correlated data (Figure 10(a)) increasing the number of tuples in the database by five times, requires only doubling the number of materialized views to cover 100% of the possible queries. Increasing the number of tuples fifty times, requires tripling the number of materialized views to cover 100% of the queries. Notice how only ten views are enough to cover 90% of the query space for a dataset with 10,000 correlated tuples (Figure 10(a)). Since the distribution of tuple values is skewed, the distribution of scores in each view is expected to be skewed as well. It appears that for this dataset, as the number of tuples increases, the sizes of the generated covered spaces are smaller, since the number of tuples greater than a specific watermark value decreases, due to skew. Consequently, for a fixed number of views, we expect a smaller fraction of the preference attribute space to be covered. This explains the smaller slopes of the curves for increasing number of tuples.

Figure 10(b) presents the results of the same experiment on the random dataset. In the case of random data (uniformly distributed attribute values) the number of additional views required to assure that all queries provide guarantees appears to grow very slowly with database size. Since we are dealing with uniform data the score values are expected to be evenly distributed. The fraction of tuples greater than a specific watermark value essentially remains constant (for a truly uniform distribution). The MBH sizes are varying in size in our case, as we just deal with a sample of a uniform distribution having some random variation, but are not drastically different. As a result, the difference in the fraction of space covered with the same number of MBHs does not vary a lot as the number of tuples increases.

In Figure 10 for a fixed number of views there are two reasons for missing the guarantee. The first reason is the space that remains uncovered as a consequence of the imposed constraint on the number of views(essentially storage space). The second is the approximation of the convex polytope solutions with MBHs during the phase that we locate which view is relevant for an incoming query. In both cases, the error due to the approximation with MBHs is less than 3% on the average, signifying that our use of approximations of the solutions is not an important source of error.



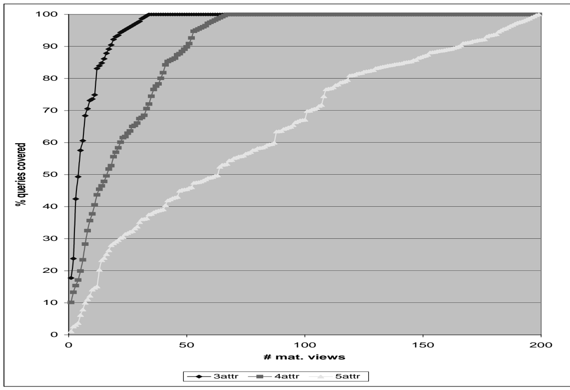
(a) Correlated dataset



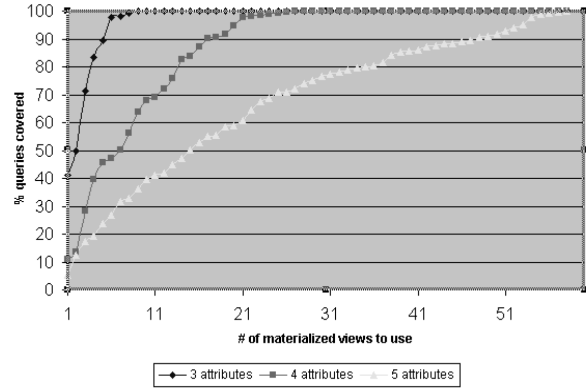
(b) Random dataset

Figure 10: Varying the dataset size

Varying the number of attributes. Figure 11 presents the results of an experiment assessing the scalability of the view selection algorithm with respect to the number of attributes in the underlying dataset, which has 500,000 tuples. Figure 11(a) presents the results of the experiment for the correlated dataset. The number of tuples in the datasets is the same, so as the number of attributes increases the distribution of distances between the tuples is expected to increase as well. We expect that the distribution of score values in each view becomes increasingly more skewed as the dimensionality increases, for the types of preference functions we consider in this paper. The number of tuples with scores larger than a specific watermark value decreases for this dataset as the number of attributes increases, yielding smaller MBHs. This explains the different slopes of the curves as the number of attributes increases. Contrasting with figure 11(b) which presents the results of the same experiment for random data, we observe that the overall trends are the same, the curves however for random data, especially as the number of attributes increase are steeper (have higher slope). This is expected, since the distribution is not as skewed and as a result, a larger fraction of the



(a) Correlated dataset



(b) Random dataset

Figure 11: Varying the number of attributes

preference attribute space is covered for the same number of materialized views. The error due to the approximation of the solution with MBHs increases with the number of attributes, but again is not the dominant source of error.

Query performance as a function of required guarantees. Figure 12 presents the results of an experiment assessing the query performance of PREFER as a function of the guarantees requested. We use four-attribute datasets in this experiment. We vary the guarantees provided by the queries, by increasing the maximum number of view tuples read to report the first result of queries. Figures 12(a)(b) show the results for the correlated dataset for two dataset sizes, and Figures 12(c)(d) show the results for the random datasets.

In each figure we report two curves each for different number of materialized views. We observe that in all cases, with twenty views, the majority of queries satisfy a guarantee as small as 500 tuples. A similar phenomenon with the impact of skew exists in this case. For random data (Figure 12(c)(d)) for the same dataset size the fraction of queries providing a specific guarantee is higher than in the case of correlated data.

Comparison With The Onion Technique. Figure 13 presents an experimental comparison or PREFER against the Onion technique, which was briefly described in Section 2. We implemented the Onion technique and we report on the number of tuples retrieved from the database, for a database with 50K tuples and 3 attributes, increasing the number of query results requested. We vary the number of results requested and the number of views materialized in our technique. The Onion technique requires approximately 2.5 hours to construct the index for such a relation(50K tuples and 3 attributes). The time is exponential to the number of attributes. This was the maximum experiment we could run with the Onion technique that would require a reasonable amount of time for preprocessing.

For this experiment we construct materialized views by imposing a guarantee of 500 tuples only for the first query result (the guarantee is not that important in this case , since we don't cover the whole query space.) Thus the views are constructed in a way that *no guarantees* are provided for additional results with our technique and, so, we level the query performance playground in order to fairly compare with Onion, which is focused on the first result as well. Figure 13(a) presents the results for the correlated dataset. The proposed technique is superior to the Onion technique even with a single view available, for all requested results. We also observe that the performance of our technique deteriorates slightly as the number of requested tuples increases. This is not the case for the Onion technique. The performance deteriorates rapidly and when more than 20 results are requested it has to scan the entire dataset. This is because this dataset is decomposed into 20 convex hulls by the Onion technique. It is interesting to notice that in this experiment the views are constructed with a guarantee of 500 tuples only for the first result. Even in this case, the proposed technique is capable of outperforming the Onion technique for all requested results. Figure 13(b) presents the results for the random dataset. We observe that when only one view is available, the Onion technique is better for the first result, but its performance deteriorates rapidly for additional results. Moreover as the number of views increases, our technique becomes much better for all results retrieved, even though the views where constructed without guarantees for additional results. For more than 10 results, the Onion technique essentially performs a scan of the entire dataset, because there are only 10 convex hulls in the Onion index.

Query running time comparison to a commercial DBMS. We present results of an experiment that compares the average time that PREFER needs to output the top results of a query, as the number of results varies, to the time that a commercial DBMS requires for the same task. We use a 50000 tuples correlated dataset with four attributes for this experiment. To measure the time of the DBMS, we issue a SQL query containing the preference function in the ORDER BY clause (required to order the result by the score of the preference function) and measure the time to output the top results. PREFER contains 34 materialized views, that are chosen using algorithm View Selection for a guarantee of 500 tuples, in a pre-processing step. This set of views covers the whole preference vector space for that guarantee. The results of the experiment are shown in Fig-

ure 14.

One can observe that the performance benefits are very large. Even for 500 results requested, PREFER still requires half the time of a straightforward SQL based approach. Notice, that the time required by the DBMS is almost the same for all results as the entire relation has to be ranked before a single result is output.

7. CONCLUSIONS

The widespread use of the world wide web as a front end to database systems creates new opportunities for enhanced query capabilities. In this direction we have introduced algorithms to enhance database selection queries with user preferences. Our algorithms make use of multiple database views and are able to provide performance guarantees for the types of selection queries considered in this paper. We presented a methodology to derive the answer of a preference selection query from a materialized view containing the output of another preference selection query. We have presented algorithms to select the best views to materialize given a constraint on the available space and have implemented our algorithms on a prototype system called PREFER on top of a commercial relational database management system demonstrating the practical utility of our approach. Our results demonstrate that when compared with other approaches proposed for this problem, one can achieve great savings both in construction time as well as execution time, using our proposed algorithms.

Many research issues remain for exploration. Considering other important database operations in conjunction with the preference framework would be of great interest. Query optimization of such operators as well as various dynamic aspects of their execution are important issues for further study. We plan to investigate these questions in our future and ongoing work.

8. ACKNOWLEDGMENTS

We wish to thank Divesh Srivastava and Pavel Velikhov for very useful discussions.

9. REFERENCES

[1] Y. S. A. Levy, A. Mendelzon and D. Srivastava. Answering Queries Using Views. *PODS*, pages 95–104, 1995.

[2] R. Agrawal and E. Wimmers. A Framework For Expressing and Combining Preferences. *Proceedings of ACM SIGMOD*, pages 297–306, June 2000.

[3] Y. chi Chang, L. Bergman, V. Castelli, C. Li, M. L. Lo, and J. Smith. The Onion Technique: Indexing for Linear Optimization Queries. *Proceedings of ACM SIGMOD*, pages 391–402, June 2000.

[4] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting Aggregate Queries Using Views. *PODS*, pages 155–166, 1999.

[5] H. V. j. D. Srivastava, S. Dar and A. Levy. Answering Queries with Aggregation Using Views. *Proceedings of VLDB*, pages 318–329, 1996.

[6] O. Duschka and M. Genesereth. Answering Recursive Queries Using Views. *PODS*, pages 109–116, 1997.

[7] R. Fagin. Combining Fuzzy Information from Multiple Systems. *PODS*, pages 216–226, June 1996.

[8] R. Fagin. Fuzzy Queries In Multimedia Database Systems. *PODS*, pages 1–10, June 1998.

[9] R. Fagin and E. Wimmers. Incorporating User Preferences in Multimedia Queries. *ICDT*, pages 247–261, Jan. 1997.

[10] A. Guttman. R-trees : A Dynamic Index Structure for Spatial Searching. *Proceedings of ACM SIGMOD*, pages 47–57, June 1984.

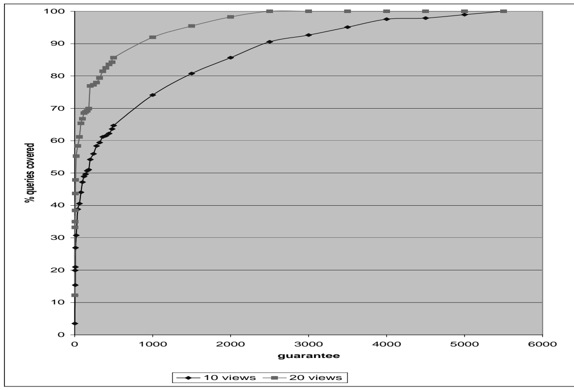
[11] D. Hockbaum. *Approximation Algorithms for NP-Hard Problems.* ITP, 1997.

[12] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Dover.

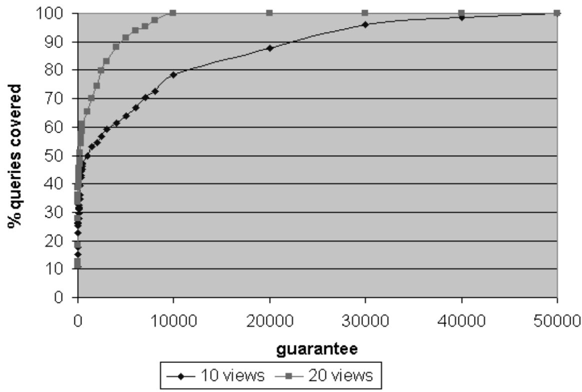
[13] Y. Papakonstantinou and V. Vassalos. Query Rewriting For Semistructured Data. *Proceedings of ACM SIGMOD*, pages 455–466, 1999.

[14] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+ -tree : A Dynamic Index for Multi-dimensional Data. *Proceedings of VLDB 1987*, pages 507–518, Sept. 1987.

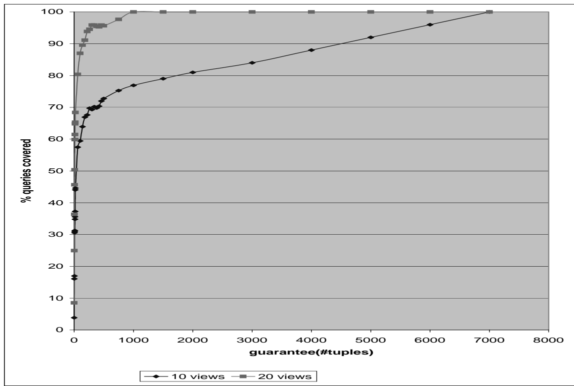
[15] V. Vassalos and Y. Papakonstantinou. Expressive Capabilities, Description Languages and Query Rewriting Algorithms. *JLP*, 2000.



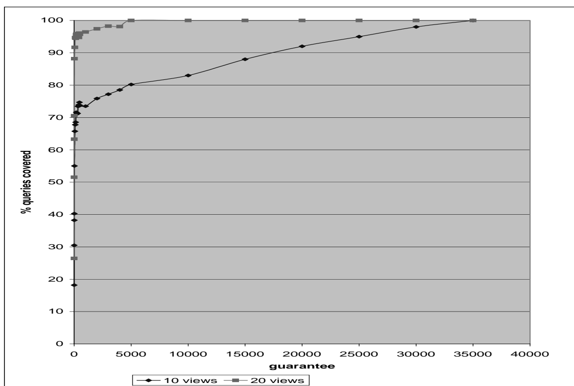
(a) Correlated dataset, 50K tuples



(b) Correlated dataset, 500K tuples

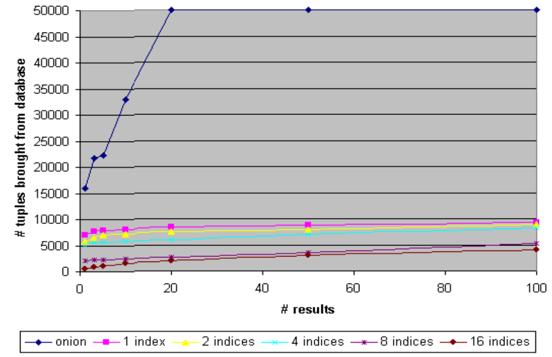


(c) Random dataset, 50K tuples

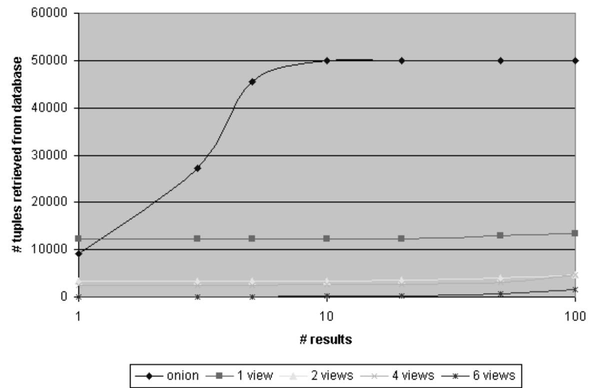


(d) Random dataset, 500K tuples

Figure 12: Varying guarantees



(a) Correlated dataset



(b) Random dataset

Figure 13: Comparison with the Onion Technique

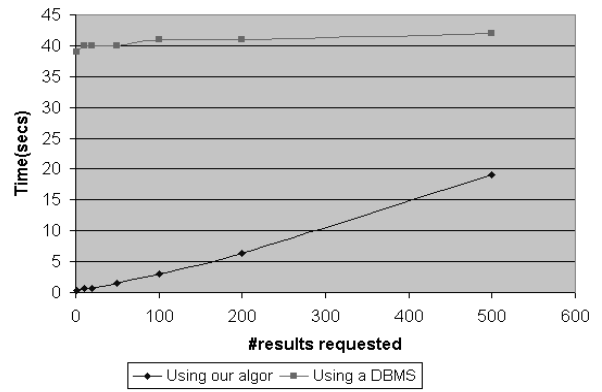


Figure 14: Execution Times