VLDB Journal manuscript No.
(will be inserted by the editor)

Algorithms and Applications for answering Ranked Queries using Ranked Views*

Vagelis Hristidis[1], **Yannis Papakonstantinou**[2]

[1] University of California, San Diego, e-mail: vagelis@cs.ucsd.edu
[2] University of California, San Diego, e-mail: yannis@cs.ucsd.edu

Abstract Ranked queries return the top objects of a database according to a preference function. We present and evaluate (experimentally and theoretically) a core algorithm that answers ranked queries in an efficient pipelined manner using materialized ranked views. We use and extend the core algorithm in the described PREFER and MERGE systems. PREFER precomputes a set of materialized views that provide guaranteed query performance. We present an algorithm that selects a near optimal set of views under space constraints. We also describe multiple optimizations and implementation aspects of the downloadable version of PREFER. Then we discuss MERGE, which operates at a meta-broker and answers ranked queries by retrieving a minimal number of objects from sources that offer ranked queries. A speculative version of the pipelining algorithm is described.

Key words Ranked queries–merge ranked views–materialization

1 Introduction

An increasing number of Web applications allow queries that rank the source objects according to a function of their attributes [7,6,3]. For example consider a database containing houses available for sale. The properties have attributes such as price, number of bedrooms, number of bathrooms, 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 number of bathrooms 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 number of bathrooms. 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 small and have few bathrooms.

The functionality of ranked queries is exposed to the user by interfaces such as the one of the PREFER system, shown in 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.

Other Web applications allow their users to rank their source objects according to a few "canned" ranking functions. For example, many Web-based brokers (e.g., Schwab, E-Trade) allow the user to declare his/her investment goals and then return to the user a list of mutual funds that is ranked in accordance to the goals. For example, if the user declares that he/she is an aggressive long-term investor the system ranks the mutual

* Work supported by NSF Grant No. 9734548.

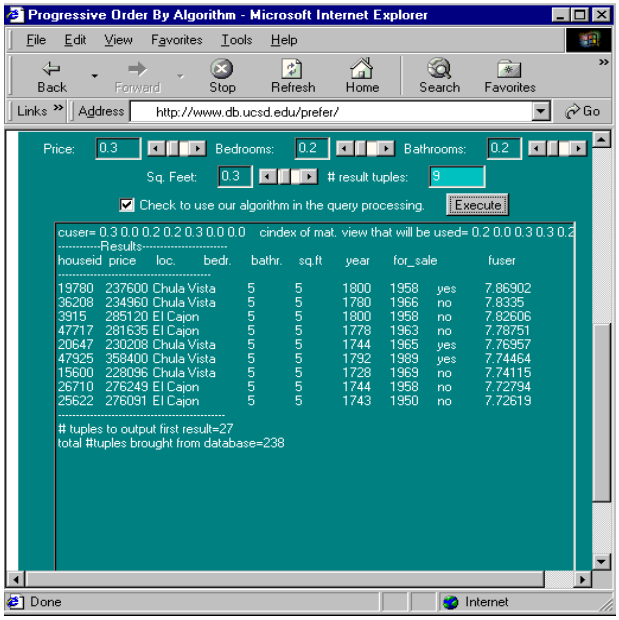


Fig. 1 Preference Queries

funds using a function that gives high weight to the growth rate, low weight to the volatility, and very low weight to the produced income.

The user in the above examples has a *preference* about the importance (or weight) of the attributes associated with the entities (houses and mutual funds) searched. The source objects may be relational tables, documents, images, or other types of data where ranked search makes sense. In this paper we assume that all attributes of an object are contained in a single relation $R(A_1, A_2, \ldots, A_k)$. In the context of the PREFER system, the objects reside in a single source. In the context of MERGE they reside in multiple sources, where each source S_i contains exactly one relation R_i, and R is the union of the relations from all sources ([12,4,18] consider the case where R is "vertically split", i.e., different attributes of the same object are found in each source). The user provides a *preference function* $f(A_1, \ldots, A_k)$ and a *ranked query* returns the tuples of R ordered according to $f(A_1, \ldots, A_k)$.

Given the current database technology, we have to retrieve the whole relation R in order to find the top tuples with respect to a preference function f on the attributes of the relation, except if R is already ordered by f. The key observation behind PREFER and MERGE is that we can efficiently extract the top results for f from a source that ranks the objects by a preference function f', if f' is "close" to f. We apply this observation to solve two problems: (i) to efficiently evaluate ranked queries on a single source and (ii) to efficiently merge the results from multiple sources with different preference function at a *meta-broker*.

We present the *PipelineResults* algorithm that uses a ranked view V to efficiently answer a ranked query q, which may have different weight values than V. A ranked view, which we also refer to as preference view or just view, is a relational view that is ordered according to a preference function. *PipelineResults* traverses V and outputs the top results of q in a pipelined manner. Hence, the top results of q are output by retrieving only a prefix of V. The intuition is that when the query's preference function is "similar" to the view's preference function the required prefix is small.

We define the property that the view and the query preference functions must have, in order to be able to output the top results of the query without retrieving the whole view. We focus on three types of preference functions: (i) linear combinations of the source attributes, (ii) linear combinations of monotone functions (eg: logarithmic) of the source attributes, and (iii) cosine functions (in the spirit of cosine functions used in document retrieval).

The key idea of answering a ranked query using a ranked view has many applications in database and information retrieval problems. We present two applications, PREFER and MERGE, and the solutions to additional problems they have posed.

PREFER. PREFER is a system that lies on top of commercial relational databases and allows the efficient evaluation of ranked queries. PREFER is available to download at *http://www.db.ucsd.edu/PREFER*. Currently, an application that answers ranked queries would have to retrieve the whole database, apply the preference function to each tuple, and sort accordingly.

The PREFER system provides excellent response time for ranked queries, by using pre-materialized ranked views. PREFER works as follows: Given a relation and the performance requirements of the system, it decides which views should be materialized. Then, when a ranked query q arrives, PREFER selects the materialized view V that answers q most efficiently and runs the *PipelineResults* algorithm on V to retrieve the top-N results of q. 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. We study how the performance of PREFER is improved when only a prefix of the views is materialized (in contrast to [21]), since the last tuples of the view are usually not needed in answering a top-N query. We experimentally evaluate this approach.

MERGE. The second application of the *PipelineResults* algorithm that we present in this paper is the MERGE system. MERGE is a system that uses the *PipelineResults* algorithm to merge the ranked results coming from multiple sources at a *meta-broker* (also called meta-searcher). A meta-broker ([16,15]) uses multiple underlying sources to answer to user queries by merging the results that these sources produce (see Figure 2). For example, a real estate meta-broker allows the user to provide the weights he/she assigns to the price, year, and square feet of a house. Then the meta-broker contacts multiple real estate sites, obtains house records and merges them into a single answer list.

An example of such a Web site is *mySimon.com*, which is a meta-searcher for various products like books, computers, etc. It sends the user's query to multiple underlying sources (online retailers) and then merges the results ordered by an attribute of the products. MERGE is more general in that it allows the objects to be ordered according to a function of their attributes. Consider for example two online bookstores that rank their resulting books according to their price and their delivery time, measured in number of days from today. The first bookstore assigns a weight of 0.8 to price and 0.2 to delivery time and the second 0.9 and 0.1 respectively. Now suppose a user request from the meta-broker to rank the books assigning weight 0.7 to price and 0.3 to delivery time. The naive algorithm would get all results from both sources, evaluate the user's preference function for each book and output the ranked results. On the other hand, MERGE only retrieves a prefix of the results from each source and combines these books to output the top results at the meta-broker. It then keeps retrieving, merging and outputting results in a pipelined manner.

The key efficiency problem is how to output the top-N results of the user query by accessing the minimum prefix of the result list that each source produces. This is a hard problem because the sources typically use different ranking functions from each other and from the function requested by the user. The meta-broker needs to efficiently access the sources and merge the results. MERGE uses a merging algorithm that is based on the principles of the *PipelineResults* algorithm and runs at the meta-broker.

Notice that the sources usually have different access speeds. Hence the slower sources become the bottleneck of the query at the meta-broker. For such cases, we use a modification of the merging algorithm, which sacrifices some accuracy of the results in favor of the execution time. The *speculative* version of the merging algorithm retrieves fewer tuples from the slower sources in order to output the top-N results of a ranked query. However, there is a chance that a tuple from a slower source is mistakenly present or absent from the output results. The speculation applied to each source is proportional to its response time. We experimentally evaluate the effects of speculation and show that it considerably boosts the performance of the MERGE system.
In summary, this paper makes the following contributions:

- We present an algorithm that computes the top-N results of a query by using the minimal prefix of a ranked view. Preference functions that are linear combinations of monotone single-attribute functions and cosine preference functions are supported. We also provide a probabilistic analysis that shows how the relationship between the view and the query preference functions influences the performance of the *PipelineResults* algorithm.
- We specify the set of queries for which a view can provide a guarantee about the number of view tuples examined in order to provide the top-N tuples of the query.
- We present an approximation algorithm to the NP-hard problem of selecting the "best views" to build in PREFER, 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

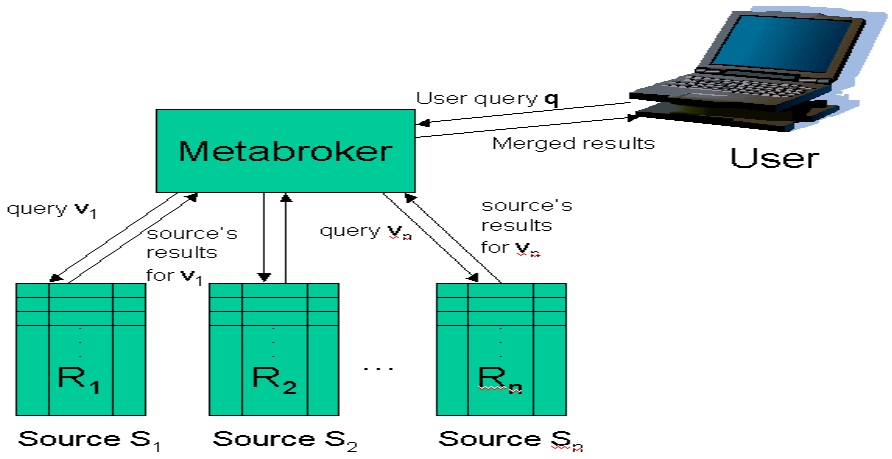


Fig. 2 Meta-broker Architecture

queries. We also show that the performance can be improved by storing only prefixes of views, as opposed to whole views, and utilize the space to precompute and save more views.

- We present a pipelined algorithm that merges the ranked results from multiple sources by retrieving the minimum prefix from each of them. We also describe a speculative version of that algorithm to handle slower sources. These algorithms have been implemented in the MERGE system.
- The performance of PREFER scales with the number of views that are materialized. 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 a detailed experimental evaluation comparing our proposed algorithms with current state of the art and show that our approach provides good scalability. In particular, we show that PREFER scales well both in terms of data set size and number of attributes. MERGE, on the other hand, scales well with the number of sources. We also examine the importance of the distance between the preference functions of the meta-broker and the sources.
- We have developed PREFER[1] on top of a commercial database management system, demonstrating the practical utility of our overall approach. A user-friendly interface is provided that allows the easy deployment of PREFER on any database.

The paper is organized as follows: In Section 2 we discuss related efforts and describe how this work is related to our prior work ([21]). Section 3 describes the definitions and the notation used. Section 4 presents the *PipelineResults* algorithm and methods for calculating the watermark value for various preference function types. It also presents the condition that must hold for a preference function for which a watermark is computable, and some improving modifications to the *PipelineResults* algorithm. Section 5 and 6 present the PREFER and the MERGE systems respectively. A detailed experimental evaluation of the systems is presented in Section 7. Finally, the details of the implementation of PREFER are presented in Section 8.

2 Related Work

There is a considerable amount of work on the problems of optimizing ranked queries and merging ranks from multiple sources. First we present related work on the problem of answering a ranked query using a view. Second, the work relevant to PREFER and MERGE is presented.

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 [3] put the notion on preferences into perspective and introduce a framework for their expression and combination. 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 [3]

The problem of answering a ranked query using a ranked view is addressed in [16]. They defined the term "user query is manageable by source" to indicate that a ranked user query can be answered using only a

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

prefix of the source data. [16] made the same architectural assumptions with our work (known source ranking function, metabroker retrieves a prefix the source) but did not focus on any specific class of ranking functions. Consequently, [16] did not provide a prefix computing algorithm. Our work focuses on a specific class of functions described in Section 3 and fully solves the problem for this class of functions.

A significant amount of work has been published the last years on answering queries using views [19]. The earlier work focused on conjunctive queries and views (e.g., [2]) and subsequent work extended into more powerful queries, views, and view set descriptions [9,27,25]. Rewriting aggregate queries using views has also been addressed [26,8]. 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 PREFER, is the work by Chang et. al., [6]. 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 key observation behind Onion 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 a few 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 dependent on the characteristics of the dataset and no guarantees in performance can be provided. A major advantage of PREFER over the Onion technique is that the performance scales gracefully as the available space increases. In contrast, Onion does not exploit the availability of some extra space. We evaluate the performance of Onion in Section 7.

Goldstein and Ramakrishnan [14] provide a framework similar to PREFER for the case of nearest neighbor (NN) queries. In particular, they propose the P-Sphere tree, which materializes a set of sample NN queries, which are subsequently used to efficiently evaluate other NN queries. Their approach scales well when the available space increases, similarly to PREFER. However, their algorithm is less involved, since once the right P-Sphere p is found, the points in p are searched for the NN. In contrast, in our work, finding the right view V is just one of the challenges. Next, we execute a pipelining algorithm that retrieves the minimal prefix of V to answer the ranked query. Another difference is that when a query point in [14] is not contained in any P-Sphere, the index (P-Sphere tree) is not used in calculating the NN. Instead a traditional linear cost NN algorithm is used. In contrast, in our work, when a query point is not covered by any view, the most "suitable" view is used to evaluate the query, which leads to a better performance than the naive algorithm (scan whole database).

The MERGE work assumes that the source ranking functions are known. This is an assumption that does not generally apply today to Web sources and search engines. Such systems typically do not disclose their ranking functions. Our work can benefit by a class of works that propose ways to get information about the source ranking functions by asking training queries as in [28] or by calibrating the document scores of the sources using statistics as in [5]. We can employ similar training techniques to learn the source ranking functions if they are not given. Equally important for our work are recent initiatives, such as [15] and [22], that allow search engines to export their ranking functions.

To the best of out knowledge, no work has been published on the problem that MERGE tackles, i.e., merging the union of the ranked results from multiple sources. However many papers have been published on answering ranked queries when the information about each object is distributed across multiple sources. In [12], [18], [17], [23], algorithms are provided to combine ranked lists of attributes in order to efficiently retrieve the top results according to an aggregate function of the attributes. In these papers a sorted list is used for each attribute in order to efficiently retrieve the top-N ranked results from a single source. [4] provides algorithms to retrieve the top-N results, even when some of the sources only allow random access. MERGE is different because our sources order different objects according to functions of their attributes, instead of ranking the same objects by one attribute each time.

Notice the difference between the *threshold value*, defined in [12] and [4], and the *watermark value* defined in this work. In particular, the threshold value is the minimum query score that the N-th result tuple so far must

have in order to output the top result tuple. On the other hand, the watermark value is a score with respect to the ranking function of the view and not of the query, and determines how deep in the ranked view we must go to output the top result tuple.

Multimedia sources have also received significant attention in the context of ranked queries [10,13,11,7,12]. [10] and [7] propose rank merging solutions for such sources, where objects are ranked according to how well they match the query values. The solutions in [10], [7] and [12] are based on a richer architectural framework than the one we assume in our work. In particular, our algorithm does not require any random accesses to the sources, which is a very important property as explained in [12]. Other random access assumptions have been made in [10] and [7].

3 Notation and Definitions

This section defines queries, views, sources and other relevant notation in the context of PREFER (first) and MERGE (second). 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.

The preference function $f_q(t)$, $\prod_{i=1}^{k}[m_i, M_i] \rightarrow \mathcal{R}^+$ defines a numeric *score* for each tuple $t \in R$. Every query q consists of a *preference function* $f_q(.)$ and a single relation 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. We use the term *query space* to refer to all possible valid preference queries that can be presented to a relation.

Notice that the preference functions handled by PREFER need to satisfy the monotonicity property described in Section 4.2. Some common functions do not satisfy this property. For example, in the houses database, a preference function that ranks the houses according to their distance from an arbitrarily chosen point cannot be handled, because the ranking of a house also depends on the chosen point and not only on the house's X and Y coordinates.

The algorithms presented in this paper are applicable for a wide class of preference functions, as described in Section 4.2. In this paper we provide detailed algorithms for three kinds of preference functions:

- linear, $f_{\mathbf{v}}(t) = \sum_{j=1}^{k} v_j A_j(t)$
- linear combination of monotone single-attribute functions, $f_{\mathbf{v}}(t) = \sum_{j=1}^{k} v_j \cdot h(A_j(t))$, where $h(A_j(t))$ is monotone for $j = 1, \ldots, k$. For example, $h(.) = log(.)$.
- cosine, $f_{\mathbf{v}}(t) = \frac{1}{|\mathbf{t}||\mathbf{v}|} \sum_{j=1}^{k} v_j A_j(t)$, where $|\mathbf{t}|$ denotes the Euclidean norm of vector $\mathbf{t}$.

We chose these functions because they are widely used in Web and multimedia applications that require ranking and they can be efficiently pipelined using the techniques we present in this paper. The vector $\mathbf{v} = (v_1, \ldots, v_k)$ 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 $\mathbf{v}$. 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, since 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.

The following additional definitions are specific to the MERGE system (Section 6). MERGE operates on top of multiple sources $S_1, \ldots, S_n$. Each source S_i exports a relation R_i. The exported relations have the same schema or at least some common attributes. A metabroker [16] M over $S_1, \ldots, S_n$ exports the relation $R = R_1 \cup R_2 \cup \ldots \cup R_n$ if all relations have the same schema. If the sources do not have the same schema, R is defined as follows:

- The set of attributes $(A_1, \ldots, A_k)$ of R is the union of the sets of attributes of $R_1, \ldots, R_n$.
- If the relation $R_i(A_{i_1}, \ldots, A_{i_m})$ has a tuple $t \equiv [A_{i_1} : a_{i_1}, \ldots, A_{i_m} : a_{i_m}]$ then the relation R has a tuple $t' \equiv [A_1 : a_1, \ldots, A_k : a_k]$ where $a_j = a_{i_l}$, if $A_j \equiv A_{i_l}$ and $a_j = NULL$, if $A_j \notin \{A_{i_1}, \ldots, A_{i_m}\}$.

Each source S_i, in MERGE, supports a set Q_i of preference queries over the exported relation R_i. A metabroker [16] typically has two problems. First, to choose the right ranked query to invoke at each source and second

to merge the results. We are concerned with the second problem. To simplify the abstraction we assume that each source outputs a single ranked query. The metabroker supports preference queries over R that have the same kind of preference function with the underlying sources, which also have the same kind of preference functions with each other. We need efficient pipelined execution of the queries on R. In particular we want to be able to derive the top-m tuples $[t_q^1, \ldots, t_q^m]$ from relation R according to query q by reading a minimal subset of each relation R_i.

4 Pipelining 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 $\mathbf{v}$, 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 $\mathbf{q}$. 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.[2] 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 construct such an "oracle". Section 4.2 discusses when such an "oracle" is possible and provides guidelines for the computation of the watermark for such functions. Detailed computations are presented for linear, linear combination of monotone single-attribute functions, and cosine functions. Section 4.3 presents an example of using this algorithm. Section 4.4 describes a speculative version of the pipelining algorithm. Finally, Section 4.5 provides a probabilistic analysis of the 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 $T_{v,q}^1 \in \mathcal{R}^+$ with the property:

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

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

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

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

The theorem guarantees that the top-s tuples, according to $f_q(.)$, in the prefix $[t_v^1, t_v^2, \ldots, t_v^{w-1}]$ 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.

[2] 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.

Rank in View	*% of queries*
1	80
2-20	13.3
21-3000	6.7

Fig. 3 Rank of top query tuple in view

4.1 The Core of the Pipelining Algorithm

The algorithm *PipelineResults* in Figure 4 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 from WINDOW, which stores the set of tuples that have been retrieved from R_v, but have not been output yet. We select as t_v^{top} the tuple in WINDOW that maximizes f_q [3], and repeat the process. A new sequence of tuples having the highest score according to q among the remaining tuples will be determined and output.

Notice that the number of retrieved tuples could be reduced by re-calculating the watermark value on the fly for each retrieved tuple, instead of only calculating the watermark once for each tuple that is output. We call this modified algorithm *PipelineResultsOptimal*. In particular, the following lines replace lines 4 and 7.

```
4a. repeat {
4b.   Get next tuple t_v^w from R_v
4c.   Let T_{v,q}^w = DetermineWatermark(t_v^w)
4d.   If T_{v,q}^w > T_{v,q}^top
4e.   then{
4f.     T_{v,q}^top = T_{v,q}^w
4g.     Set t_v^top = t_v^w  }
4h. } until (f_v(t_v^w) < T_{v,q}^top)

7a.  Let s+1 be the index of the first tuple in the sorted order whose watermark
         is lower than f_v(t_v^{w-1})
    /* Notice that s will always be equal or greater than the index of t_v^top*/
```

However, this modification would require calling the $DetermineWatermark(.)$ as many times as the number of retrieved tuples, instead of the number of output tuples. Furthermore, we have found that this modification only marginally reduces the number of retrieved tuples. For example, in the case where we have 50000 tuples in the database, with 3 attributes and 6 materialized views, the number of retrieved tuples is reduced by a factor between 0.3 and 0.7%, when the number of requested results varies from 1 to 100. The intuition behind this for the case of top-1 queries is that the top tuple of the view is usually also the top tuple of the query (see Figure 3), so it produces a maximal watermark value. Hence, we do not use this modification in the experiments.

The correctness of the *PipelineResults* algorithm is proved as a special case of Theorem 8 of Section 6, which considers multiple sources as well. The correctness of the *PipelineResultsOptimal* algorithm is proved in the same way. The *PipelineResultsOptimal* algorithm is optimal in the sense described in Theorem 2.

Theorem 2 *There is no algorithm that given a prefix of R_v, and the preference functions f_v and f_q, outputs a larger prefix of R_q than the PipelineResultsOptimal algorithm.*

Proof: Assume that there was such an algorithm A. Assume that A outputs one more tuple t than $PipelinedMergeOptimal$ by retrieving the same prefix P of R_v. Since t was not output by $PipelinedMergeOptimal$ it means that its watermark is lower than $f_v(t_v^{w-1})$, that is, lower than the $f_v(.)$ score of the last tuple of P. Then, by the fact that the watermark is optimally (tightest) calculated, there could be a tuple t' not yet retrieved, with $f_v(t_v^{w-1}) > f_v(t') > T_{v,q}^{w-1}$ and $f_q(t') > f_q(t)$. That is, t' should have been output before t. □

[3] In [21] we select as t_v^{top} the next unprocessed tuple of R_v, which is less efficient because it leads to a smaller watermark value.

Corollary 1 *There is no sequential scan algorithm that retrieves less tuples than PipelinedMergeOptimal to answer a ranked query.*

```
Algorithm PipelineResults(R_v,q,v, N){
1.  Let t_v^top ≡ t_v^1
2.  Let WINDOW= ∅
3.  while (less than N tuples in the output) {
4.   Let T_{v,q}^top = DetermineWatermark(t_v^top)
5.   Scan R_v and determine the first tuple t_v^w with f_v(t_v^w) < T_{v,q}^top
6.   Add all tuples t ∈ [t_v^1, t_v^{w-1}] to temporary relation WINDOW
 /*Notice that t_v^1 denotes the first tuple of R_v that has not been retrieved yet*/
7.   Sort WINDOW by f_q
8.   Let s be the index of t_v^top in the sorted order
9.   Output the first s tuples from WINDOW
10.  If the size of WINDOW is s then set t_v^top ≡ t_v^w
11.  else let t_v^top be the tuple with index s+1 in WINDOW.
12.  Delete the first s tuples from WINDOW.
 }
}
```

Fig. 4 Algorithm to output the first N tuples according to q

4.2 Determining the Watermark

In this section we present a theorem that specifies when the calculation of a $useful$ watermark value is possible and general directions for calculating the watermark value $T_{v,q}^{top}$ for two arbitrary preference functions f_q, f_v. Then we describe algorithms for the calculation of the watermark value in the case of linear, linear combination of monotone single-attribute functions, and cosine functions.

A $useful$ watermark value can be calculated for a source preference function f_v and a ranked query q with preference function f_q, when there exist instances of the source relation R_v such that the top results of q can be found before retrieving all the tuples in the source. This property is equivalent to the *manageability* property in [16], where a preference function f_q is called *manageable* at a source S if

$$\exists 0 \leq \epsilon < 1, \ such\ that\ \forall t \in R, \ f_q(t) \geq f_v(t) - \epsilon \tag{2}$$

That is, the manageability property, the existence of a useful watermark and Equation 2 are equivalent. Notice that this definition assumes that $0 \leq f_q(t), f_v(t) \leq 1$.

The following theorem is presented in [16].

Theorem 3 *Given the source and the query preference functions $0 \leq f_v(t), f_q(t) \leq 1$ respectively, a useful watermark value can be calculated iff Equation 2 is satisfied.*

Theorem 4 *Given the source and the query preference functions $0 \leq f_v(t), f_q(t) \leq 1$ respectively, if there is an attribute A_l such that f_v and f_q have the same proper monotonicity on A_l, then a useful watermark value can be calculated.*

Proof: By Theorem 3, it is enough to prove that Equation 2 is satisfied. We select

$$\epsilon = max(f_v(t) - f_q(t)), \ t \in [0,1]^m \tag{3}$$

We prove that $\epsilon < 1$. ϵ can be 1 only when there is a tuple t', such that $f_v(t') = 1$ and $f_q(t') = 0$. Without loss of generality, assume that both f_q and f_v are properly increasing with A_l. Then $f_v(t') = 1$ and $f_q(t') = 0$ imply that $A_l(t') = 1$ and $A_l(t') = 0$ respectively. Hence we come to a contradiction, so $\epsilon < 1$. □

The following Corollary, specializes Theorem 4 for linear functions.

Corollary 2 *Given the source and the query linear preference functions* $f_v(t) = \sum_{j=1}^{k} v_j A_j(t), f_q(t) = \sum_{j=1}^{k} q_j A_j(t)$ *respectively, a useful watermark value can be calculated if there is an index* $1 \leq l \leq k$*, such that both* v_l *and* q_l *have the same sign.*

Next, we present a general formula that calculates the watermark value $T_{v,q}^{top}$ for a tuple t^{top} and two arbitrary preference functions f_v, f_q, which satisfy Theorem 3. From Equation 1 we get

$$f_q(t) < f_q(t^{top}) \;\Leftrightarrow\; f_v(t) < f_q(t^{top}) + f_v(t) - f_q(t)$$

and since $f_v(t) < T_{v,q}^{top}$, we have the following Theorem:

Theorem 5 *The watermark value* $T_{v,q}^{top}$ *for a tuple* t^{top} *and two arbitrary preference functions* f_v, f_q *is*

$$T_{v,q}^{top} = f_q(t^{top}) + min(f_v(t) - f_q(t)) \tag{4}$$

where $min(f_v(t) - f_q(t))$ *is calculated over all tuples* t *in the domain of relation* R *that satisfy the equation* [4]*.*

$$f_v(t) \leq f_v(t^{top}) \tag{5}$$

We now focus on three types of preference functions: linear, linear combination of monotone single-attribute functions, and cosine. The inputs of the *DetermineWatermark* algorithm are the user function f_q, the view's function f_v and a tuple t^{top}. Consider that the highest possible $f_v(t)$ in Equation 1 is achieved for an imaginary tuple t'. Thus we will determine the maximum $T_{v,q}^{top} = f_v(t')$ value while satisfying the following equation and thus the watermark.

$$f_q(t') < f_q(t^{top}) \tag{6}$$

4.2.1 Watermark computation for Linear functions Since we know the values of t^{top}, $\mathbf{q}$ and $\mathbf{v}$, 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,

$$f_q(t) = \sum_{i=1}^{k} q_i A_i(t) = f_v(t) + \sum_{i=1}^{k} (q_i - v_i) A_i(t) \tag{7}$$

By substituting Equation 7 into Equation 1 we get

$$\forall t \in R, f_v(t) \leq T_{v,q}^{top} \Rightarrow f_v(t) + \sum_{i=1}^{k} (q_i - v_i) A_i(t) \leq f_q(t^{top}) \tag{8}$$

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

$$f_v(t') + \sum_{i=1}^{k} (q_i - v_i) A_i(t') \leq f_q(t^{top}) \tag{9}$$

We will treat Equation 9 as equality; since the left side of Equation 9 is linear on $f_v(t')$, the corresponding inequality is trivially satisfied. Since our objective is to determine the maximum $f_v(t')$ value that satisfies Equation 9, 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 9 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 $[m_i, M_i]$.

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

$$A_i(t') = \frac{f_v(t') - \sum_{j<>i}^{k} v_j A_j(t')}{v_i} \leq \frac{f_v(t') - \sum_{j<>i}^{k} v_j m_j}{v_i} \tag{10}$$

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 $A_i(t') = min(U_i, M_i)$.

[4] This condition allows the calculation of a tighter watermark value.

$$A_i(t') = \begin{cases} min(\frac{f_v(t') - \sum_{j<>i}^{k} v_j m_j}{v_i}, M_i) & q_i > v_i <> 0 \\ M_i & q_i > v_i = 0 \\ 0 & q_i = v_i \\ max(\frac{f_v(t') - \sum_{j<>i}^{k} v_j M_j}{v_i}, m_i) & q_i < v_i \end{cases} \quad (12)$$

Fig. 5 Bounds for A_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:

$$A_i(t') = \frac{f_v(t') - \sum_{j<>i}^{k} v_j A_j(t')}{v_i} \geq \frac{f_v(t') - \sum_{j<>i}^{k} v_j M_j}{v_i} \quad (11)$$

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 5 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 9 directly, lies on the existence of *min* and *max* terms, with two operands each, in the expressions derived for the attribute bounds (Figure 5). 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 whether $f_v(t')$ 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 9. Thus we have $k+1$ possible expressions for the left side of Equation 9. Each expression, $E_j, 1 \leq j \leq k+1$ is produced by:

$$E_j = f_v(t') + \sum_{i=1}^{k} (q_i - v_i) E(i,j) \quad (13)$$

Theorem 6 *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 $T_{v,q}^{top}$. 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 6. 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^top) {
    for j from k + 1 downto 1 {
      Solve E_j = f_q(t_v^top) and determine watermark
      if watermark ∈ I_j return watermark
    }
}
```

Fig. 6 Algorithm *DetermineWatermark*

$$h(A_i(t')) = \begin{cases} min(\frac{f_v(t') - \sum_{j<>i}^{k} v_j \cdot h(m_j)}{v_i}, h(M_i)) & q_i > v_i <> 0 \\ h(M_i) & q_i > v_i = 0 \\ 0 & q_i = v_i \\ max(\frac{f_v(t') - \sum_{j<>i}^{k} v_j \cdot h(M_j)}{v_i}, h(m_i)) & q_i < v_i \end{cases} \quad (15)$$

Fig. 7 Bounds for $h(A_i)$

4.2.2 Watermark computation for linear combination of monotone single-attribute functions It is $f_q(t) = \sum_{i=1}^{k} q_i \cdot h(A_i(t))$ and $f_v(t) = \sum_{i=1}^{k} v_i \cdot h(A_i(t))$. Following a procedure similar to the one for linear functions, we get

$$f_v(t') + \sum_{i=1}^{k} (q_i - v_i) \cdot h(A_i(t')) \leq f_q(t^{top}) \quad (14)$$

where the bounds are shown in Figure 7.

4.2.3 Watermark computation for Cosine functions The calculation of the watermark for a cosine function differs from the case of linear combination of monotone single-attribute functions because we can not have a separate term in the preference function for each attribute. The reason for this is the $|\mathbf{t}|$ term in $f(t) = \frac{1}{|\mathbf{t}||\mathbf{v}|} \sum_{j=1}^{k} v_j A_j(t)$. This means that we need to calculate an upper bound for $f_q(t) - f_v(t)$ as a whole. Equation 1 can be written as:

$$f_v(t') + (f_q(t') - f_v(t')) \leq f_q(t^{top}) \quad (16)$$

where t' is the tuple that maximizes $f_v(t')$ while satisfying Equation 16 . Now we need to find an upper bound for $f_q(t') - f_v(t')$ to plug it in Equation 16.

$$f_q(t') - f_v(t') = \frac{\mathbf{q} \cdot \mathbf{t'}}{|\mathbf{q}||\mathbf{t'}|} - \frac{\mathbf{v} \cdot \mathbf{t'}}{|\mathbf{v}||\mathbf{t'}|} = \frac{\mathbf{t'}}{|\mathbf{t'}|}(\frac{\mathbf{q}}{|\mathbf{q}|} - \frac{\mathbf{v}}{|\mathbf{v}|}) \quad (17)$$

Since $(\frac{\mathbf{q}}{|\mathbf{q}|} - \frac{\mathbf{v}}{|\mathbf{v}|})$ is fixed for every pair of functions and $\frac{\mathbf{t}}{|\mathbf{t}|}$ has size 1, the above expression is maximized when $\mathbf{t}$ is parallel to $(\frac{\mathbf{q}}{|\mathbf{q}|} - \frac{\mathbf{v}}{|\mathbf{v}|})$ and its maximum value is $|\frac{\mathbf{q}}{|\mathbf{q}|} - \frac{\mathbf{v}}{|\mathbf{v}|}|$. This means that the watermark value for a tuple t^{top} is $f_v(t') = f_q(t^{top}) - |\frac{\mathbf{q}}{|\mathbf{q}|} - \frac{\mathbf{v}}{|\mathbf{v}|}|$.

4.3 An Example

Let us present an example of the algorithm's operation. Assume q is a query with $\mathbf{q} = (0.1, 0.6, 0.3)$ and R_v a view with $\mathbf{v} = (0.2, 0.4, 0.4)$. The preference functions of both the query and the view are linear. Let $m_1 = m_2 = m_3 = 5$ and $M_1 = M_2 = M_3 = 20$. The sequence R_v is shown in Figure 8. To populate table E we use the equations of Figure 5 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.

$$e_1 = 17,\ e_2 = 11,\ e_3 = 14$$

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

Fig. 8 View R_v and scores of each tuple based on f_v and f_q

$T^{top}_{v,q}$	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	$\frac{f_v(t')-12}{0.4}$	$\frac{f_v(t')-12}{0.4}$

Fig. 9 Table E

We are now ready to fill table E. The table is presented in Figure 9. Recall that t^1_v is the first tuple of R_v. Now we solve Equation 9, with t^1_v as t^{top}_v,for each of the 4 intervals starting with the last one. In interval I_4, solving Equation 9 results in $f_v(t') = 8.8$ which is not in I_4 and it is rejected. In I_3 we 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^3_v 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 $t \in [t^1_v, t^3_v]$. We order these three tuples by f_q and output t^2_v and t^1_v. Now in order to get further results we locate the first unprocessed(not yet output) tuple in R_v, which is t^3_v and use it as t^{top}_v in Equation 9. 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^3_v, which we output. Next we use t^4_v in Equation 9 and get $f_v(t') = 8.26$, so the prefix is $[t^4_v, t^6_v]$. We sort these tuples and output t^5_v and t^4_v. Next we use t^6_v in Equation 9 and get $f_v(t') = 7.66$, so our fourth prefix is just t^6_v, which we output. Finally output t^7_v, which is the last unprocessed tuple in R_v.

4.4 Using speculation for approximate results

Very often, the applications that rank their objects according to a preference function do not require that the output sequence is 100% accurate. For example when a user presents a preference query to a database with houses for sale, then the weights that are given are inherently speculative for two reasons. First some properties of a house are hard to be quantified, like the criminality of the region or the quality of the schools near the house. Second, the user has a difficulty to precisely quantify a preference. Hence, many of the applications that we deal with, have an inherent approximation factor. This fact has motivated us to devise methods that sacrifice some accuracy of the results in order to boost the performance. Recall that by performance we mean the number of tuples that we need to retrieve from a view to output the top-N results of a query.

The speculative version of the *PipelineResults* algorithm has the following modification over the original algorithm in Figure 4: We use a higher threshold than the watermark value to determine the window W that contains the top result of q. The *threshold value* T'^{top} is:

$$T'^{top} = (1 + \varepsilon) \cdot T^{top}_{v,q} \tag{18}$$

where $\varepsilon \geq 0$ is the *speculation factor*. Hence, W is smaller and we determine the top result $t^1_{q,spec}$ faster. However, there is a chance that the actual top tuple t^1_q is not contained in W. As the value of ε increases, we retrieve the top results faster, but the accuracy of their ranking decreases. We elaborate more on the speculative version of *PipelineResults* in Section 6.1 , where we show how speculation can boost the performance of the MERGE system.

We define the *cost of speculation* as the average difference between the indices of the tuples in the speculative results sequence and in the real results sequence. In Section 7, we experimentally evaluate the impact of speculation.

4.5 Probabilistic Analysis

The following probabilistic analysis indicates how the expected number Q of tuples retrieved from a view V by the *PipelineResults* algorithm in order to output the top tuple $\mathbf{t}_q^1$, according to query q, increases as a function of the distance of f_q and f_v. We also show how Q is affected by the distribution of the tuples.

We focus on linear functions, although the same analysis applies to any functions satisfying Equation 2. We assume without loss of generality that $m_i = 0, M_i = 1$, for $i = 1, \ldots, k$. We also assume that the database has N tuples $\mathbf{t}_1, \ldots, \mathbf{t}_N$ and each tuple $\mathbf{t}$ has density distribution function $\mathbf{f}_d(\mathbf{t})$. We use the letters $\mathbf{f}$ and $\mathbf{F}$ for density distribution functions (ddf) and cumulative distribution functions (cdf) respectively. Also, bold small letters (eg: $\mathbf{t}$) correspond to vector (tuple) variables and regular letters (eg: x) to number variables.

From Equation 9, the watermark value $T_{v,q}^1$ of $\mathbf{t}_v^1$ is:

$$T_{v,q}^1 = f_q(\mathbf{t}_v^1) - \sum_{i=1}^{k}(q_i - v_i)A_i(\mathbf{t}_v^1) \leq f_q(\mathbf{t}_v^1) - \sum_{q_i > v_i}(q_i - v_i) \tag{19}$$

The above inequality is an approximation of $T_{v,q}^1$ and becomes an equality when $v_i \leq T_{v,q}^1$ for $q_i > v_i$ and $T_{v,q}^1 \leq 1 - v_i$ for $q_i < v_i$. Then

$$T_{v,q}^1 = f_q(\mathbf{t}_v^1) - \frac{\sum_{i=1}^{k}(|q_i - v_i|)}{2} \tag{20}$$

Equation 20 shows clearer how $T_{v,q}^1$ is affected by the distance between f_q and f_v. However, if one of the above conditions does not hold we use Equation 19 instead.

Next, we calculate the ddf of $f_q(\mathbf{t}_v^1)$. First we calculate the ddf $\mathbf{f}_v^1(\mathbf{t})$ of $\mathbf{t}_v^1$. Consider the probability P_1 that a random tuple $\mathbf{t}_i$ is the top tuple $\mathbf{t}_v^1$ in V and $\mathbf{t}_i$ is in $[\mathbf{t}, \mathbf{t} + d\mathbf{t}]$, which is the hyperrectangle defined by the points $\mathbf{t}$ and $\mathbf{t} + d\mathbf{t}$.

$$P_1 = \mathbf{f}_d(\mathbf{t})d\mathbf{t} \cdot \prod_{j<>i}^{N} P(f_v(t_j) < f_v(\mathbf{t}_i)) = \mathbf{f}_d(\mathbf{t})d\mathbf{t} \cdot (\mathbf{F}_{f_v}(f_v(\mathbf{t})))^{N-1} \tag{21}$$

where $\mathbf{F}_{f_v}(x)$ is the cdf of $f_v(\mathbf{t})$. Hence, the probability P_2 that $\mathbf{t}_v^1$ is in $[\mathbf{t}, \mathbf{t} + d\mathbf{t}]$ is

$$P_2 = N \cdot \mathbf{f}_d(\mathbf{t})d\mathbf{t} \cdot (\mathbf{F}_{f_v}(f_v(\mathbf{t})))^{N-1} \tag{22}$$

Hence

$$\mathbf{f}_v^1(\mathbf{t}) = N \cdot \mathbf{f}_d(\mathbf{t}) \cdot (\mathbf{F}_{f_v}(f_v(\mathbf{t})))^{N-1} \tag{23}$$

The cdf $\mathbf{F}_{f_q}^{v,1}(x)$ of $f_q(\mathbf{t}_v^1)$ is

$$\mathbf{F}_{f_q}^{v,1}(x) = \int \ldots \int^{f_q(\mathbf{t})<x} \mathbf{f}_v^1(\mathbf{t})d\mathbf{t} \tag{24}$$

Let $\mathbf{F}_T$ be the cdf of $T_{v,q}^1$. From Equations 20 and 24, it is

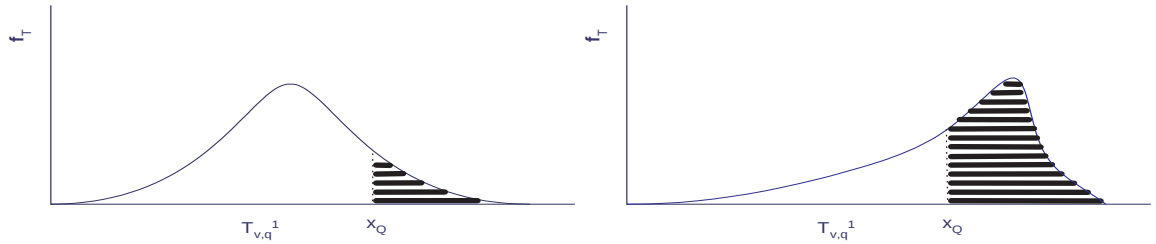
$$\mathbf{F}_T(x) = \mathbf{F}_{f_q}^{v,1}(x - \frac{\sum_{i=1}^{k}(|q_i - v_i|)}{2}) = \int \ldots \int^{f_q(\mathbf{t})<x-\frac{\sum_{i=1}^{k}(|q_i-v_i|)}{2}} \mathbf{f}_v^1(\mathbf{t})d\mathbf{t} \tag{25}$$

Let x_Q be the value of $T_{v,q}^1$, such that the expected number of tuples retrieved from V to output $\mathbf{t}_q^1$ is Q. That is

$$Q = N \cdot \int_{x_Q}^{1} \mathbf{f}_{f_v}(x)dx \tag{26}$$

We show that the probability that $T_{v,q}^1 \geq x_Q$, i.e., the number of tuples retrieved is less than Q, increases as f_v and f_q get closer. It is:

$$P(T_{v,q}^1 \geq x_Q) = 1 - \mathbf{F}_T(x_Q) = 1 - \int \ldots \int^{f_q(\mathbf{t})<x_Q-\frac{\sum_{i=1}^{k}(|q_i-v_i|)}{2}} \mathbf{f}_v^1(\mathbf{t})d\mathbf{t} \tag{27}$$



(a) f_q and f_v are not close (b) f_q and f_v are close

Fig. 10 ddf of $T^1_{v,q}$

Using Equation 23, we get

$$P(T^1_{v,q} \geq x_Q) = 1 - \int \ldots \int_{f_q(\mathbf{t}) < x_Q - \frac{\sum_{i=1}^{k}(|q_i - v_i|)}{2}} \mathbf{f}_d(\mathbf{t}) \cdot N \cdot (\mathbf{F}_{f_v}(f_v(\mathbf{t})))^{N-1} d\mathbf{t} \quad (28)$$

Equation 28 shows that when $f_v(\mathbf{t})$ and $f_q(\mathbf{t})$ are close, $P(T^1_{v,q} \geq x_Q)$ is maximized, which happens when the integral is minimized. This happens for two reasons: First, if we ignore the term $\frac{\sum_{i=1}^{k}(|q_i - v_i|)}{2}$, the integral is minimized because it is calculated over all small values of $f_q(\mathbf{t})$. If $f_v(\mathbf{t})$ and $f_q(\mathbf{t})$ are close, then $f_v(\mathbf{t})$ is also small, hence the integral is minimized. Intuitively, this reason holds because $f_q(t^1_v)$ increases as $f_v(\mathbf{t})$ and $f_q(\mathbf{t})$ get closer. Second, the term $\frac{\sum_{i=1}^{k}(|q_i - v_i|)}{2}$ further minimizes the integral as $f_v(\mathbf{t})$ and $f_q(\mathbf{t})$ get closer. The variance of the watermark value with respect to the distance between $f_v(\mathbf{t})$ and $f_q(\mathbf{t})$ is shown graphically in Figure 10, where the selected areas are equal to $P(T^1_{v,q} \geq x_Q)$.

5 Using the *PipelineResults* algorithm to efficiently answer ranked queries: The PREFER system

The PREFER system runs at a single source, which contains a relation R, and aims at efficiently answering ranked queries on R. It materializes in advance multiple views in order to provide short response time to client queries. Before any query arrives, it builds a set of views that rank R according to several preference functions. This preprocessing process is carried out by the *view selection* module (see Figure 11). When a query q arrives, PREFER selects the "best" view V available, as described in Section 5.3. Then, the *PipelineResults* algorithm is executed to answer q using V. In this section we focus on linear preference functions. The conclusions that we reach can easily be carried to logarithmic and cosine functions.

In its simplest version the view selection module (see Figure 11) 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 materializes a set of view sequences $\mathcal{V}$ such that for every query q there is at least one 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 7 we show experimentally that the number of views needed to cover the whole space of possible queries by retrieving at most 1% of the tuples of R_v is in the order of 10 to 100, when the number of attributes is two to five. In particular, the number of ranked views grows exponentially with the number of attributes due to the exponential growth of the query space. 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. Furthermore, in Section 5.2.1, we present a novel approach to decrease the required space with minor performance degradation, by only storing prefixes of the materialized views. The details and the properties of the view selection algorithm are described in Section 5.2. Note that, in a similar fashion, PREFER

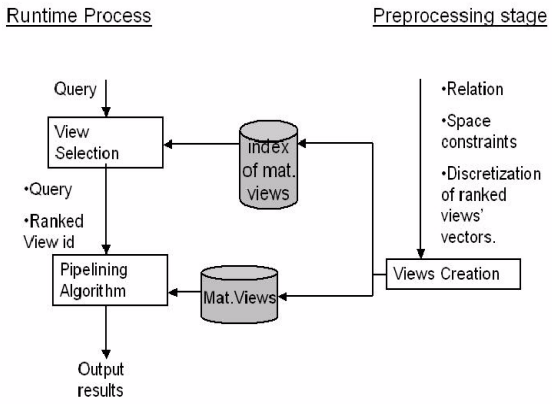


Fig. 11 PREFER's Architecture

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 definition 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* **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[5] whether q is covered by R_v using l tuples.[6] 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 $S^l_{R_v}$.

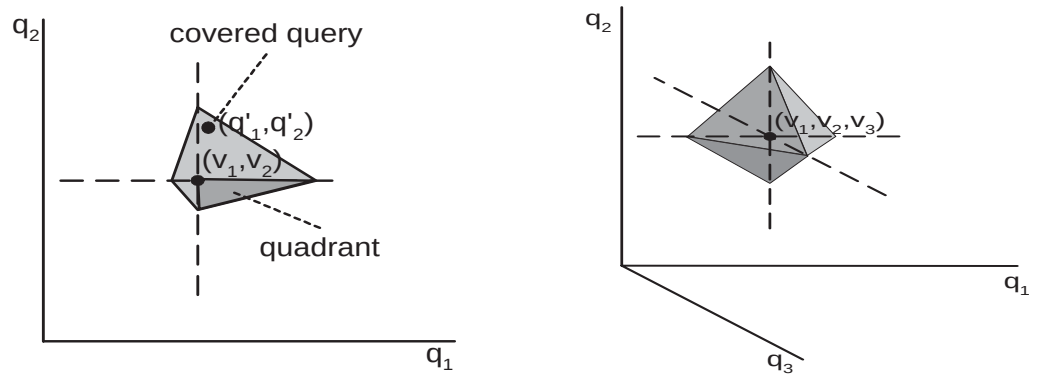
For both algorithms the key point is the following: Since we want to guarantee that at most l tuples from R_v will be read whenever a query q uses R_v we have to place the first watermark at t^l_v or higher. By the watermark properties and a mathematical manipulation similar to the one of Section 4.2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

$$f_v(t^l_v) + \sum_{i=1}^{k}(q_i - v_i)A_i(t^l_v) \leq f_q(t^1_v) \tag{29}$$

In Equation 29 the 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 29. The view

[5] for a fixed number of attributes k.

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



(a) 2D query space (b) 3D query space

Fig. 12 Coverage area of a view

cover problem requires solving Equation 29, which is a linear function. Its solution $S^l_{R_v}$ is in general a convex polytope [24] and, in particular, it is a convex diamond-like shape (i.e., a polyhedron) where all corners lie on the axes centered at $(v_1, \ldots, v_k)$ and there is exactly one corner on each semi-axis. The coverage areas for two and three attributes are shown in Figure 12. We solve Equation 29 in each specific k-dimensional "quadrant". We define as quadrant the space of $\mathbf{q}$ where the relationship between each q_i and v_i pair is constant, i.e., for any two points $q[q_1, \ldots, q_k]$ and $q'[q'_1, \ldots, q'_k]$ in a quadrant, it is $(q_i - v_i) \cdot (q'_i - v_i) > 0$ for $i = 1, \ldots, k$. Thus in a specific quadrant we always use the same case from the cases shown in Figure 5 for the bounds $A_i(t^l_v)$. Furthermore we can pick the right argument of the *min* and *max* terms , since all the terms of their arguments are known values. Hence in a specific quadrant Equation 29 is of the form $\sum_{i=1}^{k} c_i \cdot q_i \leq C$, where c_i's and C are constants, which describes a halfspace. This halfspace contains the point $(v_1, \ldots, v_k)$ since $f_v(t^l_v) \leq f_q(t^1_v)$. The intersection of these halfspaces for all 2^k quadrants is a diamond that is centered at the point $(v_1, \ldots, v_k)$. Notice that the size and shape of the diamonds for two different views is different.

Notice that if a query point does not satisfy Equation 29 it may still be covered by the view R_v, if the *PipelineResultsOptimal* algorithm is used, that is, on-the-fly watermark calculation is performed for each retrieved tuple. However this is not possible when a watermark is calculated only for t^1_v as in Figure 4. PREFER only uses the view cover decision algorithm to avoid this complexity.

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. The queries falling inside the intersection of the corresponding convex polytopes 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 coverage areas $S_1, \ldots, S_N$. The queries falling in $\bigcap_{i=1}^{N} S_i$ satisfy all guarantees.

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. This generates the set of points $\{(x_1, \ldots, x_k) | x_i = r_i d, r_i \in Z, x_i \in [0,1], \sum_{i=1}^{k} x_i = 1\}$ 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. Notice that if the query workload is known a priori and the whole query space cannot be covered due to space limitations, there is an opportunity to cover more queries by placing the views close to the queries. However, we assume that no such information is available and that all queries are equally possible.

There are environments where only a finite number of views, C can be actually materialized. This can be due either to space constraints or to maintenance issues related to updates of the database. 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.

```
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 ⋃ S_v
 }
}
```

Fig. 13 Ranked View Selection Under Space Constraint

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

Problem 1 is an instance of the *maximum coverage problem* [20], 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 7 (Greedy Approximation) *The Greedy Heuristic is an* $1 - \frac{1}{e}$ *approximation for maximum coverage.*

Proof: See [20].

The Greedy heuristic works iteratively by picking the next view from the collection $R_v^1, \ldots, R_v^s$ that covers the maximum number of uncovered elements of $[0,1]^k$. Figure 13 summarizes our approach.

5.2.1 Decreasing the Depth of the Views In environments with space constraints, updates of the database and efficient view maintenance considerations, building a large number of materialized views becomes expensive. The obvious solution would be to materialize a small number of views. This would mean that only a portion of the query space would be covered. Furthermore, when the query space is adequately covered by views then only the top prefix of the views is used in answering the queries when the number of requested tuples is relatively small. These observations led us to experiment with materializing only a prefix of each view. We add an extra step to the views selection and creation process. We select the views that will be materialized as described in Section 5.2 and then we materialize only a prefix of them of size D, which we call *depth* of the views. Then when a query q is presented to PREFER, the best view V is selected using the algorithm in Section 5.3 and the *PipelineResults* algorithm is executed. If we need to retrieve more than D tuples from V in order to output the top-N results that the user requested then this view V may not contain the top-N results according to q. In this case, we use the original relation R in order to answer the query. Alternatively, we could also try using other views "close" to q and if these also fail use R. Fortunately this is very rarely the case when the number of views and their depth are adequetely big and the user only cares about the top results. We further elaborate on the optimal view depth in Section 7.

5.3 Selecting A Ranked View for a Preference Query

Query processing, once C views have been materialized proceeds as follows. The description of the coverage areas of the views (Equation 29) are stored in a main memory data structure. When a query q arrives, we find the view that minimizes the expression $f_v(t_v^l) + \sum_{i=1}^{k}(q_i - v_i)A_i(t_v^l) - f_q(t_v^1)$. That is, the inequality of Equation 29 is more "strongly" satisfied.

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 not covered by any of the stored views. For such queries, we cannot provide performance guarantees based on our construction. The same heuristic as for the covered queries is used. That is, the inequality of Equation 29 is not satisfied but R_v is "closer" to satisfaction than the other views.

6 Using the *PipelineResults* algorithm to efficiently answer ranked queries at a meta-broker: The MERGE system

The $PipelinedMerge$ algorithm presented next efficiently queries the underlying sources $S_1, \ldots, S_n$ and merges their results, when a query q is presented to the metabroker, as shown in Figure 2. It outputs the results in a pipelined manner, i.e., without retrieving the complete result sequences from the sources. We assume that each source S_i exports exactly one query v_i. The key to the algorithm is the computation of a prefix $R^1_{v_i}$ of the result sequence R_{v_i} of source S_i that is sufficient to assure that if the first tuple t^1_q of $R = R_1 \cup R_2 \cup \ldots \cup R_n$ according to f_q is in R_i, then t^1_q is in $R^1_{v_i}$. In each step, a watermark value is calculated for each of the sources.

The algorithm $PipelinedMerge$ is described in Figure 14. It inputs: (a) the preference function $f_{v_1}, \ldots, f_{v_n}$ for each source, (b) the result sequences $R_{v_1}, \ldots, R_{v_n}$ that the sources produce for these functions, (c) the user's preference function f_q and (d) the number N of desired results.

```
Algorithm PipelinedMerge(R_{v_1},...,R_{v_n}, f_q, f_{v_1},...,f_{v_n}, N){
for i = 1 to n do {
  Retrieve first tuple t^{top}_{v_i} from R_{v_i} and compute f_q(t^{top}_{v_i})
  }
Let t^{top} be the tuple t^{top}_{v_i} that has the maximum f_q(t^{top}_{v_i})
while (less than N tuples in the output) {
  for i = 1 to n do {
    T^{top}_{v_i,q} = DetermineWatermark(t^{top}, v_i)
    Scan R_{v_i} and determine the first tuple t_w with f_{v_i}(t_w) < T^{top}_{v_i,q}
    Add all non-processed tuples up to t_w to temporary relation WINDOW
  }
  Sort WINDOW by f_q and let s be the index of t^{top} in WINDOW
  Output and delete the first s tuples from WINDOW
  if WINDOW is empty then
    Add to WINDOW the first unprocessed tuple from each source
    Sort WINDOW by f_q
  Let t^{top} be the first tuple in WINDOW.
  }
}
```

Fig. 14 Algorithm to output the first N tuples according to q

Theorem 8 proves the correctness of the $PipelinedMerge$ algorithm.

Theorem 8 (Correctness) *Algorithm PipelinedMerge outputs the correct ranked results.*

Proof: First we prove that the top tuple according to f_q of a single source S_i is contained in the window of tuples from R_i ending at the last tuple that has a smaller or equal f_{v_i} value to the watermark value $T^{top}_{v_i,q}$. From the definition of the watermark vector we see that all tuples t below the watermark value have $f_q(t) < f_q(t^{top})$. That is, none of these tuples could be the top one according to f_q. Hence the top tuple according to f_q is in the retrieved window.

So in each iteration of the $PipelinedMerge$ algorithm (each iteration of the while loop) the top tuple according to q from each source is contained in the tuples that are added to WINDOW. Hence the top tuple according to f_q over the union of all relations will be in WINDOW and will be output. □

tuple	A1	A2	A3	$f_{v_1}(t)$	$f_q(t)$
t_2	10	17	20	16.8	17.2
t_1	20	20	11	16.4	17.3
t_6	15	10	5	9	9
t_7	12	5	5	6.4	5.7

(a) R_{v_1}

tuple	A1	A2	A3	$f_{v_2}(t)$	$f_q(t)$
t_3	17	18	12	15	16.1
t_4	5	10	12	11	10.1
t_5	15	10	8	9	9.9

(b) R_{v_2}

Fig. 15 Sources S_1, S_2 and scores of each tuple based on f_{v_1}, f_{v_2} and f_q

Example Let us present an example of the algorithm's operation. Assume q is a query with $\mathbf{q} = (0.1, 0.6, 0.3)$ and there are two sources S_1 and S_2 that produce the result sequences R_{v_1} and R_{v_2} respectively. Their preference vectors are $\mathbf{v_1} = (0.2, 0.4, 0.4)$ and $\mathbf{v_2} = (0, 0.5, 0.5)$ respectively. The query and the sources have linear preference functions. Let $m_1 = m_2 = m_3 = 5$ and $M_1 = M_2 = M_3 = 20$. The sequences R_{v_1} and R_{v_2} are shown in Figure 15.

First we retrieve the first tuple from each sequence and find the one that has the maximum f_q value. It is $f_q(t_2) > f_q(t_3)$, so $t^{top} \equiv t_2$. We calculate the watermark vector $(T^{top}_{v_1,q} = T^{top}_{v_2,q}) = (14.26, 15.33)$. The calculation of the watermark is described in Section 4.2. Hence t_2 and t_1 are added to WINDOW because $f_{v_1}(t_1) > T^{top}_{v_1,q}$. We sort the WINDOW by f_q and output t_1 and t_2, which have bigger or equal f_q values than t^{top}. Next we add two fresh tuples t_6 and t_3, because WINDOW is empty, and t_3 becomes t^{top}. We calculate the watermark vector $(T^{top}_{v_1,q} = T^{top}_{v_2,q}) = (13.1, 13.5)$. So no tuples are added to WINDOW and t_3 is output. Now $t^{top} \equiv t_6$. The algorithm continues and outputs t_4, t_5, t_6 and t_7.

6.1 Speculative version of MERGE

As we explain in Section 4.4, many of the applications that we deal with, have an inherent approximation factor. For these applications, it makes sense to sacrifice some of the accuracy of the resulting sequence, in order to improve the performance in terms of the response time. In the case of MERGE, where multiple sources are queried, there is an additional reason why speculation is useful. Suppose that a source S_j is considerably slower than the other sources. Then, S_j becomes the bottleneck in the $PipelinedMerge$ algorithm. To tackle this problem, we assign to each source S_i a speculation factor ε_i proportional to its response time. Hence the sources with longer response times have a higher *threshold value* T'^{top}_i, where

$$T'^{top}_i = (1 + \varepsilon_i) \cdot T^{top}_{v_i,q} \tag{30}$$

The speculative $PipelinedMerge$ algorithm differs from $PipelinedMerge$ in that it uses T'^{top}_i instead of $T^{top}_{v_i,q}$. We evaluate the impact of speculation to MERGE in Section 7.

7 Experimental Results

To evaluate PREFER's and MERGE's algorithms for the efficient execution of preference queries, we carried a detailed performance evaluation. First we compare PREFER's execution time with the time required by a commercial database management system to complete the same task. Then we measure the running time of PREFER's preprocessing step, where the materialized view selection is performed. Then we evaluate PREFER's query performance as different parameters vary. A key *query performance* metric is 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 PREFER with other proposed state-of-the-art solutions. We also measure the performance boost we get when we decrease the depth of the materialized views. Notice that all accesses to the materialized views in PREFER are sequential, which makes it superior to any index-based system which performs random accesses.

For the MERGE system, we measured the average prefix of the source relations that we need to retrieve as the number of sources and the distance between the sources' and the query's functions varies. Finally we evaluated the application of speculation to MERGE.

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: HOUSEID, PRICE, BEDROOMS, BATHROOMS, SQ_FT and YEAR. We performed experiments that

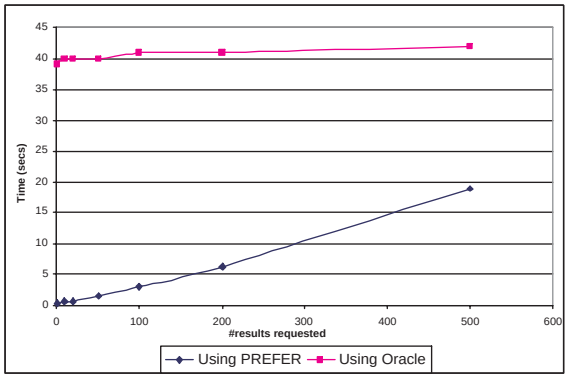


Fig. 16 Execution Times

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 [1] (we did not use these datasets because they were relatively small in size). The correlation coefficient between BEDROOMS and the rest of the attributes (except for YEAR), 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 significant (> 200) number of possible preference vectors and stress the view selection algorithm. Linear functions were used in all experiments.

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. Both PREFER and MERGE are implemented in Java. The two computers were connected through a 10Mbps LAN.

PREFER's 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. We use the top-N hint available in Oracle, although we found that it does not considerably improve the performance. For example, we measured that an order by query with the top-100 hint, executes just 3% faster than the same query with no hint. 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 Figure 16.

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. Also notice that alternative straightforward algorithms could be included in the comparison, which however are clearly inferior to PREFER, since they have to scan the whole relation. One such example is to scan the relation and keep a priority queue with the top-N results so far.

PREFER's 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 17 presents the running time of the algorithm for various parameters of interest, namely the number of attributes in the underlying dataset, the discretization of the domain of preference vectors and the number of result tuples (1 or 10) that we require guarantees for,

attributes	*Top-1 tuple*	
	Discretization 0.1	*Discretization 0.05*
3	39sec — 6views	40sec — 6views
4	136.5sec — 21views	149.5sec —23views
5	377sec — 58views	396.5sec —61vews

attributes	*Top-10 tuples*	
	Discretization 0.1	*Discretization 0.05*
3	43sec —6views	44sec — 6views
4	141sec — 22views	155sec — 25views
5	384sec — 60views	404sec — 64views

Fig. 17 View Selection Algorithm Running Time

on a 50K tuple database. The guarantee provided is that the size of the view prefix is less than 500 tuples. The times in the figure include the time to build the selected materialized views[7], 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.

PREFER's Query Performance as function of the Dataset size. Figure 18 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 PREFER scales gracefully with the dataset size. For the case of correlated data (Figure 18(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 almost tripling the number of materialized views to cover 100% of the queries. Notice that only ten views are enough to cover 90% of the query space for a dataset with 10,000 correlated tuples (Figure 18(a)).

The smaller slopes of the curves for increasing number of tuples is due to the skew. In particular, since the distribution of tuple values is skewed, the distribution of scores in each view is skewed as well. 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, a smaller fraction of the query space is covered when the number of tuples of the database increases.

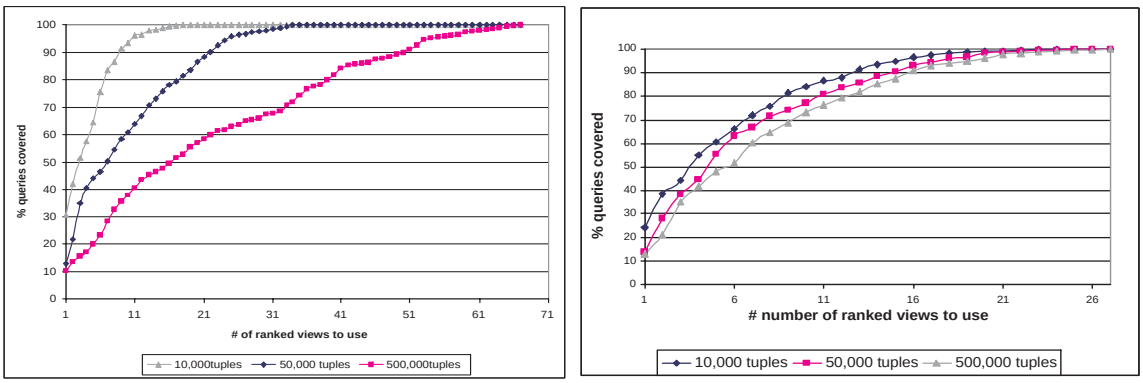
Figure 18(b) presents the results of the same experiment on the random dataset. In the case of random data (uniformly distributed attribute values) the number of additional views required to assure that all queries provide guarantees appears to grow very slowly with database size.

The difference in the fraction of space covered with the same number of views does not vary a lot as the number of tuples increases. This happens because we are dealing with uniform data which means that the values of a preference function f_q are sparse in the region where f_q takes its maximum values. Hence, the fraction of tuples greater than a specific watermark value essentially remains constant (for a truly uniform distribution).

In Figure 18 for a fixed number of views, we often miss the guarantee, because a portion of the query space remains uncovered as a consequence of the imposed constraint on the number of views.

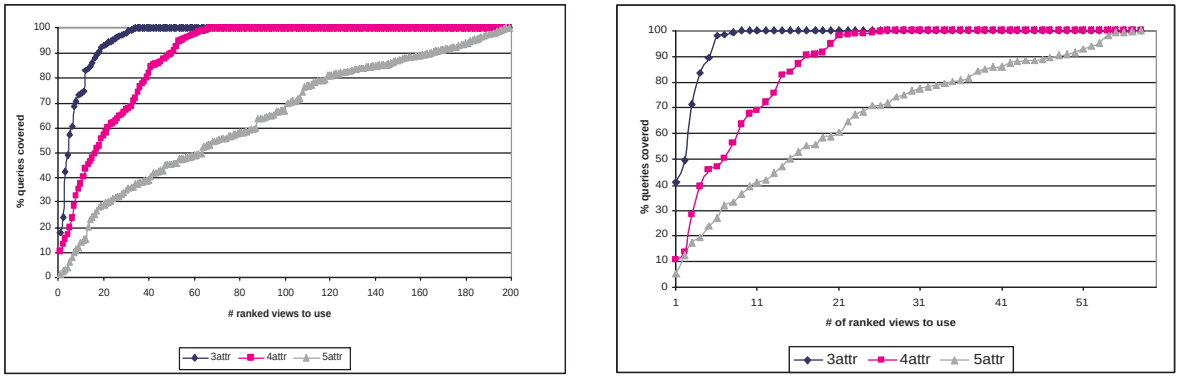
Varying the number of attributes in PREFER. Figure 19 presents the results of an experiment assessing the scalability of the view selection algorithm with respect to the number of attributes in the underlying dataset,

[7] We store the 500-th tuple of every view that we build and build the views directly from a native database interface and not through JDBC. Hence the overall performance is better than [21].



(a) Correlated dataset (b) Random dataset

Fig. 18 Varying the dataset size in PREFER



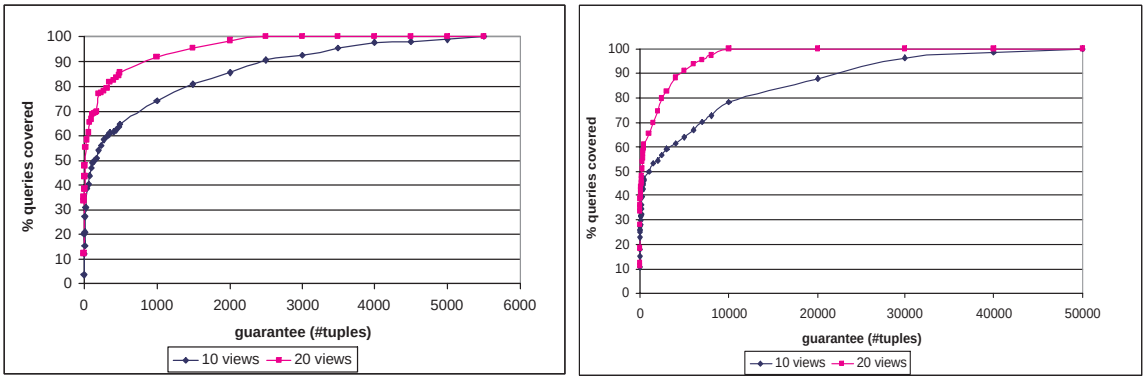
(a) Correlated dataset (b) Random dataset

Fig. 19 Varying the number of attributes

which has 500,000 tuples. Figure 19(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. This explains the different slopes of the curves as the number of attributes increases. 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 coverage areas. Contrasting with figure 19(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 preference attribute space is covered for the same number of materialized views.

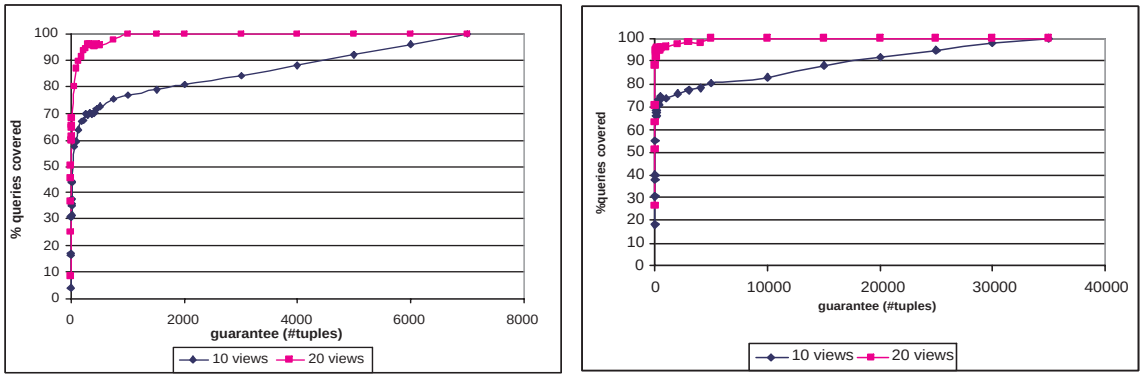
PREFER's query performance as a function of required guarantees. Figure 20 presents the results of an experiment assessing the query performance of PREFER as a function of the guarantees requested. We use four-attribute datasets in this experiment. We vary the guarantees provided by the queries, by increasing the maximum number of view tuples read to report the first result of queries. Figures 20(a)(b) show the results for the correlated dataset for two dataset sizes, and Figures 20(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 20(c)(d)) for the same dataset size the fraction of queries providing a specific guarantee is higher than in the case of correlated data.



(a) Correlated dataset, 50K tuples

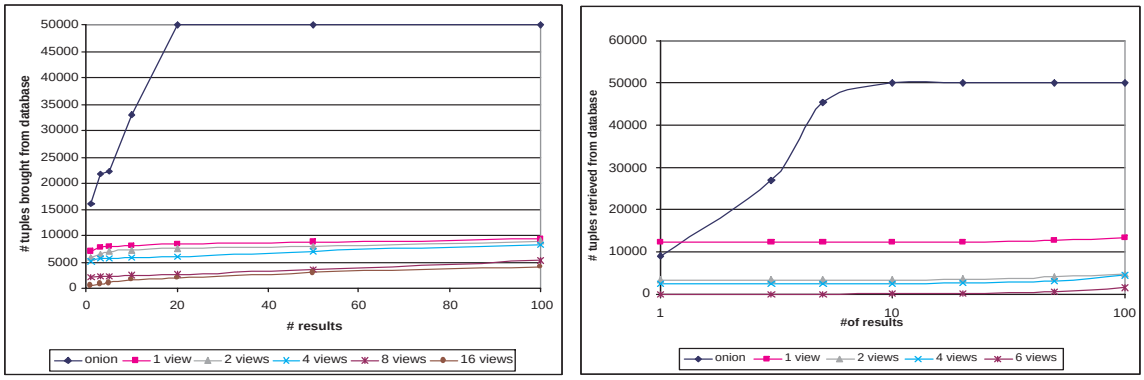
(b) Correlated dataset, 500K tuples



(c) Random dataset, 50K tuples

(d) Random dataset, 500K tuples

Fig. 20 Varying guarantees

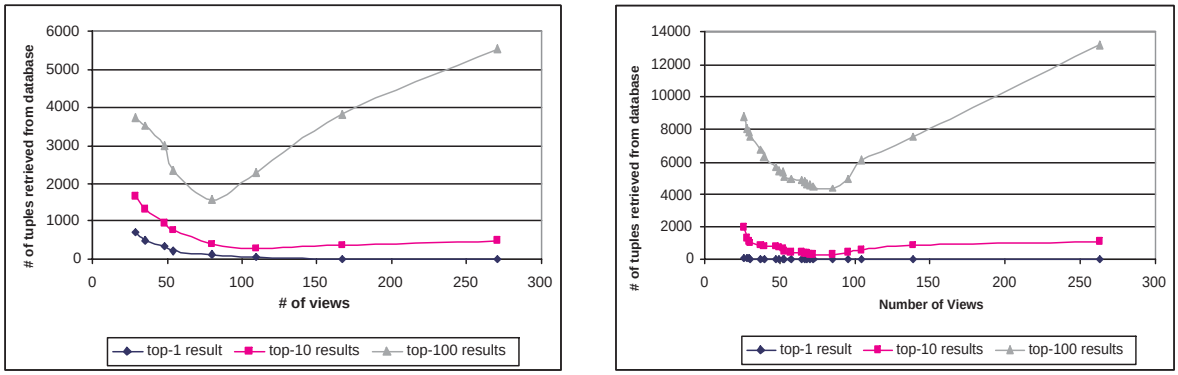


(a) Correlated dataset

(b) Random dataset

Fig. 21 Comparison with the Onion Technique

Comparison of PREFER with the Onion technique. Figure 21 presents an experimental comparison or PREFER against the Onion technique, which was briefly described in Section 2. The Onion technique retrieves in the worst case (which is the average case as well in our experiments) z convex hulls to answer a top-z query. 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.



(a) Correlated dataset (b) Random dataset

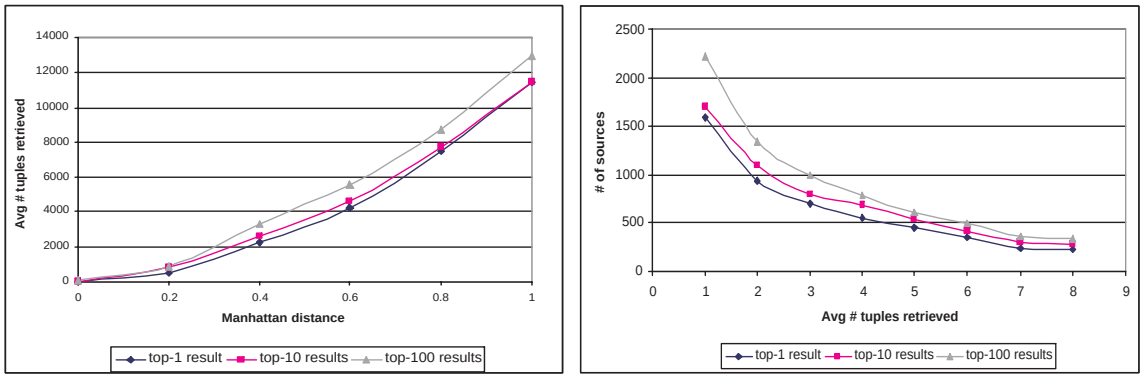
Fig. 22 Varying the number of views and the depth

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. Figure 21(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. The number of convex hulls decreases when the dataset has attributes with small cardinalities. 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 21(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 were 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.

Decreasing the depth of the materialized views in PREFER. We performed a series of experiments to evaluate what impact the decrease of the depth of the views has on the performance of PREFER. We used two synthetic datasets of 50,000 tuples each, the one with independent and the other with correlated attribute values. Four attributes were used and the views and queries discretization was 0.05. We assumed that the space that is available is 10 times the size of the relation. That is, if we construct m views, each of them will have depth $\frac{10 \cdot 50{,}000}{m}$. We increased the number of views and measured the average number of tuples that we need to retrieve from the database to output the top-1, top-10 and top-100 results. The average was calculated over all possible queries with discretization 0.05.

The resulting graphs for the two datasets are shown in Figure 22. These graphs show the trade-off between having a big number of views and having deep views. We observe that for the top-1 result the average number of tuples retrieved is decreasing as the number of views increases and reaches 2, which is the absolute minimum, at 260 to 270 views. This happens because the number of tuples that we need to retrieve to get the first result is generally very small so the depth of the views does not matter so much and when the number of views increases there is always a view very close to the query. The absolute minimum number of tuples that we need to retrieve in order to output the top result is 2, because we always have to read the top tuple of R_v to calculate the watermark and the second tuple of R_v to check if its f_v score is smaller than the watermark value.

On the other hand, notice that for the top-10 and the top-100 results the average number of tuples retrieved is initially decreasing with the number of views until we reach 85 views, where the minimum number of tuples are retrieved and then it increases again. This happens because when the number of views increases beyond



(a) Varying Manhattan distance (b) Varying number of sources

Fig. 23 Performance of MERGE

a certain point (85 for this dataset) the depth decreases so much that a significant number of queries need to query the original relation R in order to retrieve the top-N results. Thus they retrieve all 50,000 tuples and affect the average significantly.

Evaluation of MERGE. The performance of the MERGE system depends on the number of sources and on the similarity of the function f_q, which is used by the user query, to the functions $f_{v_1}, \ldots, f_{v_n}$ supported by the sources. Every source's relation has 50,000 tuples and is generated synthetically as described above. The attribute values are independent from each other. We use a discretization of 0.05 for the domain from which we draw source and query preference vectors (0 through 1, in increments of 0.05).

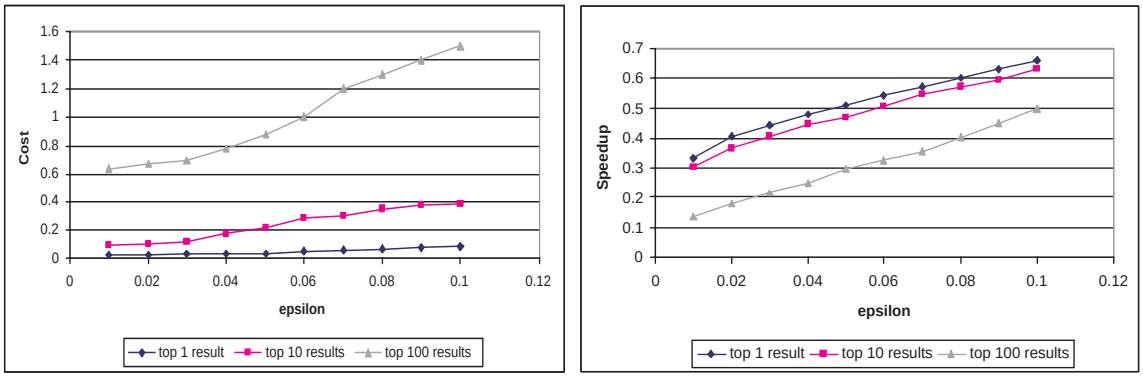
Prefix size for varying distance between metabroker's and sources' queries in MERGE. Our first experiment on MERGE assesses the average prefix size that we need to retrieve from each of the underlying sources in order to present to the user of the metabroker the top 1, 10 and 100 results, as a function of the Manhattan distance MD between the preference vectors of the user query and the source queries ($MD = \sum_{j=1}^{k} |q_j - v_j|$). For simplicity in the report, we assumed the distance of the query from each source is the same. Notice that the Manhattan distance between any two source queries can be at most two times the distance between the metabroker and the source queries. Given the discretization and the fact that we work with four attributes, the maximum Manhattan distance of the query's preference vector from each source's preference vector is 2. We use four sources for this experiment and the resulting graph is shown in Figure 23 (a). We see a slightly superlinear performance deterioration due to the fact that the preference scores have a concentration towards the "average" score.

Prefix size for varying number of sources in MERGE. This experiment measures the average prefix size that we need to retrieve from each of the underlying sources in order to present to the user of the metabroker the top 1, 10 and 100 results, as a function of the number of sources that are used. The Manhattan distance between the metabroker query and the source queries is fixed to 0.2. The result is shown in Figure 23 (b). The average number of tuples retrieved from each source decreases as more sources are used.

Speculative version of MERGE. In this experiment we apply a speculation factor ε to one of the four sources and measure the cost of speculation and the *speedup* that we get. Recall that the cost of speculation is the average difference between the indices of tuples in the speculative results sequence and in the real results sequence. The speedup is the average ratio of the decrease in the number of tuples that we need to retrieve from each source to output the top-N results. In Figure 24 we show how the average cost and speedup vary with the speculative factor, to present to the user of the metabroker the top 1, 10 and 100 results.

8 Implementation of PREFER

The overall system architecture is shown in Figure 11. Using algorithm *ViewSelection* we select a number of views and we materialize them. A relational DBMS is used for storing the views.



(a) Speculative cost (b) Speedup

Fig. 24 Speculation in MERGE

We have developed a Java API to allow programmers to use PREFER's functionality in applications. We have also implemented PREFER as a stand alone application that runs on Microsoft Windows and connects to Oracle database servers via JDBC. PREFER provides to the end users an easy to use interface that automates the process of selecting and materializing the views and querying the database using these views. The application can be divided into two parts. The first part automates the selection and materialization of the views. The user inputs (see Figures 25(a) and 25(b)): (a) the name of a relation R, (b) the list of attributes of R that will be used in the preference functions, (c) the maximum number S of views that will be created, (d) the depth D of the views, (e) a constraint C, that denotes the maximum number of tuples that are retrieved to output the first result, (f) the granularity g_v (discretization) of the views' attribute weights and (g) the granularity g_q (discretization) of the queries' attribute weights.

The construction part of the application:

- Creates at most S non-overlapping views whose weights are multiples of g_v. By non-overlapping we mean that none of the views should be contained in the query space covered by the other views. Notice that the constraint C is needed at this point to define the covered space for each view as described in Section 5.1. Only the top-D tuples of each view are stored. One should give a large S value if covering the whole space with respect to the constraint C and the granularity g_v is important.
- Stores the name and the weights of each view in a new table called INFO table.
- Tests all queries whose weights have granularity g_q and find the view that covers each of them. We create a table called PAIRS and we insert the weights of the query and the "best" view that we found for each tested query.

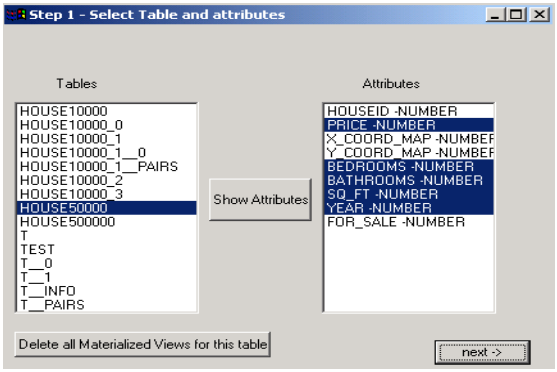
The second part of the application uses the views that were created to efficiently answer to user queries. The user inputs (see Figure 25(c)): (a) the name of a relation R, (b) The preference vector $\mathbf{q}$, which contains the requested weights for each attribute and (c) the number N of results to be output.

The querying part of the application (see Figure 25(d)):

- Looks up the PAIRS table to find the "best" view V for the query. If $\mathbf{q}$ is not in PAIRS table then we select the view from INFO table that is closer to $\mathbf{q}$ with respect to the Manhattan distance.[8]
- Retrieves the preference vector $\mathbf{v}$ for V by looking up INFO table.
- Uses PipelineResults algorithm to output the top-N results according to $\mathbf{q}$.

The specification of the PREFER API and the PREFER application are available at http://www.db.ucsd.edu/PREFER.

[8] Recall that in Section 5.3 we have proposed a heuristic for the case where none of the views covers a query. We pick the view that minimizes the expression $f_v(t_v^l) + \sum_{i=1}^{k}(q_i - v_i)A_i(t_v^l) - f_q(t_v^1)$. We approximate this idea by picking the view that has the smallest Manhattan distance to the query.



(a) Table and attributes selection

(b) Construction parameters selection

(c) Query parameters selection

(d) Query results

Fig. 25 Screenshots of the Application

9 Acknowledgements

We would like to thank the reviewer who pointed out the non-optimality of the *PipelineResults* algorithm, which led us to write the *PipelineResultsOptimal* algorithm. The quality of the paper benefited greatly from the constructive comments of the reviewers and we are very thankful for their efforts.

References

1. http://www.realtor.com.
2. Y. S. A. Levy, A. Mendelzon and D. Srivastava. Answering Queries Using Views. *ACM PODS*, 1995.
3. R. Agrawal and E. Wimmers. A Framework For Expressing and Combining Preferences. *ACM SIGMOD*, 2000.
4. N. Bruno, L. Gravano, and A. Marian. Evaluating Top-k Queries over Web-Accessible Databases. *ICDE*, 2002.
5. J. P. Callan, Z. Lu, and W. B. Croft. Searching distributed collections with inference networks. *SIGIR Conference*, 1995.
6. Y. Chang, L. Bergman, V. Castelli, C. Li, M. L. Lo, and J. Smith. The Onion Technique: Indexing for Linear Optimization Queries. *ACM SIGMOD*, 2000.
7. S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. *ACM SIGMOD*, 1996.
8. S. Cohen, W. Nutt, and A. Serebrenik. Rewriting Aggregate Queries Using Views. *ACM PODS*, 1999.
9. O. Duschka and M. Genesereth. Answering Recursive Queries Using Views. *ACM PODS*, 1997.
10. R. Fagin. Combining Fuzzy Information from Multiple Systems. *PODS*, 1996.
11. R. Fagin. Fuzzy Queries In Multimedia Database Systems. *ACM PODS*, 1998.
12. R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. *ACM PODS*, 2001.
13. R. Fagin and E. Wimmers. Incorporating User Preferences in Multimedia Queries. *ICDT*, 1997.
14. J. Goldstein and R. Ramakrishnan. Contrast Plots and P-Sphere Trees: Space vs. Time in Nearest Neighbour Searches. *VLDB*, 2000.

15. L. Gravano, C. K. Chang, H. G. Molina, and A. Paepcke. STARTS: Stanford proposal for Internet meta-searcing. *ACM SIGMOD*, 1997.
16. L. Gravano and H. G. Molina. Merging Ranks from Heterogeneous Internet Sources. *VLDB*, 1997.
17. U. Guntzer, W. Balke, and W. Kiessling. Optimizing multi-feature queries in image databases. *VLDB*, 2000.
18. U. Guntzer, W. Balke, and W. Kiessling. Towards Efficient Multi-Feature Queries in Heterogeneous Environments. *IEEE Intenational Conference of Information Technology (ITCC)*, 2001.
19. A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal 10(4)*, 2001.
20. D. Hockbaum. *Approximation Algorithms for NP-Hard Problems*. ITP, 1997.
21. V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. *ACM SIGMOD*, 2001.
22. A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. *VLDB*, 1996.
23. S. Nepal and M. Ramakrishna. Query processing issues in image (multimedia) databases. *ICDE*, 1999.
24. C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover.
25. Y. Papakonstantinou and V. Vassalos. Query Rewriting For Semistructured Data. *ACM SIGMOD*, 1999.
26. D. Srivastava, S. Dar, H. V. Jagadish, and A. Levy. Answering Queries with Aggregation Using Views. *VLDB*, 1996.
27. V. Vassalos and Y. Papakonstantinou. Expressive Capabilities, Description Languages and Query Rewriting Algorithms. *JLP 43(1)*, 2000.
28. E. M. Voorhees, N. K. Gupta, and B. Johnson-Laird. The collection fusion problem. *Text Retrieval Conference (TREC-3)*, 1995.