Ordering the Attributes of Query Results

Gautam Das *
University of Texas at Arlington
gdas@cse.uta.edu

Vagelis Hristidis†
Florida International University
vagelis@cis.fiu.edu

Nishant Kapoor‡
University of Florida
nkapoor@cise.ufl.edu

S. Sudarshan§
IIT Bombay
sudarsha@cse.iitb.ac.in

ABSTRACT

There has been a great deal of interest in the past few years on ranking of results of queries on structured databases, including work on probabilistic information retrieval, rank aggregation, and algorithms for merging of ordered lists. In many applications, for example sales of homes, used cars or electronic goods, data items have a very large number of attributes. When displaying a (ranked) list of items to users, only a few attributes can be shown. Traditionally, these are selected manually. We argue that automatic selection of attributes is required to deal with different requirements of different users. We formulate the problem as an optimization problem of choosing the most "useful" set of attributes, that is, the attributes that are most influential in the ranking of the items. We discuss different variants of our notion of attribute usefulness, and propose a hybrid Split-Pane approach that returns a composite of the top attributes of each variant. We conduct both a performance and a user study illustrating the benefits of our algorithms in terms of efficiency and quality of explanation.

1. INTRODUCTION

In recent years, there has been a great deal of interest in developing effective techniques for ad-hoc search and retrieval in structured data repositories such as relational databases. In particular, a large number of emerging applications require exploratory querying on such databases; examples include users wishing to search databases and catalogs of products such as homes, cars, cameras, restaurants, and so on. The following running example is frequently used throughout the paper to illustrate key concepts.

Example 1: *Consider an inventory database of an auto dealer, which contains a single table T with N rows and M attributes where each tuple represents a car for sale. The table has numerous attributes that describe details of the car, such as Price, Make, Model, Age, Zipcode, Mileage, EngineSize, NumCylinders, AccidentHistory, SecuritySystem, AirConditioning, and so on.*

Current database query languages such as SQL follow the Boolean retrieval model, i.e., tuples that exactly satisfy the selection conditions laid out in the query are returned - no more and no less. While extremely useful for the expert user, this retrieval model is inadequate for ad-hoc retrieval by exploratory users who cannot articulate the perfect query for their needs - either their queries are very specific, resulting in no (or too few) answers, or are very broad, resulting in too many answers. In the example above, a simple conjunctive query such as "Select * from T where Model=sedan and Price $\leq$ 16000 and Mileage $\leq$ 20000" may overwhelm the user with too many answers.

To address the limitations of the Boolean retrieval model for such queries, a technique that has received widespread attention in recent years is that of *ranking* of database query results. Ranking systems typically compute a *score* of a tuple which represents the degree to which the tuple is "relevant" for the query, and return a few tuples with high scores (e.g., top-n tuples where n is a small number such as 10 or 100) to the user.

The ranking problem in databases has been extensively investigated in recent years [11, 15, 10, 1, 9, 6]. The various ranking techniques for database queries range from simple scoring functions (e.g., similarity functions based on Euclidean distances with pre-defined weights on attributes), to more sophisticated, query-specific functions developed by domain experts or automatically derived user preferences from previously available workloads (e.g. [1, 6]). These sophisticated approaches hinge on their ability to convert user queries - which typically specify simple selection conditions on a small set of attributes - into comprehensive scoring functions that also involve many other attributes *in addition to* the ones specified in the original query. In other words, these ranking systems "extend" the original query by drawing on available knowledge of previous user preferences for the unspecified attribute values, much as a knowledgeable salesperson may suggest features of potential interest to a customer who has given some initial requirements. In the auto database example above, for users seeking used cars in New York, the ranking system may give higher scores to cars with theft prevention systems, whereas for users seeking used cars in Texas, cars with air conditioning systems may be favored. Sections 2.2 and 5 contains further discussion on database ranking methods.

However, the focus of this paper is not on new ranking models. Rather, the main thrust of this paper is the introduction of an orthogonal problem in ad-hoc exploratory querying of database - that of *selecting the top-m attributes* of query results (where $m << M$).

*Partly supported by a grant from Microsoft Research.
†Partly supported by NSF grant IIS-0534530.
‡Part of work performed while a graduate student at the University of Texas at Arlington.
§Part of work performed while visiting Microsoft Research.

SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.

This problem is motivated by the fact that many domains of interest have a very large number of attributes: for example, the number of attributes in typical automotive and home databases ranges from 25 to a hundred or more, as can be easily verified from car sales or homes web sites. With such a large number of attributes, it is usually not possible to display all attributes for the top-n answers to a query. Tabular displays of answers on web sites therefore typically display only a few attributes that are considered to be most "useful" to the user. However, the decision on what attributes to display is usually made manually, and fixed for all users, regardless of what attributes are likely to be useful to a particular user. Consequently, the answer tuples to a query may get displayed with a fixed set of attributes that do not reveal important relevant details, leading to a less-than-satisfying experience for the user. We thus pose the following problem:

The Attribute Selection Problem Intuitively, the attribute selection problem can be defined as follows. *Given a query and a ranking function for query results, return the top-m most useful attributes of the answer tuples, where m is a small number (e.g., $m \leq 15$).*

Of course, any principled approach for solving this problem will have to make reasonable assumptions about what it means for an attribute to be useful to a user. In this paper we focus on one primary notion of attribute usefulness, which may be intuitively described as: *an attribute is considered useful if it plays an influential role in the computation of the top-n ranked answer tuples of a query*. In other words, we seek to determine those attributes that best "explain" the top-n tuples returned by the ranking function. Intuitively, these would correspond to the attributes that an expert would find useful in ranking tuples. Although a naive user may not have thought of specifying such attributes in the initial query, the user is likely to find the attributes useful after seeing them in the result.

In the auto example above, for New York customers the top attributes may include SecuritySystem, whereas for Texas customers the top attributes may include AirConditioning. We believe that this notion of attribute usefulness is simple, intuitive, and covers many important scenarios. Continuing to develop other notions of attribute usefulness - e.g., scenarios where attributes may be useful to users, yet have nothing to do with how the top tuples are selected - is an intriguing research area that is left for future work (we discuss this issue briefly again in Section 2).

Our notion of attribute usefulness is quite broad, and several interesting variants can be developed, each variant conveying different types of information. We discuss these variants below, focusing on their differences at an intuitive level - more precise definitions are presented in Section 2.

1. *Score-Based Attribute Selection:* In this variant of attribute selection, we are interested in determining the top-m attributes that have the most influence in the computed *scores* of the top-n returned tuples of a query. To motivate this, consider a query which seeks late model two-door sports cars. A reasonable ranking function for such cars may be one where Price and EngineSize have the largest influences in computing their scores. The task is thus to determine and return these attributes.
2. *Rank-Based Attribute Selection:* This variant of attribute selection is subtly different from the previous variant. Here we are interested in determining the top-m attributes that are most influential in ranking the top-n tuples higher than the rest of the tuples also satisfying the query conditions. If we consider the above query again, although Price and EngineSize may influence the scores the most, it may happen that *most tuples*, even those outside the top-n, have similar prices and engine sizes. Thus other attributes (such as Make or Age) may have been critical in differentiating between the top-n tuples and the rest.
3. *Relative Rank-Based Attribute Selection:* In this variant of attribute selection, we wish to know the top-m attributes that are most influential in preserving the relative rankings of the returned top-n tuples amongst each other. Note that this third variant is quite different from the second - there we wished to know which attributes separated the top-n tuples from the rest of the tuples, whereas here we wish to know which attributes are most responsible for determining the particular permutation in which the top-n tuples are returned. For example, the relative ranks of the top-n tuples may most closely match the relative ranks of the corresponding Mileage and NumCylinders attribute values of these tuples.

The returned top attributes for each of the above variants convey different types of information, each of which has a role to play in helping the users compare the ranked answers to a query, and in understanding why these answers were returned. We therefore propose a hybrid approach, which we call the *Split-Pane* approach, which returns and displays the top attributes from each of these variants side-by-side. We show experimentally that this approach is clearly more intuitive to users than the individual constituent approaches underlying Split-Pane, as well as other baseline attribute-selection techniques considered.

Technical Challenges The key technical challenge is to automatically come up with a selection of the top-m attributes that best explain the score, rank or relative rank of tuples, based on the scoring/ranking function, without any user intervention. In particular, our user study shows that attribute selections must be based on the query, and a pre-defined query-independent selection has limited performance; instead, the attribute selections must be computed automatically based on the scores/ranks of the query results.

The score-based attribute selection is perhaps the simplest of the above variants in terms of algorithmic complexity, because if the scoring function is known in advance, determining the influential attributes is relatively straightforward - however, this problem becomes more interesting if we are not privy to the innards of the actual scoring function used, i.e., if the scoring function is viewed as a black box. The other variants are more involved, because we have to determine useful attributes by analyzing how they affect the final rankings of the result tuples. One way of measuring the effect of a set A of attributes on the rank (or relative rank) is to consider the rankings that would be produced if only attributes in A were used. We make this notion more precise later, and show that the selection of a set of attributes that minimizes the rank error is NP-hard for both the score- and the rank-based variants. We present algorithms for these variants, based on a combination of greedy heuristics as well as non-trivial partial computations of the attribute selections.

The *Attribute Ordering* problem has the additional requirement of ordering the (selected) attributes in decreasing order of their usefulness. Such an ordering can be quite useful: for example, in designing a tabular interface, where it may be preferable to place the attributes from left to right in decreasing order of usefulness. We use the term *vertical ranking* to denote the order in which tuples appear, and the term *horizontal ordering* to denote the ordering of attributes.

Defining an attribute ordering is complicated by the fact that the usefulness of an attribute (i.e., its influence on the vertical ranking) is affected by what other attributes have been selected. We present

two greedy heuristics, based on two different ways of defining the usefulness of an attribute, either ignoring or taking into account the choice of other attributes. These greedy heuristics can also be used for the attribute selection problem by selecting the top m attributes from the attribute orderings they generate. We emphasize, however, that the technical focus of this paper lies in the area of attribute selection, with attribute ordering being an extension of our core results on attribute selection.

Contributions We summarize the main contributions of our paper as follows

1. We introduce *attribute selection/ordering* as a new database retrieval model that complements the tuple ranking model, yet is orthogonal to the specifics of the ranking function used.
2. We present several interesting variants of the attribute selection problem, where each variant conveys different information to the user. We also present the *Split-Pane* approach, which combines the top attributes from each of these variants.
3. We analyze the computational complexity of the different variants of the attribute selection problem, and show that most of them are NP-complete.
4. We present algorithms for the different variants of attribute selection, based on a combination of greedy heuristics as well as non-trivial partial computations of attribute selections. In some instances we prove that our algorithms are optimal.
5. We performed user studies and detailed performance evaluations to show that the attribute selection problem is useful, and can be efficiently solved by the Split-Pane approach.

Related Work The most closely related works to ours are in the areas of ranking of query results, feature selection, and data visualization. We describe related work in detail in Section 5, but outline key differences here. Work in the area of ranking of query results (such as [11, 15, 10, 1, 9, 6]) do not consider the problem of ordering/choosing important attributes from a large set of attributes. Although the problem of choosing attributes is related to the area of feature selection ([14]), our work differs from the extensive body of work on feature selection in several ways: (1) our goal is to explain the ranking of results to end users, not to reduce the cost of building a mining model such as classification or clustering, and (2) our approaches measure the amount by which the *ranks* of the top-n tuples changes when a subset of attributes are used in score computations, whereas feature selection approaches are not based on changes in ranks. Data visualization systems form another category of related work, but to our knowledge none of the related work in this area (e.g. [16, 20, 2]) has considered the problem of choosing what attributes to display that influence the rankings of results. See Section 5 for more details on related work.

The rest of this paper is organized as follows. Section 2 describes the problem framework and notation. Section 3 presents results on complexity, while greedy algorithms are presented in Section 4. Section 5 describes related work. Section 6 presents quality and performance studies, and Section 7 concludes the paper.

2. SYSTEM FRAMEWORK

Our dataset consists of a relation (or view) R which has N tuples $t_1, \ldots, t_N$ and M attributes $\mathcal{A} = a_1, \ldots, a_M$. Let $t[a_i]$ denote the value of tuple t for attribute a_i. Let query Q specify conditions on some subset of attributes A_Q (we shall frequently use upper-case alphabets such as A and A_Q to refer to subsets of the attributes set $\mathcal{A}$).

In the following subsection we first review various tuple ranking functions. Later we formally introduce the attribute selection problem, which is the main focus of this paper.

2.1 Review of Tuple Ranking Functions

Most common methods to rank the result tuples of query Q are centered around developing a scoring function $S(Q, t)$ that assigns relevance scores to tuple t. A few (top-n) of the most relevant tuples are then returned in decreasing order of relevance score to the user. Numerous scoring functions have been developed in the literature, ranging from Euclidean distance functions (primarily for numeric data), cosine similarity functions, as well as complex functions based on probabilistic retrieval models. We briefly discuss some common scoring functions here; also see Section 5 and the references therein. Some of these scoring functions only operate on the attributes specified by the query - i.e., on a projection over the A_Q columns - while others try and extend the query with additional query conditions such that all attributes are specified, yet others attempt to directly seek out correlations between the query values and the unspecified attribute values of the tuple.

In solving our problem, it is often necessary for us to determine the "contribution" of certain attributes in scoring. Consequently, we assume scoring functions to be of the form $S(Q, A, t)$, where A is an optional set of attributes that specifies which attributes are to be utilized when computing the score of the tuple - i.e., the remaining attributes are "masked out", or ignored from the scoring calculations. (The set A is not to be confused with the set A_Q; the latter is the subset of attributes specified by the query). The ability to mask certain attributes is a reasonable requirement on scoring functions, and as we shall see below, most common scoring functions possess this property. Moreover, attributes masking is used for the primary purpose of solving our attribute selection problem, and is not used in the tuple ranking problem.

A simple class of scoring functions is the *additive* scoring function, defined as follows. Let weights w_i be associated with each attribute a_i, and let function $f_i(x, y)$ return a value in $[0..1]$ indicating how related value x is to y for values in the domain of attribute a_i. Let Q specify equality conditions on some set of attributes A_Q, and let the value specified for any $a_i \in A_Q$ be v_i. Then

$$S(Q, A, t) = \sum_{a_i \in A_Q \cap A} (w_i * f_i(t[a_i], v_i)) \quad (1)$$

An example of an additive scoring (used in our experiments) is a weighted version of the Euclidean function (henceforth referred to as *WeightedL2*), defined as follows. For categorical attributes $f_i(x, y) = 1$ if $x = y$ (or $x \in y$ if we allow y to be a set of values), and is 0 otherwise. For numerical attributes $f_i = 1 - (n_x - n_y)^2$, where n_x and n_y are the normalized[1] values of x and y in the range $[0..1]$. Such scoring functions are usually defined by domain experts, although in certain applications the user may specify some of the weights in addition to some attribute values. Some applications may even automatically extend a user query so that all attributes get assigned weights as well as query values.

Another common example of additive scoring functions is the *cosine similarity* function (used in [1] for ranking database query results) which computes the normalized dot product between the query and the tuple values. In certain applications (e.g., keyword

[1]We normalize using the minimum and maximum values for the domain of a_i; other standard normalization techniques are also possible.

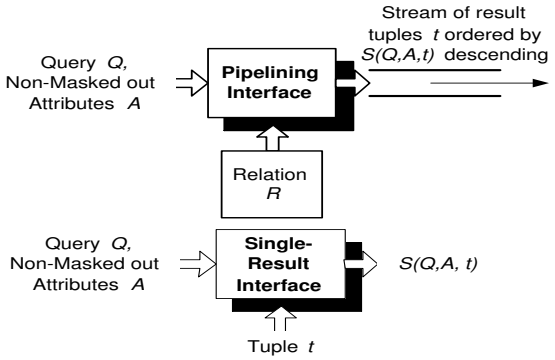

Figure 1: Execution Interfaces.

queries in [3]), scoring functions are *multiplicative*, i.e., the final score is a product of the contributions of the scores of each attribute. Multiplicative scoring functions can be converted to additive functions by viewing the scores on a log-scale.

A *general* scoring function is a more complex function in which terms cannot be separated according to the attribute they refer to. An example of such a scoring function, which is used in our experiments, is the *Conditional* scoring function based on probabilistic information retrieval models (defined in Chaudhuri et al. [6])

$$S(Q,A,t) = \prod_{a_j \in C} p(t[a_j]) * \prod_{a_j \in C, a_i \in A_Q \cap A} p(v_i, t[a_j]) \quad (2)$$

where $C = \mathcal{A} - (A_Q \cup A)$ and $p()$ is a function defining the popularity (for one argument) and correlation (for two arguments) of values in the query workload and database. More details on the Conditional scoring function can be found in [6].

Execution Model for Tuple Ranking Functions Although we assume that we are provided with a black box vertical scoring function $S(Q,A,t)$, the way such a scoring function is implemented greatly affects the performance of our attribute selection algorithms, since it determines which algorithms are feasible and efficient. We define two interfaces (Figure 1) for the vertical ranking black box, which are natural and supported by previous works like Fagin et al. [11], Chaudhuri et al. [6], the PREFER system [15] and Bruno et al. [4].

In particular, the *pipelining interface* $S(Q,A)$ inputs the query Q, a set A of attributes to be utilized (i.e., not masked out) and a relation R and outputs a stream of tuples $t \in R$ ranked descending according to $S(Q,A,t)$ along with their scores (scores are not needed in some cases). The cost incurred in using this interface is the number of tuples retrieved (we can stop retrieving tuples at any time). The *single-result interface* $S(Q,A,t)$ inputs Q, A and a tuple t and outputs the score $S(Q,A,t)$. This interface incurs unit cost.

2.2 The Attribute Selection Problem

As we have emphasized in the introduction, our focus is on the *attribute selection problem*. Given a relation R, a scoring function $S(Q,A,t)$, a number n of requested result-tuples, and a number m of requested attributes, we are concerned with the problem of selecting a subset A of m attributes that were the "most influential" in the computation of the top-n ranked answer tuples. A design goal for our techniques is that they should be able to work with any scoring function. We now formally define the three problem variants mentioned in the introduction.

1. Score-Based Attribute Selection: In this version of the problem, we are interested in determining the top-m attributes that have the most influence in the computed *scores* of the top-n returned tuples of a query. Let *Topn* be the set of top-n tuples returned by the query. Let A be any subset of m attributes. The *score distance*, $SDist(\mathcal{A}, A)$ is defined as

$$\textit{SDist}(\mathcal{A}, A) = \sum_{t \in \textit{Topn}} (S(Q,\mathcal{A},t) - S(Q,A,t))^2 \quad (3)$$

Intuitively, $SDist(A, \mathcal{A})$ measures how close the scores of the top tuples are to the true scores if only a subset of attributes A were involved in the calculations. The objective is to determine the subset A that minimizes $SDist$. Although this problem variant is simpler than the others, we argue that it is not trivial. For example, even in the case of the additive scoring model where the weights may be known beforehand, we cannot pre-determine A by simply taking the m attributes with the largest weights. This is because as Equation 1 shows, the contribution of the ith attribute to the score of a tuple t is $w_i * f_i(t[a_i], v_i)$, and the second factor can only be determined at runtime. For example, the weight of the Price attribute may be more than the weight of Mileage, however, for a specific query the top tuples returned may contain cars whose prices differ greatly from the desired price, but whose mileages are very close to the desired mileage (see Section 6 for an experimental comparison of score-based versus simpler weight-based attribute selection approaches). This problem is exacerbated in the case where weights are not known beforehand and where scoring schemes assign "query specific" weights to unspecified attributes (e.g., for users seeking late model sports cars, EngineSize may get higher weight than for users seeking economy cars). Likewise, this problem is more involved for more general scoring functions where the contributions of each attribute cannot be easily separated from the others.

2. Rank-Based Attribute Selection: In this version of the problem, we are interested in determining the top-m attributes that are most influential in *ranking* the top-n tuples higher than the rest of the tuples. Let L be the ranked list (or permutation) of all tuples when ranked by the scoring function with all attributes contributing to the scoring - i.e., when the database is ranked by $S(Q,\mathcal{A},t)$. Let L_A be the ranked list when only a subset A of m attributes contribute to the scoring - i.e., when the database is ranked by $S(Q,A,t)$. The *rank distance*, $RDist(\mathcal{A}, A)$ is defined as

$$\textit{RDist}(\mathcal{A}, A) = \sum_{t \in \textit{Topn}} |L(t) - L_A(t)| \quad (4)$$

where $L(t)$ denotes the position (or rank) of element t in L.

Intuitively, L is the "ground truth" ranking, and $RDist(A, \mathcal{A})$ measures how closely L_A approximates L with respect to the top-n tuples. Note that this measure of comparing ranked lists is based on Spearman's *footrule formula* for comparing ranked lists [7], the only difference being it only considers the *Topn* subset of the elements of the two permutations.[2] The objective is to determine the subset A that minimizes $SDist$.

This version of the problem is subtly different from the score-based version. For example, if we consider a user seeking late model sports cars, although Price and EngineSize may influence the scores the most, it may happen that *most tuples* that satisfy the

[2]We could have used other metrics such as *Kendall Tau*, but Spearman's footrule is more appropriate for our more problem because the ranks of subsets of the tuples are used in comparing rankings. Our metrics can also be extended using the partial ranking metrics defined by Fagin et al [9] in case the scoring function produces many ties. Likewise, our metrics can also be extended to give more importance to tuples at higher positions in the lists (as in [21]).

query condition, even those outside the top-n, have similar prices and engine sizes. Thus other attributes (such as Make or Age) may have been critical in differentiating between the top-n tuples and the rest.

This problem is also related to the *feature selection* problem in classification algorithms in machine learning - the similarities and differences are discussed in more detail in Section 5 (also see our experiments in Section 6).

3. Relative-Rank Based Attribute Selection: In this problem, we wish to know the top-m attributes that are most influential in preserving the relative rankings of the returned top-n tuples amongst each other. The formal specification of the problem is very similar to the second problem variant - except that we replace the entire database with the top-n tuples only. Let L' be the corresponding ranked list for the set *Topn*, i.e., when the tuples of *Topn* are ranked by the scoring function with all attributes contributing to the scoring. Let L'_A be the ranked list (permutation of L') when only a subset A of the m attributes contributes to the scoring - i.e., when the database is ranked by $S(Q, A, t)$. The *relative-rank distance*, $RRDist(\mathcal{A}, A)$ is defined as

$$\textit{RRDist}(\mathcal{A}, A) = \sum_{t \in \textit{Topn}} |L'(t) - L'_A(t)| \quad (5)$$

where $L'(t)$ denotes the position (or rank) of element t in L'.

The semantics of this problem variant are quite different from the second variant - there we wished to know which attributes separated the top-n tuples from the rest of the tuples, whereas here we wish to know which attributes are most responsible for determining the particular permutation in which the top-n tuples are returned. Thus, these attributes serve to more carefully differentiate among the returned tuples than any of the previous problems. For example, the relative ranks of the top-n tuples may most closely match the relative ranks of the corresponding Mileage and NumCylinders attribute values.

Discussion While the above problem variants are the main focus of this paper, we take the opportunity to briefly discuss several other interesting variations which are part of our ongoing work but we do not describe here due to lack of space. One such variant is the *Per-Tuple Attribute Selection problem*, where instead of being restricted to choosing the same set of m attributes for all the top-n returned tuples, we can choose to display attributes on a per-tuple basis - that is, different attributes can be chosen for each returned tuple. This allows a more detailed look into each returned tuple than would otherwise be achievable with the previous variants. For example, fuel efficiency may not be considered useful enough to display in general, but it may be worth mentioning for a car with exceptionally good fuel efficiency.

In fact, the per-tuple attribute selection problem has an interesting sub-variant - determining *bad* as well as *good* attributes. Intuitively, an attribute for a tuple is bad (respectively good) if, without its influence, the tuple's ranking improves (respectively degrades). Knowledge of such attributes provides more fine-grained understanding of the reasons behind a specific tuple's final rank that would otherwise be overlooked by the more aggregated variants, e.g., it is useful to know that a specific car has poor gas mileage, even if it is exemplary in other respects.

The *Multiple Queries Attribute Selection* problem is yet another variant where, given a set of queries - such as a representative query workload indicative of the type of users of the system - the task is to choose the top-m attributes that are *collectively* the most influential in rankings the results of these queries. As an example, consider an auto dealer preparing an advertisement listing a number of cars in a tabular fashion, highlighting the attributes that the dealer feels would be most important for the typical customer.

Finally, we mention that in some applications users may find certain attributes to be useful even if they have *very little* to do with the ranking of tuples. E.g., users may be always interested in seeing attributes of products such as Vehicle Identification Number (VIN) and Price - perhaps for looking up more information from an external database - whereas in most applications ranking functions ignore serial numbers, and in some applications price may not be an influential attribute in ranking. Continuing to develop this and other notions of attribute usefulness is an intriguing research area that is left for future work.

3. COMPLEXITY RESULTS

In this section we analyze the computational complexity of the variants of the attribute selection problem. We show that some of the variants are NP-Complete, and outline several greedy heuristics that are shown to be optimal in certain cases.

3.1 Rank-Based Attribute Selection

In this subsection we shall show that the rank-based attribute selection problem is intractable, thus necessitating the development of heuristic and approximation methods for finding the best m attributes. To prove intractability, we shall consider a simplified version of our problem, which we call the *TopOneTuple* problem.

Recall that R is a relation with N tuples over an attribute set $\mathcal{A}$ with $|\mathcal{A}| = M$. Consider only queries Q that specify equality conditions on the specified set A of attributes, i.e., conditions such as $a_i = v_i$ for all $a_i \in A$. Let the scoring function be the simple additive function defined as the *dot product* between the query and the tuple, i.e.,

$$S(Q, A, t) = \sum_{a_i \in A} t[a_i] \cdot v_i$$

TopOneTuple Problem: Given a query Q, let t be the highest ranked tuple by the above scoring function. Given an integer $m < M$, is there a subset $A \subset \mathcal{A}$ of m attributes, such that when all attributes other than A are masked out, the highest ranked tuple remains as t?

THEOREM 1. *The TopOneTuple problem is NP-Complete.*

Proof: Clearly the problem is in NP as a solution can be easily verified in polynomial time. To prove NP-Hardness, we shall reduce from the *Vertex-Cover* problem [13], which is defined as follows: Given a graph $G = (V, E)$ consisting of v vertices and e edges, and an integer k, is there a subset of k vertices such that each edge has an endpoint in this set?

Given an instance of the Vertex-Cover problem, we shall construct an instance of the TopOneTuple problem as follows. Consider a relation R with v attributes and $e + 1$ tuples. Each edge (p, q) is represented as a tuple where the values of attributes p and q are each 0 and the remaining attribute values are each $v/(v-1)$. In addition, there is a special tuple t in which all attribute values are 1. We define a query Q that specifies 1 for all attribute values. We complete the instance of TopOneTuple by setting $m = k$.

Clearly, when no attributes are masked out, t is the highest ranked tuple for this query Q, since its score is v, whereas the score of any of the other tuples is $(v-2)v/(v-1)$.

Now suppose the instance of Vertex-Cover has a solution. Let A_Q be the subset of k vertices such that all edges in E have an endpoint in A_Q. Suppose we mask out the attributes not in A_Q. It is easy to see that t will remain the highest ranked tuple as its score

will be k, whereas each of the other tuples will have at least one 0 among the values of the attributes of A_Q and hence will have a maximum score of $(k-1)v/(v-1)$.

To prove the reverse, assume that the instance of TopOneTuple has a solution. Let A_Q be the set of the k attributes that have not been masked out, then t will remain the highest ranked tuple with a score of k. This is only possible if each of the other tuples has at least a 0 in one of the attributes of A_Q. If this was not so, then the score of such a tuple would have been $kv/(v-1)$, which is larger than the score of t which is a contradiction. Hence A_Q represents a solution for the Vertex-Cover instance. □

Thus, rank-based attribute selection is NP-complete.

3.2 Score-Based Attribute Selection

It intuitively appears that score-based attribute selection is easier than rank-based attribute selection, and indeed, as we shall shortly show, unlike rank-based attribute selection which is NP-Complete even for additive scoring functions, simple optimal score-based attribute selection algorithms exist for additive scoring functions. However, if we allow *arbitrarily complex* scoring functions, score-based attribute selection can easily be seen to be NP-Complete by a simple reduction from TopOneTuple.

THEOREM 2. *The score-based attribute selection problem is NP-Complete for general scoring functions*

Proof: (sketch) Given a TopOneTuple problem instance, create a score-based attribute selection instance where the score of a tuple *is its rank* in the TopOneTuple's database. I.e., scores of tuples are in the range [1..N]. It is trivial to see that the score-based attribute selection problem is NP-Complete. □

Note that in the above proof, the scoring function was very contrived, as it had to examine the entire database to determine the score of a tuple. In practice, scoring functions are much simpler, and we shall show in the next section that for additive scoring functions, simple greedy algorithms suffice. (In contrast, rank-based attribute selection is NP-Complete even for additive scoring functions).

4. ALGORITHMS FOR ATTRIBUTE SELECTION

In this section we present algorithms to solve the different attribute selection problem variants. In particular, we first present at a high level the optimal and two greedy algorithms. In Section 4.1, we specify when the greedy algorithms are optimal. Then we describe the implementation of the greedy algorithms for different variants of the attribute selection problem, given the execution interfaces of Figure 1. In Section 4.2, we present algorithms for rank-based as well as relative rank-based attribute selection, and in Section 4.3 we present algorithms for score-based attribute selection.

Optimal Algorithm Due to lack of space we simply sketch the details of the *optimal algorithm*. For example, in the case of rank-based selection, the optimal algorithm calculates all combinations of M attributes in groups of up to m and for each group invokes a pipelining interface to fetch answers and calculate the rank difference. The complexity is ${}^{M}C_{m}$, that is, exponential on m. The same (or worse) complexity applies for the other versions as well.

Greedy Algorithms Let $Dist(\mathcal{A}, A)$ be generic notation for $SDist(\mathcal{A}, A)$, $RDist(\mathcal{A}, A)$ or $RRDist(\mathcal{A}, A)$. The *Non Cumulative Greedy Algorithm* starts with an empty set A, and in each iteration, adds attribute a_i not already in A that minimizes the value of $Dist(\mathcal{A}, \{a_i\})$, stopping when m attributes have been picked. In contrast, the *Cumulative Greedy Algorithm* starts with an empty set A, and in each iteration, adds attribute a_i not already in A that minimizes $Dist(\mathcal{A}, A \cup \{a_i\})$, stopping when m attributes have been picked.[3].

```
Algorithm Rank-GreedyCum(R, Q, m, n)
{
01. Invoke pipelining interface S(Q, A) with output stream L
      and store top-n tuples of L in list L'
02. A = ∅ /*A is the set of attributes selected so far*/
03. for each j in 1, ..., m do
    { /*repeat until we find top-m attributes*/
04.   for each a_i in A − A do
      { /*Invoke M − |A| pipelining interfaces*/
05.     Invoke pipelining interfaces S(Q, A ∪ {a_i})
                with output streams L_i
06.     P(L_i) = ∅
        /*P(L_i) is the prefix of L_i retrieved so far*/
      }
07.   for each i in 1, ..., M − |A| do
      { /*retrieve tuples in parallel*/
08.     If not all tuples in L' are in P(L_i) then
        {
09.       Retrieve top tuple from L_i and add it to P(L_i)
10.       Calculate MinPossibleRDist(L, L_i) /* Equation 6 */
        }
11.     If all tuples in L' are in P(L_i) and RDist(A, A ∪ {a_i})
            < min_{l≠i} MinPossibleRDist(L, L_l) then
12.       add a_i to A, and break
      }
    }
13. return A
}
```

Figure 2: Cumulative Greedy Algorithm for Rank-based Attribute Selection

4.1 Optimality of Greedy Algorithms

We show that in the special case of score-based attribute selection for additive scoring functions, both the greedy algorithms are optimal.

THEOREM 3. *Both cumulative and non-cumulative score-based greedy algorithms are optimal for additive scoring functions.*

Proof: The scoring function is shown in Equation 1. For $m = 1$ the greedy is obviously optimal. We assume it is optimal for $m = l$, that is, we have selected a set A of l attributes that minimize $SDist(\mathcal{A}, A)$. If we add attribute a_i to A then $SDist(\mathcal{A}, A \cup \{a_i\}, S)$ - $SDist(\mathcal{A}, A) = SDist(\mathcal{A}, \{a_i\})$. Hence, we must choose a_i that minimizes $SDist(\mathcal{A}, \{a_i\})$. That is, the non-cumulative greedy algorithm is optimal. Also, maximizing $SDist(\mathcal{A}, \{a_i\})$ is equivalent to maximizing $SDist(\mathcal{A}, A \cup \{a_i\})$ in the additive case, so the cumulative greedy is also optimal. □

Notice that none of the greedy algorithms are optimal for the score-based attribute selection problem for general scoring functions (e.g., Equation 2).

[3]Both greedy algorithms can also be designed to operate in reverse, i.e., start with the complete attribute set and remove attributes one by one. While such a procedure has the potential for greater stability, it is significantly less efficient when we consider that $m << M$, and thus we do not pursue it any further.

4.2 Rank-Based Attribute Selection Algorithm

We present the cumulative greedy algorithm for the rank-based attribute selection problem, which is the most complex version. This algorithm exploits the pipelining interfaces and extracts minimal prefixes from the output streams to perform attribute selection. The non-cumulative greedy is a straightforward modification (simplification) and is not presented due to space concerns.

The intuition of the algorithm, which is shown in Figure 2, is the following. First, we compute the top-n results from the query output stream L (i.e., with all attributes $\mathcal{A}$ utilized), and store them in list L'. Then we compare a set of candidate combinations of attributes (to be utilized) with respect to their $RDist$ distance from $\mathcal{A}$, by invoking a pipelining interface for each of them and retrieving a minimal prefix from each that guarantees that our choice is correct. One attribute at a time is greedily added to the combinations of attributes for m iterations. Figure 3 shows an instance of the execution of the algorithm.

In more detail, the algorithm starts by computing the list L' of the top-n results of Q by invoking the pipelining interface $S(Q, \mathcal{A})$, i.e., by utilizing all attributes. Then we retrieve tuples from M pipelining interfaces $S(Q, \{a_1\}), \ldots, S(Q, \{a_M\})$ in parallel. For each stream L_i, we maintain the minimum possible rank-based distance *MinPossibleRDist*$(\mathcal{A}, \{a_i\})$, which is the tightest lower bound on *RDist*$(\mathcal{A}, \{a_i\})$ given the prefix $P(L_i)$ that has currently been retrieved.

The parallel retrieval from the L_i's terminates when all tuples $t \in L'$ have been retrieved from a stream L_i and the value of *RDist*$(\mathcal{A}, \{a_i\})$ is smaller than *MinPossibleRDist*(L, L_l) for all $l \neq i$. At this point, a_i is added to the set A.

In the $(j+1)$-th iteration, where j attributes have already been chosen, the next attribute is chosen by considering the $M - j$ alternatives created by adding each remaining attribute to the current A. We repeat until m attributes have been added to A.

Calculation of *MinPossibleRDist* *MinPossibleRDist* is calculated by Equation 6.

MinPossibleRDist$(L_Q, L_i, S) =$

$$\sum_{t \in L'} minTupleDist(L(t), L_i(t)) \tag{6}$$

where recall that $L(t)$ is the position of tuple t in list L, and $minTupleDist(L(t), L_i(t))$, which is defined in Equation 7, denotes the minimum possible distance between the positions of tuple t in L and L_i.

$minTupleDist(L(t), L_i(t)) =$

$$= \begin{cases} |L(t) - L_i(t)|, \ if\ tuple\ t \in P(L_i) \\ 0, \ if\ tuple\ t \notin P(L_i)\ and\ |P(L_i)| < L(t) \\ |L(t) - |P(L_i)| - 1|, \ if\ tuple\ t \notin P(L_i) \\ \qquad\qquad\qquad and\ |P(L_i)| \geq L(t) \end{cases} \tag{7}$$

where $P(L_i)$ denotes the prefix of the list L_i that has been retrieved so far.

The three branches of the equation correspond to the cases where tuple t has already been retrieved from L_i, has not been retrieved and less than $L(t)$ tuples have been retrieved from L_i, and has not been retrieved but more than $L(t)$ tuples have been retrieved from L_i, respectively.

The astute reader will notice that Equation 6 does not always compute the tightest minimum possible score, because if the third branch of Equation 7 is used for n' of the n tuples of L', it is not possible that all n' of them will be found on the $|P(L_i)| + 1$-st position of L_i. Hence, we add $\sum_{l=1}^{n'} l$ to the third branch of Equation 7.

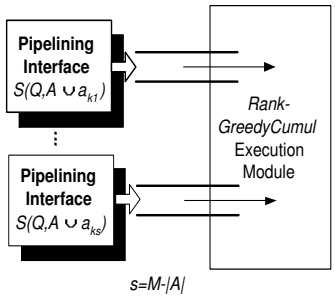

Figure 3: Instance of execution of rank-based cumulative greedy algorithm.

Algorithm Score-GreedyCum (R, Q, m, n)
{
1. Invoke pipelining interface $S(Q, \mathcal{A})$ with output stream L and store top-n tuples of L in list L'
2. $A = \emptyset$ /*A is the set of attributes selected so far*/
3. for each j in $1, \ldots, m$ do
   {/*repeat until we find top-m attributes*/
4. for each a_i in $\mathcal{A} - A$ do
   {
5. for each tuple $t \in L'$ do
   /*Invoke n single-result interfaces*/
6. Invoke single-result interfaces $S(Q, A \cup \{a_i\}, t)$
7. *SDist*$(\mathcal{A}, A \cup \{a_i\})$=$\sum_{t \in L'} |S(Q, \mathcal{A}, t) - S(Q, A, t))|$
   }
7. find a_i with minimum *SDist*$(\mathcal{A}, A \cup \{a_i\})$ and add it to A
   }
8. return A
}

Figure 4: Cumulative Greedy Algorithm for Score-Based Attribute Selection

Also notice that at an intermediate stage of the algorithm, it is possible that by adding any attribute to A, the distance will increase. That is, the *RDist*$(\mathcal{A}, A)$ calculated in line 11 is larger than the one calculated in the previous iteration. In this case, we terminate the algorithm and output A ($|A| < m$).

The relative rank-based version of the algorithm differs in that once the Top-n results of Q are calculated, the execution interfaces view them as the database instead of R.

Time Complexity The algorithm invokes $M + (M-1) + \cdots + (M-m+1) = m(2M-m+1)/2 = O(m \cdot M)$ pipelining interfaces and retrieves a minimal prefix of the output stream from each of them. If p is the average number of tuples retrieved from each output stream, the time complexity is $O(m \cdot M \cdot p)$. If richer interfaces for the scoring function are available, more efficient algorithms are possible. For example, if the scoring function is known and is monotone, and buffering is available, then we could use the Threshold Algorithm [11] to combine the original M streams to the next level of $M-1$ streams without re-invoking the pipelining interfaces.

4.3 Score-Based Attribute Selection Algorithm

The greedy algorithms for score-based attribute selection are significantly simpler. Figure 4 shows the cumulative greedy algorithm for this case. Once the top-n results L' are retrieved, we repeat-

edly invoke the single-result interface to get the score of the tuples in L' for each candidate attribute. We choose the attribute that minimizes the score-based distance function over all tuples in L' and repeat the above process until m attributes have been selected. The number of invocations of the single-result interface is $n(M + (M-1) \cdots + (M-m+1) = nm(2M-m+1)/2 = O(n \cdot m \cdot M)$. Hence the time complexity is $O(n \cdot m \cdot M)$ since the single-tuple interface has unit cost.

5. RELATED WORK

Dimensionality reduction and feature selection are somewhat related and well studied problems, but differ in key respects. Dimensionality reduction (Faloutsos and Lin [12], Keogh et al. [17]) aims at mapping high dimensional data to lower dimensional data, while preserving some metrics, such as distances, to the best possible extent. Applications of dimensionality reduction include indexing of data using index structures that work well only in lower dimensions. In our context we are not interested in distances between the data points, or distances from arbitrary query points, but rather in the similarity of data points to a specific query. Further, dimensionality reduction can compute new axes, whereas in our context a computed axis will be meaningless to the user, and we can only use existing dimensions (attributes) as axes.

There has been extensive work on feature selection in the area of classification machine learning. Guyon and Elisseeff [14] provide an excellent overview of feature selection, while feature selection for text classification is discussed in Yang and Pederson [22]. Among the three variants, Rank-based attribute selection is most closely related to feature selection, because in essence we wish to select attributes that best distinguish the top-n tuples of a query from the rest. For e.g., if we imagine a hypothetical Boolean class attribute that is set to `true` for each of the top-n tuples and `false` otherwise, the problem is similar to determining the best m attributes (or features) of a classifier attempting to accurately disambiguate the top-n tuples from the rest. However, the main differences are:

1. In feature selection, the important features are determined as an useful preprocessing step before building the classifier (and this is usually done by selecting attributes that are highly correlated with the class attribute), whereas in our problem, the scoring function is already available (albeit as a black box), and we wish to determine its most influential attributes that explain the high ranks of the top-n tuples.

2. $RDist$ measures the amount by which the *ranks* of the top-n tuples changes when a subset of attributes are used in score computations, whereas feature selection approaches are not based on changes in ranks. As an example, consider two attributes a_i and a_j, each over the domain $\{1, 2, 3\}$. Let the top-n tuples returned by a query have the following values for these attributes: $(3, 3), (3, 3), \ldots, (3, 3), (2, 1)$. Let the remaining $N - n$ tuples have the following values for these attributes: $(3, 3), (1, 2), (1, 2), \ldots, (1, 2)$. From the symmetry, it is quite clear that feature selection methods cannot distinguish between a_i and a_j because both are equally correlated with the hypothetical boolean class attribute (mentioned above). However, $RDist(\mathcal{A}, \{a_i\}) = 1$, whereas $RDist(\mathcal{A}, \{a_j\}) = N - n$ (since the nth tuple in the top-n will be placed at the very bottom of the table if we ranked only according to the values of a_j).

In Section 6 we experiment with a baseline feature selection approach based on selecting attributes that are most correlated to the top-n tuples by using the chi-square measure after suitable discretization of numeric attributes. Feature selection techniques are less appropriate for the other proposed variants, score-based and relative rank-based.

Although there has been a significant amount of work on visual interfaces to databases (e.g. VisDB [16], Polaris [20] and Tioga-2 [2]) none of this work has addressed the selection of attributes to display. Feature selection has been used in the context of information visualization, but the goals (which are similar to those of dimensionality reduction) have been preservation of distance metrics or clusters (Rheingans and desJardins [19]) and not ranking.

Techniques to find top-K results where the overall score is a combination of scores on individual attributes are described by Fagin et al. [11, 10] and Hristidis et al. [15]. Different ways of comparing (and combining, or aggregating) rankings are compared by Dwork et al. [8]. These include measures such as the Kendall τ method, and Spearman's footrule. Ranking comparison and rank aggregation in the context of ranking with ties is described in Fagin et al. [9].

In the context of querying on items with multiple attributes, the problem of ranking the query results in the presence of many answers is considered by Agrawal et al. [1] and Chaudhuri et al. [6]. In particular, [6] exploits the query workload (past user queries) to discover user preferences (popularity of attribute values and correlations between them), which are then used in ranking the results. On the other hand, Nambiar and Kambhampati [18] tackle the few answers problem, and relax attributes of the query that have the least amount of correlation to other attributes. Their techniques can be used to perform automatic query extension when needed.

Chakrabarti et al. [5] consider the problem of browsing large sets of query answers, and suggests ways in which to choose attributes as candidates for creating branches in a browsing tree. They use a workload of past queries to pick an ordering of attributes. Attributes that occur in fewer than some fraction of queries are not considered further. An exploration cost model is used to pick the best attribute to use as the branching attribute at the next level of the tree. Intuitively, attributes that appear often in queries, with selections that significantly reduce the number of candidate results are preferred. Their exploration cost model is not relevant in our scenario. The idea of preferring attributes that are most often queried upon is relevant in our context, but is a coarse approximation since they may not be highly correlated with the score or rank of tuples in a particular answer.

6. EXPERIMENTS

In this section we report on the results of an experimental evaluation of our attribute selection algorithms. We implemented the attribute selection variants (including the hybrid *Split-Pane* approach) proposed in this paper (see Section 4), as well as several baseline attribute selection methods (such as selecting by weight and selecting based on how correlated the attributes were with the query rankings). We evaluated both the quality of the selections obtained, as well as the performance of the various approaches.

6.1 Experimental Setup and Tuple Ranking Functions

We used Microsoft SQL Server 2000 RDBMS on a P4 2.8-GHz PC with 1 GB of RAM and 120 GB HDD for our experiments. We implemented all algorithms in C#, and connected to the RDBMS through DAO. We used two datasets. First, we use portions of an online used-car automotive dealer's nationwide database. In this database, each tuple represents a car for sale, and each column represented an attribute of the car. For our studies we considered

Query	*Score-Based*	*Rank-Based*	*Relative-Rank-Based*	*Split-Pane*	*By-Weight*	*By-Correlation*
Cars-1	3.00	2.60	2.50	4.60	2.40	1.40
Cars-2	3.17	2.33	1.83	4.50	3.00	2.00
Cars-3	3.33	2.50	2.50	4.50	3.17	1.50
Cars-4	3.50	2.83	2.17	4.17	2.83	2.17
Cars-5	3.25	2.38	1.88	4.00	2.63	2.00
Cars-AVG	**3.25**	**2.53**	**2.18**	**4.35**	**2.81**	**1.81**
Homes-1	2.50	1.67	1.50	3.83	2.33	2.00
Homes-2	2.00	2.33	1.67	3.67	3.00	2.67
Homes-3	2.67	1.67	3.33	3.33	3.33	2.00
Homes-4	3.00	2.33	1.67	3.00	3.00	2.00
Homes-5	3.00	1.50	1.50	2.00	3.50	2.50
Homes-AVG	**2.63**	**1.90**	**1.93**	**3.16**	**3.03**	**2.23**

Figure 5: User Study: Compare various attribute selection methods

37 attributes, including numeric attributes such as Price, Engine Size, Year; categorical attributes such as Make, Model, Color; and Boolean attributes such as AC, Power Brakes, etc. To make our quality experiments more manageable, we restricted our database to about $15,191$ cars sold in the Dallas-Ft Worth Metroplex. The second dataset is a US home properties dataset extracted from a nationwide realtor's website. This dataset has $20,943$ properties listed with 25 attributes like square footage, price, number of bedrooms, and so on.

Since attribute selection depends on the specific tuple scoring function used, we considered two tuple scoring functions: WeightedL2 and Conditional, which have described in Section 2. In our experiments, users were initially allowed to specify weights as well as the values of query attributes. The average of these weights was recorded, and used as default value for unspecified attributes for all experiments including our user study. We emphasize that the goal of this paper is *not* to investigate the quality of this (or any other) tuple scoring function. Tuple scoring functions of various kinds have already appeared in prior published work (e.g. the Conditional algorithm, [6]), while the others (such as WeightedL2) are largely specified by domain experts or by the users themselves. Our goal in this paper is to demonstrate that, *given a tuple scoring function* such as the above, our attribute selection algorithms indeed pick the top attributes that best "explain" these rankings.

6.2 User Studies

We first investigated whether our basic premise is reasonable, i.e., that users are indeed interested in viewing the attributes that influence the ranking of tuples the most. We informally tested our system with lots of queries ourselves. Most of the time both score-based as well as rank-based approaches produced intuitively meaningful attributes that would be useful to a user looking for cars or homes in such databases. We also attempted a methodical user study of our system. Preparing an extensive experimental setup for such subjective testing was extremely challenging - unlike other established disciplines such as Information Retrieval where standard benchmarks are available, in our case we had to conduct user studies using limited resources in a nascent area where no benchmarks exist.

For our user studies we requested participation from 69 people from our respective universities and institutions. We used both the cars and the homes datasets and considered two tuple scoring functions: WeightedL2 and Conditional [6]. We only present the results for cars-WeightedL2 and homes-Conditional in this paper.

We solicited five typical queries for each dataset that represent a heterogeneous mix of different profiles of potential car/home buyers - young teenagers, rich couples, suburban families, etc. The queries are the following:

Cars dataset queries

1. make = "Infinity" and mileage $<$ 20000
2. make = "Lexus" and price = 40k-60k
3. make = "Honda" and price = 0-20k
4. make = "Mercedes" and price = 40-60k
5. make = "Audi" and price= 20-40k

Homes dataset queries

1. price = 400k-500k, Tarrant County, TX
2. multiple family homes, price$>$500k, bedrooms$>$5, Tarrant County, TX
3. condos with pool and clubhouse, Tarrant County, TX
4. price = 400k-500k, Tarrant County, TX
5. multi family homes, 2 stories, Tarrant County, TX

The weights used by the WeightedL2 tuple ranking, as well for the *By-Weight* attribute selection (explained below) were created after a mini-survey on the global importance of the attributes in the datasets.

For each of these queries, we selected the top attributes of the result by the following methods:

1. *Score-Based*: Attributes are selected by score (Equation 3).
2. *Rank-Based*: Attributes are selected by rank (Equation 4).
3. *Relative-Rank-Based*: Attributes are selected by relative rank (Equation 5).
4. *Split-Pane*: The displayed attributes are grouped into three vertical panes, each having the top-4 attributes selected by *Score*, *Rank* and *Relative-Rank* respectively. Duplicate attributes across panes are not displayed.
5. *By-Weight*: Attributes are selected by their weights, which are hardcoded as explained above, and do not depend on the query.
6. *By-Correlation*: Attributes are selected according to the chi-square value between them (i.e., ignoring the other attributes) and the top-n tuples (see Section 5).

Note that *By-Weight* and *By-Correlation* are used as baselines to compare against our proposed orderings. We set the number m of displayed attributes to 12 and the number n of output tuples to 10. The results were presented to users in tabular form - six 10×12 tables side by side, with attributes ordered from left to right in each table according to the order in which they were picked by the respective algorithms Users were not told how the tables were generated, in order to avoid any bias. Users were requested to examine the attributes in each table and indicate whether they agreed that the attributes were helpful in explaining the high ranks of these tuples (on a scale of [1..5] with 5 indicating that they strongly agreed with the attributes shown).

Figure 5 show a table where the responses of all users to this survey has been averaged. Notice that the rows "Cars-AVG" and "Homes-AVG" do not correspond to averaging the 5 query rows because we average per user and not per query and different numbers of users answered each query of the survey. As can be seen, most users preferred the *Split-Pane* selections, since it combines the characteristics of the other selections. Users seemed to really like the fact that different attribute sets serve to explain different facets of the ranking process. Furthermore, as we argued in Section 2.2, *Score-Based* is most closely related to *By-Weight* since the former may be considered a query-specific version of the latter. Our results were as expected - users preferred *Score-based* over *By-Weight* in one dataset, whereas the preferences were split in the other dataset. Likewise, as discussed in Section 5, *Rank-Based* is most closely related to *By-Correlation*, and in our survey users preferred the former over the latter in one dataset, but reversed their opinions in the other dataset. The *Relative Rank-Based* attribute selection was not as preferred compared to either *Score-Based* or *Rank-Based*, perhaps indicating that its top attributes should be listed only after attributes of the first two types have been displayed by a hybrid approach such as *Split-Pane*.

While the results of this user survey appear promising, we caution that it would be premature to interpret the results as conclusive evidence that one specific attribute selection approach is better that another. Rather, the clear preference of *Split-Pane* by users indicates that each variant provides very different types of information, each of which has a role to play in helping users compare the ranked tuples of a query.

6.3 Greedy versus Optimal - Approximation Quality

In this subsection we investigate the different variants of the greedy algorithm and measure how closely they approximate the selections produced by the Optimal algorithm (see Section 4). We only present the results on the cars dataset for Rank-Based which is the most complex to compute due to space considerations. We generated a set of 50 random queries as follows. For each query, we picked between one to three attributes at random, and for each attribute, we picked a value from its domain at random. For each query we ran all variants of our attribute selection algorithms for various parameter settings (m is varied between 1 and 5, while n is varied from 10, 100 to 1000).

Since Optimal is a very slow running algorithm, we could only experiment with it for 20 queries and for values of $m \leq 4$. Even so, several interesting conclusions could be derived from this relatively small experimental framework.

Figure 6 shows the average *list distance* (Spearman's footrule) between Rank-GreedyCum and Rank-GreedyNonCum varying as a function of m for the top-10 results. That is, the y-axis represents the average number of positions that an element in the original rankings is shifted by. Thus we can see that when we use

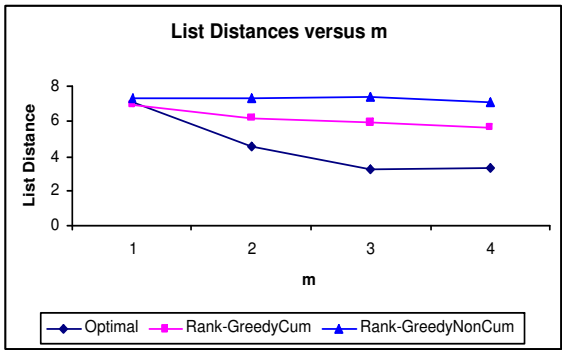

Figure 6: Rank-Greedy, Optimal vs m

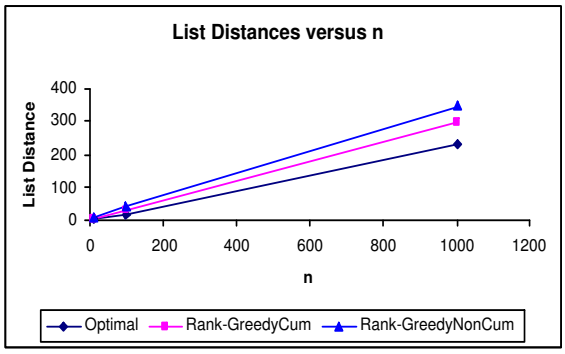

Figure 7: Rank-Greedy, Optimal vs n

4 attributes, Optimal does quite well in the sense that a top-10 tuple is only 3 positions away from its true rank on the average. The greedy variants are not that far behind, with Rank-GreedyCum (resp. Rank-GreedyNonCum) shifting top-10 tuples by 6 (resp. 7) positions on the average. This experiment thus shows that both rank-based greedy variants are good approximations of Optimal, although as m gets larger, the approximation factor does increase, with Rank-GreedyNonCum being consistently worse than Rank-GreedyCum.

Although the results for the case when $n = 10$ seem acceptable, we ran experiments to see if a similar behavior is observed for larger values of n. Figure 7 shows that with $m = 4$, as n increases, the distance between the result lists steadily worsens, with Optimal being consistently better than Rank-GreedyCum, which in turn is consistently better than Rank-GreedyNonCum. The reason for this worsening in rank quality is attributable to the fact that once the target set of tuples increases substantially, it becomes extremely difficult to have the same few (4) attributes explain their rankings. One option is to adopt the stance that tuples lower down the ranks are not very important for the query anyway, and hence it is acceptable to have large error in explaining their rankings. Another option is to try to determine the best attributes on a tuple by tuple basis.

Next, we compare rank-based versus score-based algorithms. Figure 8 shows the performances of various greedy variants versus m for $n = 10$. It was interesting to see that although rank-based variants usually produced better approximations than score-based variants, the difference was less pronounced compared to the difference between cumulative and noncumulative variants. This be-

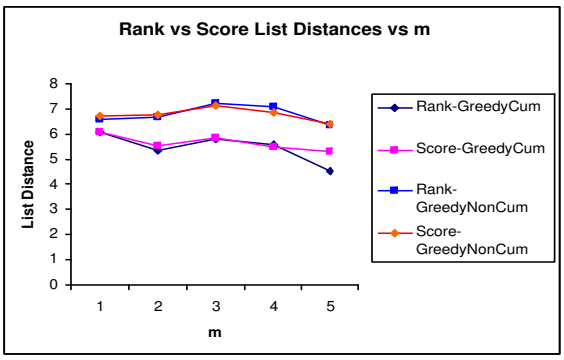

Figure 8: Rank, Score vs m

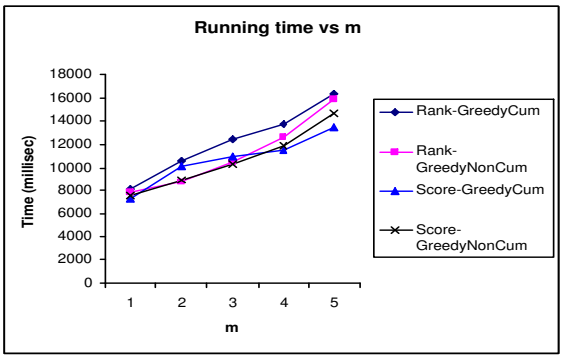

Figure 10: Running time vs m

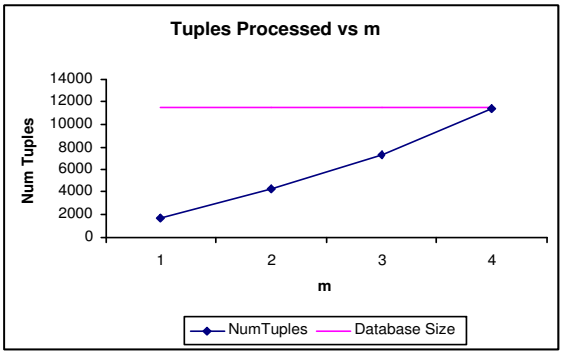

Figure 9: Tuples processed by Rank-GreedyNonCum vs m

havior was consistently observed at other values of n. However, rank-based algorithms produced much better approximations in the case where only the top result-tuples of the candidate attribute selection were considered instead of all the results, as is the case for the relative rank-based selection.

6.4 Execution Cost

Lastly, we performed experiments to compare the running time of different variants of our attribute selection algorithms. We only present results for the cars dataset. Since our algorithms were tightly coupled with the vertical scoring function WeightedL2, separating out the performances of the former from the latter was a challenging task. We implemented the WeightedL2 scoring function with masking using the pipelined execution model. Since our implementation was done through the SQL query interface using a RDBMS, the pipelining model had to be simulated with appropriate ORDER BY clauses. Naturally this is less efficient compared to a true pipelining model implemented inside the database server - this is still an active area of research. However, it was still instructive to *count* the number of tuples that were processed by our algorithms through these pipelines. We illustrate with an example. We executed the Rank-GreedyNonCum algorithm and retrieved the top-n tuples for n in $[1..10]$, and then for each attribute a_i, we determined the length of the shortest prefix of the corresponding list L_i that contains these top-n tuples. The sum of the lengths of the m shortest prefixes was an estimate of the total number of tuples that were processed through these interfaces before the algorithm terminated.

Figure 9 illustrated how the total number of tuples processed in pipelines increases as a function of m (averaged over several queries with n set to 5). Likewise, the total number of processed tuples also increased with n; we omit the details of these experiments. Note that for some cases the total number of processed tuples is smaller than that database size - thus a true pipelining implementation inside the RDBMS has the potential of being faster than a database scan, especially for larger databases.

Next, we compared the average running times of the different variants of our actual implementations (i.e., SQL-based with ORDER BY clauses) after running them on several queries. While we caution that our database was too small ($15,191$ tuples) to allow highly reliable timing experiments, we did observe expected trends. Figure 10 shows that the running times increase with increasing m (for a fixed $n = 1000$), with the rank-based (resp. cumulative) variants generally slower than the score-based (resp. non-cumulative) counterparts. The running times also increased with increasing n and database size; we omit details of these experiments due to lack of space.

7. CONCLUSIONS AND FUTURE WORK

We addressed the problem of selecting the top m attributes from the view point of helping a user understand what factors most influenced a ranking system in its ranking decisions. We presented several variants of the problem, showed that several of these variants are NP-hard, and presented efficient greedy heuristics. We performed a user study demonstrating the benefits of a hybrid approach that returns the top attributes from each of these variants. We also presented a performance study comparing two versions of the greedy heuristic, showing that the cumulative version of greedy performs better.

In the future we plan to investigate alternative attribute selection and ordering criteria and techniques. For example, how can we efficiently and effectively compute good and bad attributes, as well as perform attribute selection based on multiple queries (mentioned in Section 2.2)? Also, how can we automatically discover useful attributes that have little to do with ranking of tuples? We also plan to study attribute ordering (as opposed to attribute selection) in more detail. Finally, we plan to investigate the integration of attribute selection with specific tuple ranking functions for better performance.

Acknowledgments: We would like to thank Ullas Nambiar for his help with writing the wrapper for data extraction, Arjun Saraswat for his help in programming, and all those who participated in the user study. Finally, we would like to thank the reviewers for their valuable comments.

8. REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, Gautam Das, and A. Gionis. Automated ranking of database query results. In *CIDR*, 2003.

[2] Alexander Aiken, Jolly Chen, Michael Stonebraker, and Allison Woodruff. Tioga-2: A direct manipulation database visualization environment. In *ICDE*, 1996.

[3] A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank: Authority-Based Keyword Search in Databases. In *VLDB*, 2004.

[4] Nicolas Bruno, Luis Gravano, and Amelie Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, 2002.

[5] Kaushik Chakrabarti, Surajit Chaudhuri, and Seung won Hwang. Automatic categorization of query results. In *SIGMOD*, pages 755–766, 2004.

[6] Surajit Chaudhuri, Gautam Das, Vagelis Hristidis, and Gerhard Weikum. Probabilistic ranking of database query results. In *VLDB*, 2004.

[7] P. Diaconis and R. Graham. Spearman's footrule as a measure of disarray. In *J. of the Royal Statistical Society, Series B, 39(2):262–268*, 1977.

[8] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation methods for the Web. In *WWW Conf.*, 2001.

[9] Ronald Fagin, Ravi Kumar, Mohammad Mahdiany, D. Sivakumar, and Erik Veez. Comparing and aggregating rankings with ties. In *PODS*, 2004.

[10] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Comparing top k lists. In *Procs.ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003.

[11] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[12] Christos Faloutsos and King-Ip Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *SIGMOD*, 1995.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability–A Guide to the Theory of NP-Completeness*. W. H. Freeman And Company, 1979.

[14] Isabelle Guyon and Andre Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3(mar):1157–1182, 2003.

[15] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER:a system for the efficient execution of multi-parametric ranked queries. In *SIGMOD*, May 2001.

[16] D. Keim and H-P. Kriegel. VisDB: Database exploration using multidimensional visualization. *Computer Graphics and Applications Journal*, 1994.

[17] Eamonn Keogh, Kaushik Chakrabarti, Sharad Mehrotra, and Michael Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. In *SIGMOD*, 2001.

[18] Ullas Nambiar and Subbarao Kambhampati. Mining approximate functional dependencies and concept similarities to answer imprecise queries. In *WebDB: Int'l Workshop on the Web and Databases*, 2004.

[19] Penny Rheingans and Marie desJardins. Assessing projection quality for high-dimensional information visualization. Technical report, Univ. Maryland at Baltimore County, 2002.

[20] Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Trans. Vis. Comput. Graph*, 8(1):52–65, 2002.

[21] Lisa A. Torrey. An active learning approach to efficiently ranking retrieval engines. Technical Report TR2003-449, Dartmouth, 2003.

[22] Yiming Yang and Jan O. Pederson. A comparative study on feature selection in text categorization. In *ICML*, 1997.