Probabilistic Ranking of Database Query Results

Surajit Chaudhuri, Gautam Das
Microsoft Research
One Microsoft Way
Redmond, WA 98053
USA
{surajitc, gautamd}@microsoft.com

Vagelis Hristidis
School of Comp. Sci.
Florida Intl. University
Miami, FL 33199
USA
vagelis@cs.fiu.edu

Gerhard Weikum
MPI Informatik
Stuhlsatzenhausweg 85
D-66123 Saarbruecken
Germany
weikum@mpi-sb.mpg.de

Abstract

We investigate the problem of ranking answers to a database query when many tuples are returned. We adapt and apply principles of probabilistic models from Information Retrieval for structured data. Our proposed solution is domain independent. It leverages data and workload statistics and correlations. Our ranking functions can be further customized for different applications. We present results of preliminary experiments which demonstrate the efficiency as well as the quality of our ranking system.

1. Introduction

Database systems support a simple Boolean query retrieval model, where a selection query on a SQL database returns all tuples that satisfy the conditions in the query. This often leads to the *Many-Answers Problem*: when the query is not very selective, too many tuples may be in the answer. We use the following running example throughout the paper:

Example: *Consider a realtor database consisting of a single table with attributes such as* (*TID, Price, City, Bedrooms, Bathrooms, LivingArea, SchoolDistrict, View, Pool, Garage, BoatDock …*). *Each tuple represents a home for sale in the US.*

Consider a potential home buyer searching for homes in this database. A query with a not very selective condition such as "City=Seattle and View=Waterfront" may result in too many tuples in the answer, since there are many homes with waterfront views in Seattle.

The Many-Answers Problem has been investigated outside the database area, especially in Information Retrieval (IR), where many documents often satisfy a given keyword-based query. Approaches to overcome this problem range from *query reformulation* techniques (e.g., the user is prompted to refine the query to make it more selective), to *automatic ranking* of the query results by their degree of "relevance" to the query (though the user may not have explicitly specified how) and returning only the top-*K* subset.

It is evident that automated ranking can have compelling applications in the database context. For instance, in the earlier example of a homebuyer searching for homes in Seattle with waterfront views, it may be preferable to first return homes that have other desirable attributes, such as good school districts, boat docks, etc. In general, customers browsing product catalogs will find such functionality attractive.

In this paper we propose an automated ranking approach for the Many-Answers Problem for database queries. Our solution is principled, comprehensive, and efficient. We summarize our contributions below.

Any ranking function for the Many-Answers Problem has to look beyond the attributes specified in the query, because all answer tuples satisfy the specified conditions[1]. However, investigating unspecified attributes is particularly tricky since we need to determine what the user's preferences for these unspecified attributes are. In this paper we propose that the ranking function of a tuple depends on two factors: (a) a *global score* which captures the global importance of unspecified attribute values, and

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

[1] In the case of document retrieval, ranking functions are often based on the frequency of occurrence of query values in documents (*term frequency*, or TF). However, in the database context, especially in the case of categorical data, TF is irrelevant as tuples either contain or do not contain a query value. Hence ranking functions need to also consider values of unspecified attributes.

(b) a *conditional score* which captures the strengths of dependencies (or correlations) between specified and unspecified attribute values. For example, for the query "City = Seattle and View = Waterfront", a home that is also located in a "SchoolDistrict = Excellent" gets high rank because good school districts are globally desirable. A home with also "BoatDock = Yes" gets high rank because people desiring a waterfront are likely to want a boat dock. While these scores may be estimated by the help of domain expertise or through user feedback, we propose an automatic estimation of these scores via *workload as well as data* analysis. For example, past workload may reveal that a large fraction of users seeking homes with a waterfront view have also requested for boat docks.

The next challenge is how do we translate these basic intuitions into principled and quantitatively describable ranking functions? To achieve this, we develop ranking functions that are based on *Probabilistic Information Retrieval (PIR)* ranking models. We chose PIR models because we could extend them to model data dependencies and correlations (the critical ingredients of our approach) in a more principled manner than if we had worked with alternate IR ranking models such as the Vector-Space model. We note that correlations are often ignored in IR because they are very difficult to capture in the very high-dimensional and sparsely populated feature spaces of text data, whereas there are often strong correlations between attribute values in relational data (with functional dependencies being extreme cases), which is a much lower-dimensional, more explicitly structured and densely populated space that our ranking functions can effectively work on.

The architecture of our ranking has a pre-processing component that collects database as well as workload statistics to determine the appropriate ranking function. The extracted ranking function is materialized in an *intermediate knowledge representation layer*, to be used later by a query processing component for ranking the results of queries. The ranking functions are encoded in the intermediate layer via intuitive, easy-to-understand "atomic" numerical quantities that describe (a) the global importance of a data value in the ranking process, and (b) the strengths of correlations between pairs of values (e.g., "if a user requests tuples containing value y of attribute Y, how likely is she to be also interested in value x of attribute X?"). Although our ranking approach derives these quantities automatically, our architecture allows users and/or domain experts to tune these quantities further, thereby customizing the ranking functions for different applications.

We report on a comprehensive set of experimental results. We first demonstrate through user studies on real datasets that our rankings are superior in quality to previous efforts on this problem. We also demonstrate the efficiency of our ranking system. Our implementation is especially tricky because our ranking functions are relatively complex, involving dependencies/correlations between data values. We use novel pre-computation techniques which reduce this complex problem to a problem efficiently solvable using Top-K algorithms.

The rest of this paper is organized as follows. In Section 2 we discuss related work. In Section 3 we define the problem and outline the architecture of our solution. In Section 4 we discuss our approach to ranking based on probabilistic models from information retrieval. In Section 5 we describe an efficient implementation of our ranking system. In Section 6 we discuss the results of our experiments, and we conclude in Section 7.

2. Related Work

Extracting ranking functions has been extensively investigated in areas outside database research such as Information Retrieval. The vector space model as well as probabilistic information retrieval (PIR) models [4, 28, 29] and statistical language models [14] are very successful in practice. While our approach has been inspired by PIR models, we have adapted and extended them in ways unique to our situation, e.g., by leveraging the structure as well as correlations present in the structured data and the database workload.

In database research, there has been some work on ranked retrieval from a database. The early work of [23] considered vague/imprecise similarity-based querying of databases. The problem of integrating databases and information retrieval systems has been attempted in several works [12, 13, 17, 18]. Information retrieval based approaches have been extended to XML retrieval (e.g., see [8]). The papers [11, 26, 27, 32] employ relevance-feedback techniques for learning similarity in multimedia and relational databases. Keyword-query based retrieval systems over databases have been proposed in [1, 5, 20]. In [21, 24] the authors propose SQL extensions in which users can specify ranking functions via soft constraints in the form of preferences. The distinguishing aspect of our work from the above is that we espouse automatic extraction of PIR-based ranking functions through data and workload statistics.

The work most closely related to our paper is [2] which briefly considered the Many-Answers Problem (although its main focus was on the *Empty-Answers Problem*, which occurs when a query is too selective, resulting in an empty answer set). It too proposed automatic ranking methods that rely on workload as well as data analysis. In contrast, however, the current paper has the following novel strengths: (a) we use more principled probabilistic PIR techniques rather than ad-hoc techniques "loosely based" on the vector-space model, and (b) we take into account dependencies and correlations between data values, whereas [2] only proposed a form of global score for ranking.

Ranking is also an important component in collaborative filtering research [7]. These methods require

training data using queries as well as their ranked results. In contrast, we require workloads containing queries only.

A major concern of this paper is the query processing techniques for supporting ranking. Several techniques have been previously developed in database research for the Top-K problem [8, 9, 15, 16, 31]. We adopt the Threshold Algorithm of [16, 19, 25] for our purposes, and show how novel pre-computation techniques can be used to produce a very efficient implementation of the Many-Answers Problem. In contrast, an efficient implementation for the Many-Answers Problem was left open in [2].

3. Problem Definition and Architecture

In this section, we formally define the Many-Answers Problem in ranking database query results, and also outline a general architecture of our solution.

3.1 Problem Definition

We start by defining the simplest problem instance. Consider a database table D with n tuples $\{t_1, \ldots, t_n\}$ over a set of m categorical attributes $A = \{A_1, \ldots, A_m\}$. Consider a "SELECT * FROM D" query Q with a conjunctive selection condition of the form "WHERE $X_1=x_1$ AND … AND $X_s=x_s$", where each X_i is an attribute from A and x_i is a value in its domain. The set of attributes $X = \{X_1, \ldots, X_s\} \subseteq A$ is known as the set of attributes *specified* by the query, while the set $Y = A - X$ is known as the set of *unspecified* attributes. Let $S \subseteq \{t_1, \ldots, t_n\}$ be the answer set of Q. The *Many-Answers Problem* occurs when the query is not too selective, resulting in a large S.

The above scenario only represents the simplest problem instance. For example, the type of queries described above are fairly restrictive; we refer to them as *point queries* because they specify single-valued equality conditions on each of the specified attributes. In a more general setting, queries may contain range/IN conditions, and/or Boolean operators other than conjunctions. Likewise, databases may be multi-tabled, may contain a mix of categorical and numeric data, as well as missing or NULL values. While our techniques extend to all these generalizations, in the interest of clarity (and due to lack of space), the main focus of this paper is on ranking the results of conjunctive point queries on a single categorical table (without NULL values).

3.2 General Architecture of our Approach

Figure 1 shows the architecture of our proposed system for enabling ranking of database query results. As mentioned in the introduction, the main components are the preprocessing component, an intermediate knowledge representation layer in which the ranking functions are encoded and materialized, and a query processing component. The modular and generic nature of our system allows for easy customization of the ranking functions for different applications.

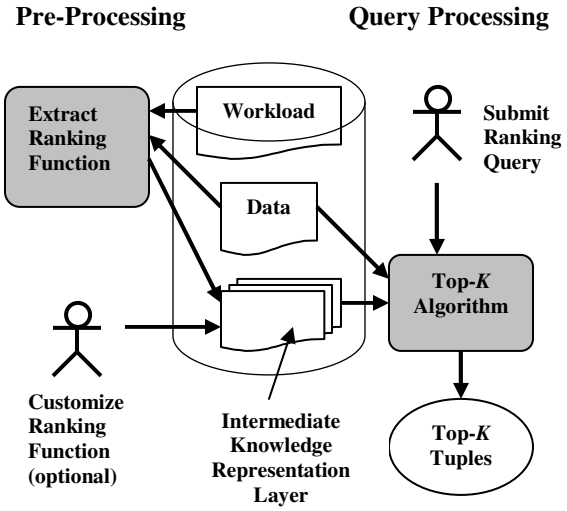

Figure 1: Architecture of Ranking System

In the next section we discuss PIR-based ranking functions for structured data.

4. Ranking Functions: Adaptation of PIR Models for Structured Data

In this section we discuss PIR-based ranking functions, and then show how they can be adapted for structured data. We discuss the semantics of the atomic building blocks that are used to encode these ranking functions in the intermediate layer. We also show how these atomic numerical quantities can be estimated from a variety of knowledge sources, such as data and workload statistics, as well as domain knowledge.

4.1 Review of Probabilistic Information Retrieval

Much of the material of this subsection can be found in textbooks on Information Retrieval, such as [4] (see also [28, 29]). We will need the following basic formulas from probability theory:

Bayes' Rule: $$p(a \mid b) = \frac{p(b \mid a)\, p(a)}{p(b)}$$

Product Rule: $$p(a, b \mid c) = p(a \mid c)\, p(b \mid a, c)$$

Consider a document collection D. For a (fixed) query Q, let R represent the set of *relevant* documents, and $\overline{R} = D - R$ be the set of *irrelevant* documents. In order to rank any document t in D, we need to find the probability of the relevance of t for the query given the text features of t (e.g., the word/term frequencies in t), i.e., $p(R|t)$. More formally, in probabilistic information retrieval, documents

are ranked by decreasing order of their odds of relevance, defined as the following *score*:

$$Score\ (t) = \frac{p(R \mid t)}{p(\overline{R} \mid t)} = \frac{\frac{p(t \mid R)\,p(R)}{p(t)}}{\frac{p(t \mid \overline{R})\,p(\overline{R})}{p(t)}} \propto \frac{p(t \mid R)}{p(t \mid \overline{R})}$$

The main issue is, how are these probabilities computed, given that R and $\overline{R}$ are unknown at query time? The usual techniques in IR are to make some simplifying assumptions, such as estimating R through user feedback, approximating $\overline{R}$ as D (since R is usually small compared to D), and assuming some form of independence between query terms (e.g., the *Binary Independence Model*).

In the next subsection we show how we adapt PIR models for structured databases, in particular for conjunctive queries over a single categorical table. Our approach is more powerful than the Binary Independence Model as we also leverage data dependencies.

4.2 Adaptation of PIR Models for Structured Data

In our adaptation of PIR models for structured databases, each tuple in a single database table D is effectively treated as a "document". For a (fixed) query Q, our objective is to derive *Score*(t) for any tuple t, and use this score to rank the tuples. Since we focus on the Many-Answers problem, we only need to concern ourselves with tuples that satisfy the query conditions. Recall the notation from Section 3.1, where X is the set of attributes specified in the query, and Y is the remaining set of unspecified attributes. We denote any tuple t as partitioned into two parts, $t(X)$ and $t(Y)$, where $t(X)$ is the subset of values corresponding to the attributes in X, and $t(Y)$ is the remaining subset of values corresponding to the attributes in Y. Often, when the tuple t is clear from the context, we overload notation and simply write t as consisting of two parts, X and Y (in this context, X and Y are thus sets of values rather than sets of attributes).

Replacing t with X and Y (and $\overline{R}$ as D as mentioned in Section 4.1 is commonly done in IR), we get

$$Score(t) \propto \frac{p(t \mid R)}{p(t \mid D)} = \frac{p(X, Y \mid R)}{p(X, Y \mid D)}$$

$$\propto \frac{p(X \mid R)\ p(Y \mid X, R)}{p(X \mid D)\ p(Y \mid X, D)}$$

Since for the Many-Answers problem we are only interested in ranking tuples that satisfy the query conditions, and all such tuples have the same X values, we can treat any quantity not involving Y as a constant. We thus get

$$Score(t) \propto \frac{p(Y \mid X, R)}{p(Y \mid X, D)}$$

Furthermore, the relevant set R for the Many-Answers problem is a subset of all tuples that satisfy the query conditions. One way to understand this is to imagine that R is the "ideal" set of tuples the user had in mind, but who only managed to partially specify it when preparing the query. Consequently the numerator $p(Y \mid X, R)$ may be replaced by $p(Y \mid R)$. We thus get

$$Score(t) \propto \frac{p(Y \mid R)}{p(Y \mid X, D)} \qquad (1)$$

We are not quite finished with our derivation of *Score*(t) yet, but let us illustrate Equation 1 with an example. Consider a query with condition "City=Kirkland and Price=High" (Kirkland is an upper class suburb of Seattle close to a lake). Such buyers may also ideally desire homes with waterfront or greenbelt views, but homes with views looking out into streets may be somewhat less desirable. Thus, p(View=Greenbelt | R) and p(View=Waterfront | R) may both be high, but p(View=Street | R) may be relatively low. Furthermore, if in general there is an abundance of selected homes with greenbelt views as compared to waterfront views, (i.e., the denominator p(View=Greenbelt | City=Kirkland, Price=High, D) is larger than p(View=Waterfront | City=Kirkland, Price=High, D)), our final rankings would be homes with waterfront views, followed by homes with greenbelt views, followed by homes with street views. Note that for simplicity, we have ignored the remaining unspecified attributes in this example.

4.2.1 Limited Independence Assumptions

One possible way of continuing the derivation of *Score*(t) would be to make independence assumptions between values of different attributes, like in the Binary Independence Model in IR. However, while this is reasonable with text data (because estimating model parameters like the conditional probabilities $p(Y \mid X)$ poses major accuracy and efficiency problems with sparse and high-dimensional data such as text), we have earlier argued that with structured data, dependencies between data values can be better captured and would more significantly impact the result ranking. An extreme alternative to making sweeping independence assumptions would be to construct comprehensive dependency models of the data (e.g. probabilistic graphical models such as Markov Random Fields or Bayesian Networks [30]), and derive ranking functions based on these models. However, our preliminary investigations suggested that such approaches, particularly for large datasets, have unacceptable pre-processing and query processing costs.

Consequently, in this paper we espouse an approach that strikes a middle ground. We only make limited forms

of independence assumptions – *given a query Q and a tuple t, the X (and Y) values within themselves are assumed to be independent, though dependencies between the X and Y values are allowed*. More precisely, we assume limited conditional independence, i.e., $p(X \mid C)$ (resp. $p(Y \mid C)$) may be written as $\prod_{x \in X} p(x \mid C)$ (resp. $\prod_{y \in Y} p(y \mid C)$) where C is any condition that only involves Y values (resp. X values), R, or D.

While this assumption is patently false in many cases (for instance, in the example in Section 4.2 this assumes that there is no dependency between homes in Kirkland and high-priced homes), nevertheless the remaining dependencies that we do leverage, i.e., between the specified and unspecified values, prove to be significant for ranking. Moreover, as we shall show in Section 5, the resulting simplified functional form of the ranking function enables the efficient adaptation of known Top-K algorithms through novel data structuring techniques.

We continue the derivation of the score of a tuple under the above assumptions:

$$Score\,(t) \propto \prod_{y \in Y} \frac{p(y \mid R)}{p(y \mid X, D)} = \prod_{y \in Y} \frac{p(y \mid R)}{\frac{p(X, D \mid y)\,p(y)}{p(X, D)}}$$

$$\propto \prod_{y \in Y} \frac{p(y \mid R)}{p(X, D \mid y)\,p(y)}$$

$$= \prod_{y \in Y} \frac{p(y \mid R)}{p(D \mid y)\,p(X \mid y, D)\,p(y)}$$

$$= \prod_{y \in Y} \frac{p(y \mid R)}{p(D \mid y)\,p(y)} \prod_{y \in Y} \prod_{x \in X} \frac{1}{p(x \mid y, D)}$$

$$\propto \prod_{y \in Y} \frac{p(y \mid R)}{\frac{p(D \mid y)\,p(y)}{p(D)}} \prod_{y \in Y} \prod_{x \in X} \frac{1}{p(x \mid y, D)}$$

This simplifies to

$$Score(t) \propto \prod_{y \in Y} \frac{p(y \mid R)}{p(y \mid D)} \prod_{y \in Y} \prod_{x \in X} \frac{1}{p(x \mid y, D)} \qquad (2)$$

Although Equation 2 represents a simplification over Equation 1, it is still not directly computable, as R is unknown. We discuss how to estimate the quantities $p(y \mid R)$ next.

4.2.2 Workload-Based Estimation of $p(y|R)$

Estimating the quantities $p(y \mid R)$ requires knowledge of R, which is unknown at query time. The usual technique for estimating R in IR is through user feedback (relevance feedback) at query time, or through other forms of training. In our case, we provide an automated approach that leverages available *workload information* for estimating $p(y \mid R)$.

We assume that we have at our disposal a workload W, i.e., a collection of ranking queries that have been executed on our system in the past. We first provide some intuition of how we intend to use the workload in ranking. Consider the example in Section 4.2 where a user has requested for high-priced homes in Kirkland. The workload may perhaps reveal that, in the past a large fraction of users that had requested for high-priced homes in Kirkland *had also requested for waterfront views*. Thus for such users, it is desirable to rank homes with waterfront views over homes without such views.

We note that this dependency information may not be derivable from the data alone, as a majority of such homes may not have waterfront views (i.e., data dependencies do not indicate user preferences as workload dependencies do). Of course, the other option is for a domain expert (or even the user) to provide this information (and in fact, as we shall discuss later, our ranking architecture is generic enough to allow further customization by human experts).

More generally, the workload W is represented as a set of "tuples", where each tuple represents a query and is a vector containing the corresponding values of the specified attributes. Consider an incoming query Q which specifies a set X of attribute values. *We approximate R as all query "tuples" in W that also request for X.* This approximation is novel to this paper, i.e., that all properties of the set of relevant tuples R can be obtained by only examining the subset of the workload that contains queries that also request for X. So for a query such as "City=Kirkland and Price=High", we look at the workload in determining what such users have also requested for often in the past.

We can thus write, for query Q, with specified attribute set X, $p(y \mid R)$ as $p(y \mid X, W)$. Making this substitution in Equation 2, we get

$$Score\,(t) \propto \prod_{y \in Y} \frac{p(y \mid X, W)}{p(y \mid D)} \prod_{y \in Y} \prod_{x \in X} \frac{1}{p(x \mid y, D)}$$

$$= \prod_{y \in Y} \frac{\frac{p(X, W \mid y)\,p(y)}{p(X, W)}}{p(y \mid D)} \prod_{y \in Y} \prod_{x \in X} \frac{1}{p(x \mid y, D)}$$

$$\propto \prod_{y \in Y} \frac{p(X, W \mid y)\,p(y)}{p(y \mid D)} \prod_{y \in Y} \prod_{x \in X} \frac{1}{p(x \mid y, D)}$$

$$= \prod_{y \in Y} \frac{p(W \mid y)\,p(X \mid W, y)\,p(y)}{p(y \mid D)} \prod_{y \in Y} \prod_{x \in X} \frac{1}{p(x \mid y, D)}$$

$$\propto \prod_{y \in Y} \frac{p(y \mid W) \prod_{x \in X} p(x \mid y, W)}{p(y \mid D)} \prod_{y \in Y} \prod_{x \in X} \frac{1}{p(x \mid y, D)}$$

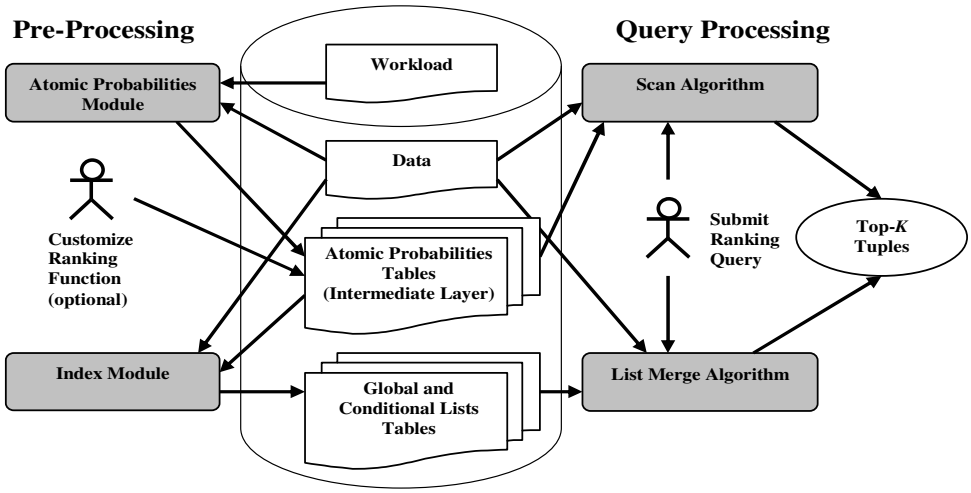

Figure 2: Detailed Architecture of Ranking System

This can be finally rewritten as:

$$Score(t) \propto \prod_{y \in Y} \frac{p(y \mid W)}{p(y \mid D)} \prod_{y \in Y} \prod_{x \in X} \frac{p(x \mid y, W)}{p(x \mid y, D)} \qquad (3)$$

Equation 3 is the final ranking formula that we use in the rest of this paper. Note that unlike Equation 2, we have effectively eliminated R from the formula, and are only left with having to compute quantities such as $p(y \mid W)$, $p(x \mid y, W)$, $p(y \mid D)$, and $p(x \mid y, D)$. In fact, these are the "atomic" numerical quantities referred to at various places earlier in the paper.

Also note that the score in Equation 3 is composed of two large factors. The first factor may be considered as the *global* part of the score, while the second factor may be considered as the *conditional* part of the score. Thus, in the example in Section 4.2, the first part measures the global importance of unspecified values such as waterfront, greenbelt and street views, while the second part measures the dependencies between these values and specified values "City=Kirkland" and "Price=High".

4.3 Computing the Atomic Probabilities

Our strategy is to pre-compute each of the atomic quantities for all distinct values in the database. The quantities $p(y \mid W)$ and $p(y \mid D)$ are simply the relative frequencies of each distinct value y in the workload and database, respectively (the latter is similar to IDF, or the *inverse document frequency* concept in IR), while the quantities $p(x \mid y, W)$ and $p(x \mid y, D)$ may be estimated by computing the confidences of pair-wise *association rules* [3] in the workload and database, respectively. Once this pre-computation has been completed, we store these quantities as auxiliary tables in the intermediate knowledge representation layer. At query time, the necessary quantities may be retrieved and appropriately composed for performing the rankings. Further details of the implementation are discussed in Section 5.

While the above is an automated approach based on workload analysis, it is possible that sometimes the workload may be insufficient and/or unreliable. In such instances, it may be necessary for domain experts to be able to tune the ranking function to make it more suitable for the application at hand.

5. Implementation

In this section we discuss the implementation of our database ranking system. Figure 2 shows the detailed architecture, including the pre-processing and query processing components as well as their sub-modules. We discuss several novel data structures and algorithms that were necessary for good performance of our system.

5.1 Pre-Processing

This component is divided into several modules. First, the *Atomic Probabilities Module* computes the quantities $p(y \mid W)$, $p(y \mid D)$, $p(x \mid y, W)$, and $p(x \mid y, D)$ for all distinct values x and y. These quantities are computed by scanning the workload and data, respectively (while the latter two quantities can be computed by running a general association rule mining algorithm such as [3] on the workload and data, we instead chose to directly compute all pair-wise co-occurrence frequencies by a single scan of the workload and data respectively). The observed probabilities are then smoothened using the Bayesian *m-estimate* method [10].

These atomic probabilities are stored as database tables in the intermediate knowledge representation layer,

with appropriate indexes to enable easy retrieval. In particular, $p(y|W)$ and $p(y|D)$ are respectively stored in two tables, each with columns {AttName, AttVal, Prob} and with a composite B+ tree index on (AttName, AttVal), while $p(x|y,W)$ and $p(x|y,D)$ respectively are stored in two tables, each with columns {AttNameLeft, AttValLeft, AttNameRight, AttValRight, Prob} and with a composite B+ tree index on (AttNameLeft, AttValLeft, AttNameRight, AttValRight). These atomic quantities can be further customized by human experts if necessary.

This intermediate layer now contains enough information for computing the ranking function, and a naïve query processing algorithm (henceforth referred to as the *Scan* algorithm) can indeed be designed, which, for any query, first selects the tuples that satisfy the query condition, then scans and computes the score for each such tuple using the information in this intermediate layer, and finally returns the Top-*K* tuples. However, such an approach can be inefficient for the Many-Answers problem, since the number of tuples satisfying the query condition can be very large. At the other extreme, we could pre-compute the Top-*K* tuples *for all possible queries* (i.e., for all possible sets of values *X*), and at query time, simply return the appropriate result set. Of course, due to the combinatorial explosion, this is infeasible in practice. We thus pose the question: how can we appropriately trade off between pre-processing and query processing, i.e., what additional yet reasonable pre-computations are possible that can enable faster query-processing algorithms than Scan?

The high-level intuition behind our approach to the above problem is as follows. Instead of pre-computing the Top-*K* tuples for all possible queries, we pre-compute ranked lists of the tuples for all possible "atomic" queries - each distinct value *x* in the table defines an atomic query *Qx* that specifies the single value {*x*}. Then at query time, given an actual query that specifies a set of values *X*, we "merge" the ranked lists corresponding to each *x* in *X* to compute the final Top-*K* tuples.

Of course, for this high-level idea to work, the main challenge is to be able to perform the merging without having to scan any of the ranked lists in its entirety. One idea would be to try and adapt well-known Top-*K* algorithms such as the *Threshold Algorithm* (*TA*) and its derivatives [9, 15, 16, 19, 25] for this problem. However, it is not immediately obvious how a feasible adaptation can be easily accomplished. For example, it is especially critical to keep the number of *sorted streams* (an access mechanism required by TA) small, as it is well-known that TA's performance rapidly deteriorates as this number increases. Upon examination of our ranking function in Equation 3 (which involves *all* attribute values of the tuple, and not just the specified values), the number of sorted streams in any naïve adaptation of TA would depend on the total number of attributes in the database, which would cause major performance problems.

In what follows, we show how to pre-compute data structures that indeed enable us to *efficiently* adapt TA for our problem. At query time we do a TA-like merging of several ranked lists (i.e. sorted streams). However, the required number of sorted streams depends only on *s* and not on *m* (*s* is the number of specified attribute values in the query while *m* is the total number of attributes in the database, see Section 3.1). We emphasize that such a merge operation is only made possible due to the specific functional form of our ranking function resulting from our limited independence assumptions as discussed in Section 4.2.1. It is unlikely that TA can be adapted, at least in a feasible manner, for ranking functions that rely on more comprehensive dependency models of the data.

We next give the details of these data structures. They are pre-computed by the *Index Module* of the pre-processing component. This module (see Figure 3 for the algorithm) takes as inputs the association rules and the database, and for every distinct value *x*, creates two lists C_x and G_x, each containing the tuple-ids of all data tuples that contain *x*, ordered in specific ways. These two lists are defined as follows:

1. **Conditional List** C_x**:** This list consists of pairs of the form <TID, CondScore>, ordered by descending CondScore, where TID is the tuple-id of a tuple *t* that contains *x* and
$$CondScore = \prod_{z \in t} \frac{p(x|z,W)}{p(x|z,D)}$$
where *z* ranges over all attribute values of *t*.
2. **Global List** G_x: This list consists of pairs of the form <TID, GlobScore>, ordered by descending GlobScore, where TID is the tuple-id of a tuple *t* that contains *x* and
$$GlobScore = \prod_{z \in t} \frac{p(z|W)}{p(z|D)}$$

These lists enable efficient computation of the score of a tuple *t* for any query as follows: given query *Q* specifying conditions for a set of attribute values, say $X = \{x_1,..,x_s\}$, at query time we retrieve and multiply the scores of *t* in the lists $C_{x1},\ldots,C_{xs}$ and in one of $G_{x1},\ldots,G_{xs}$. This requires only *s*+1multiplications and results in a score[2] that is proportional to the actual score. Clearly this is more efficient than computing the score "from scratch" by retrieving the relevant atomic probabilities from the intermediate layer and composing them appropriately.

We need to enable two kinds of access operations efficiently on these lists. First, given a value *x*, it should be possible to perform a GetNextTID operation on lists C_x and G_x in constant time, i.e., the tuple-ids in the lists

[2] This score is proportional, but not equal, to the actual score because it contains extra factors of the form $p(x|z,W)/p(x|z,D)$ where $z \in X$. However, these extra factors are common to all selected tuples, hence the rank order is unchanged.

should be efficiently retrievable one-by-one in order of decreasing score. This corresponds to the sorted stream access of TA. Second, it should be possible to perform random access on the lists, i.e., given a TID, the corresponding score (CondScore or GlobScore) should be retrievable in constant time. To enable these operations efficiently, we materialize these lists as database tables – all the conditional lists are maintained in one table called *CondList* (with columns {AttName, AttVal, TID, CondScore}) while all the global lists are maintained in another table called *GlobList* (with columns {AttName, AttVal, TID, GlobScore}). The table have composite B+ tree indices on (AttName, AttVal, CondScore) and (AttName, AttVal, GlobScore) respectively. This enables efficient performance of both access operations. Further details of how these data structures and their access methods are used in query processing are discussed in Section 5.2.

Index Module
Input: Data table, atomic probabilities tables
Output: Conditional and global lists

FOR EACH distinct value x of database DO
  $C_x = G_x = \{\}$
  FOR EACH tuple t containing x with tuple-id = TID DO
    $CondScore = \prod_{z \in t} \frac{p(x \mid z, W)}{p(x \mid z, D)}$
    Add <TID, CondScore> to C_x
    $GlobScore = \prod_{z \in t} \frac{p(z \mid W)}{p(z \mid D)}$
    Add <TID, GlobScore> to G_x
  END FOR
  Sort C_x and G_x by decreasing *CondScore* and *GlobScore* resp.
END FOR

Figure 3: The Index Module

5.2 Query Procesing Component

In this subsection we describe the query processing component. The naïve *Scan* algorithm has already been described in Section 5.1, so our focus here is on the alternate *List Merge* algorithm (see Figure 4). This is an adaptation of TA, whose efficiency crucially depends on the data structures pre-computed by the Index Module.

The *List Merge* algorithm operates as follows. Given a query Q specifying conditions for a set $X = \{x_1,..,x_s\}$of attributes, we execute TA on the following s+1 lists: $C_{x1},\ldots,C_{xs}$, and G_{xb}, where G_{xb} is the shortest list among $G_{x1},\ldots,G_{xs}$ (in principle, any list from $G_{x1},\ldots,G_{xs}$ would do, but the shortest list is likely to be more efficient). During each iteration, the TID with the next largest score is retrieved from each list using sorted access. Its score in every other list is retrieved via random access, and all these retrieved scores are multiplied together, resulting in the final score of the tuple (which, as mentioned in Section 5.1, is proportional to the actual score derived in Equation 3). The termination criterion guarantees that no more GetNextTID operations will be needed on any of the lists. This is accomplished by maintaining an array T which contains the last scores read from all the lists at any point in time by GetNextTID operations. The product of the scores in T represents the score of the very best tuple we can hope to find in the data that is yet to be seen. If this value is no more than the tuple in the Top-K buffer with the smallest score, the algorithm successfully terminates.

List Merge Algorithm
Input: Query, data table, global and conditional lists
Output: Top-K tuples

Let G_{xb} be the shortest list among $G_{x1},\ldots,G_{xs}$
Let $B = \{\}$ be a buffer that can hold K tuples ordered by score
Let T be an array of size s+1 storing the last score from each list
Initialize B to empty
REPEAT
  FOR EACH list L in $C_{x1},\ldots,C_{xs}$, and G_{xb} DO
    TID = GetNextTID(L)
    Update T with score of TID in L
    Get score of TID from other lists via random access
    IF all lists contain TID THEN
      Compute *Score*(TID) by multiplying retrieved scores
      Insert <TID, *Score*(TID)> in the correct position in B
    END IF
  END FOR
UNTIL $B[K].Score \geq \prod_{i=1}^{s+1} T[i]$
RETURN B

Figure 4: The List Merge Algorithm

5.2.1 Limited Available Space

So far we have assumed that there is enough space available to build the conditional and global lists. A simple analysis indicates that the space consumed by these lists is O(mn) bytes (m is the number of attributes and n the number of tuples of the database table). However, there may be applications where space is an expensive resource (e.g., when lists should preferably be held in memory and compete for that space or even for space in the processor cache hierarchy). We show that in such cases, we can store only a subset of the lists at pre-processing time, at the expense of an increase in the query processing time.

Determining which lists to retain/omit at pre-processing time may be accomplished by analyzing the workload. A simple solution is to store the conditional lists C_x and the corresponding global lists G_x only for those attribute values x that occur most frequently in the workload. At query time, since the lists of some of the specified attributes may be missing, the intuitive idea is to probe the intermediate knowledge representation layer (where the "relatively raw" data is maintained, i.e., the

atomic probabilities) and directly compute the missing information. More specifically, we use a modification of TA described in [9], where not all sources have sorted stream access.

6. Experiments

In this section we report on the results of an experimental evaluation of our ranking method as well as some of the competitors. We evaluated both the *quality* of the rankings obtained, as well as the *performance* of the various approaches. We mention at the outset that preparing an experimental setup for testing ranking quality was extremely challenging, as unlike IR, there are no standard benchmarks available, and we had to conduct user studies to evaluate the rankings produced by the various algorithms.

For our evaluation, we use real datasets from two different domains. The first domain was the MSN HomeAdvisor database (http://houseandhome.msn.com/), from which we prepared a table of homes for sale in the US, with attributes such as Price, Year, City, Bedrooms, Bathrooms, Sqft, Garage, etc. (we converted numerical attributes into categorical ones by discretizing them into meaningful ranges). The original database table also had a text column called Remarks, which contained descriptive information about the home. From this column, we extracted additional Boolean attributes such as Fireplace, View, Pool, etc. To evaluate the role of the size of the database, we also performed experiments on a subset of the HomeAdvisor database, consisting only of homes sold in the Seattle area.

The second domain was the Internet Movie Database (http://www.imdb.com), from which we prepared a table of movies, with attributes such as Title, Year, Genre, Director, FirstActor, SecondActor, Certificate, Sound, Color, etc. (we discretized numerical attributes such as Year into meaningful ranges). We first selected a set of movies by the 30 most prolific actors for our experiments. From this we removed the 250 most well-known movies, as we did not wish our users to be biased with information they already might know about these movies, especially information that is not captured by the attributes that we had selected for our experiments.

The sizes of the various (single-table) datasets used in our experiments are shown in Figure 5. The quality experiments were conducted on the Seattle Homes and Movies tables, while the performance experiments were conducted on the Seattle Homes and the US Homes tables – we omitted performance experiments on the Movies table on account of its small size. We used Microsoft SQL Server 2000 RDBMS on a P4 2.8-GHz PC with 1 GB of RAM for our experiments. We implemented all algorithms in C#, and connected to the RDBMS through DAO. We created single-attribute indices on all table attributes, to be used during the selection phase of the Scan algorithm. Note that these indices are not used by the List Merge algorithm.

Table	NumTuples	Database Size (MB)
Seattle Homes	17463	1.936
US Homes	1380762	140.432
Movies	1446	Less than 1

Figure 5: Sizes of Datasets

6.1 Quality Experiments

We evaluated the quality of two different ranking methods: (a) our ranking method, henceforth referred to as *Conditional* (b) the ranking method described in [2], henceforth known as *Global*. This evaluation was accomplished using surveys involving 14 employees of Microsoft Research.

For the Seattle Homes table, we first created several different profiles of home buyers, e.g., young dual-income couples, singles, middle-class family who like to live in the suburbs, rich retirees, etc. Then, we collected a workload from our users by requesting them to behave like these home buyers and post conjunctive queries against the database - e.g., a middle-class homebuyer with children looking for a suburban home would post a typical query such as "Bedrooms=4 and Price=Moderate and SchoolDistrict=Excellent". We collected several hundred queries by this process, each typically specifying 2-4 attributes. We then trained our ranking algorithm on this workload.

We prepared a similar experimental setup for the Movies table. We first created several different profiles of moviegoers, e.g., teenage males wishing to see action thrillers, people interested in comedies from the 80s, etc. We disallowed users from specifying the movie title in the queries, as the title is a key of the table. As with homes, here too we collected several hundred workload queries, and trained our ranking algorithm on this workload.

We first describe a few sample results informally, and then present a more formal evaluation of our rankings.

6.1.1 Examples of Ranking Results

For the Seattle Homes dataset, both Conditional as well as Global produced rankings that were intuitive and reasonable. There were interesting examples where Conditional produced rankings that were superior to Global. For example, for a query with condition "City=Seattle and Bedroom=1", Conditional ranked condos with garages the highest. Intuitively, this is because private parking in downtown is usually very scarce, and condos with garages are highly sought after. However, Global was unable to recognize the importance of garages for this class of homebuyers, because most users (i.e., over the entire workload) do not explicitly request for garages since most homes have garages. As

another example, for a query such as "Bedrooms=4 and City=Kirkland and Price=Expensive", Conditional ranked homes with waterfront views the highest, whereas Global ranked homes in good school districts the highest. This is as expected, because for very rich homebuyers a waterfront view is perhaps a more desirable feature than a good school district, even though the latter may be globally more popular across all homebuyers.

Likewise, for the Movies dataset, Conditional often produced rankings that were superior to Global. For example, for a query such as "Year=1980s and Genre=Thriller", Conditional ranked movies such as "Indiana Jones and the Temple of Doom" higher than "Commando", because the workload indicated that Harrison Ford was a better known actor than Arnold Schwarzenegger during that era, although the latter actor was globally more popular over the entire workload.

6.1.2 Ranking Evaluation

We now present a more formal evaluation of the ranking quality produced by the ranking algorithms. We conducted two surveys; the first compared the rankings against user rankings using standard precision/recall metrics, while the second was a simpler survey that asked users to rate which algorithm's rankings they preferred.

Average Precision: Since requiring users to rank the entire database for each query would have been extremely tedious, we used the following strategy. For each dataset, we generate 5 test queries. For each test query Q_i we generated a set H_i of 30 tuples likely to contain a good mix of relevant and irrelevant tuples to the query. We did this by mixing the Top-10 results of both the Conditional and Global ranking algorithms, removing ties, and adding a few randomly selected tuples. Finally, we presented the queries along with their corresponding H_i's (with tuples randomly permuted) to each user in our study. Each user's responsibility was to mark 10 tuples in H_i as most relevant to the query Q_i. We then measured how closely the 10 tuples marked as relevant by the user (i.e., the "ground truth") matched the 10 tuples returned by each algorithm.

	Seattle Homes		Movies	
	COND	**GLOB**	**COND**	**GLOB**
Q1	0.70	0.26	0.48	0.35
Q2	0.76	0.62	0.53	0.43
Q3	0.90	0.54	0.58	0.20
Q4	0.84	0.32	0.45	0.48
Q5	0.44	0.48	0.43	0.40

Figure 6: Average Precision

We used the formal Precision/Recall metrics to measure this overlap. Precision is the ratio of the number of retrieved tuples that are relevant, to the total number of retrieved tuples, while Recall is the fraction of the number of relevant tuples that are relevant, to the total number of relevant tuples (see [4]). In our case, the total number of relevant tuples is 10, so Precision and Recall are equal. The average precision of the ranking methods for each dataset are shown in Figure 6 (the queries are, of course, different for each dataset). As can be seen, the quality of Conditional's ranking was usually superior to Global's, more so for the Seattle Homes dataset.

User Preference of Rankings: In this experiment, for the Seattle Homes as well as the Movies dataset, users were given the Top-5 results of the two ranking methods for 5 queries (different from the previous survey), and were asked to choose which rankings they preferred. Figures 7 and 8 show, for each query and each algorithm, the fraction of users that preferred the rankings of the algorithm. The results of the above experiments show that Conditional generally produces rankings of higher quality compared to Global, especially for the Seattle Homes dataset. While these experiments indicate that our ranking approach has promise, we caution that much larger-scale user studies are necessary to conclusively establish findings of this nature.

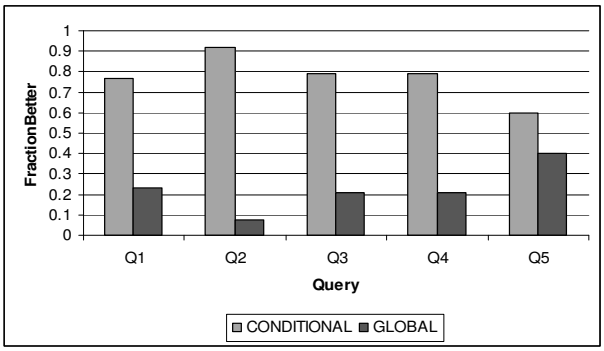

Figure 7: Fraction of Users Preferring Each Algorithm for Seattle Homes Dataset

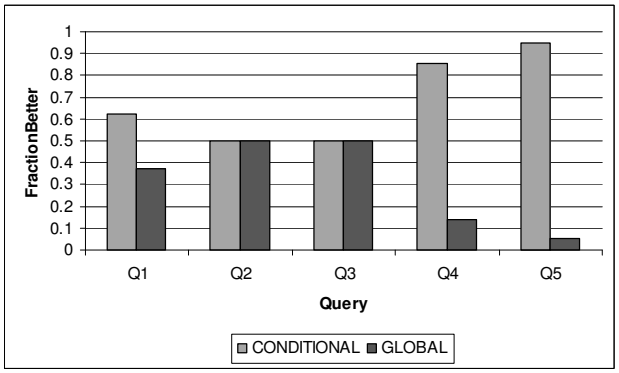

Figure 8: Fraction of Users Preferring Each Algorithm for Movies Dataset

6.2 Performance Experiments

In this subsection we report on experiments that compared the performance of the various implementations of the Conditional algorithm: List Merge, its space-saving variants, and Scan. We do not report on the corresponding implementations of Global as they had similar performance. We used the Seattle Homes and US Homes datasets for these experiments.

Preprocessing Time and Space: Since the preprocessing performance of the List Merge algorithm is dominated by the Index Module, we omit reporting results for the Atomic Probabilities Module. Figure 9 shows the space and time required to build *all* the conditional and global lists. The time and space scale linearly with table size, which is expected. Notice that the space consumed by the lists is three times the size of the data table. While this may seemingly appear excessive, note that a fair comparison would be against a Scan algorithm that has B+ tree indices built on *all* attributes (so that all kinds of selections can be performed efficiently). In such a case, the total space consumed by these B+ tree indices would rival the space consumed by these lists.

Datasets	Lists Building Time	Lists Size
Seattle Homes	1500 msec	7.8 MB
US Homes	80000 msec	457.6 MB

Figure 9: Time and Space Consumed by Index Module

If space is a critical issue, we can adopt the space saving variation of the List Merge algorithm as discussed in Section 5.2.1. We report on this next.

Space Saving Variations: In this experiment we show how the performance of the algorithms changes when only a subset of the set of global and conditional lists are stored. Recall from Section 5.2.1 that we only retain lists for the values of the frequently occurring attributes in the workload. For this experiment we consider Top-10 queries with selection conditions that specify two attributes (queries generated by randomly picking a pair of attributes and a domain value for each attribute), and measure their execution times. The compared algorithms are:

- LM: List Merge with all lists available
- LMM: List Merge where lists for one of the two specified attributes are missing, halving space
- Scan

Figure 10 shows the execution times of the queries over the Seattle Homes database as a function of the total number of tuples that satisfy the selection condition. The times are averaged over 10 queries.

We first note that LM is extremely fast when compared to the other algorithms (its times are less than one second for each run, consequently its graph is almost along the *x*-axis). This is to be expected as most of the computations have been accomplished at pre-processing time. The performance of Scan degrades when the total number of selected tuples increases, because the scores of more tuples need to be calculated at runtime. In contrast, the performance of LM and LMM actually improves slightly. This interesting phenomenon occurs because if more tuples satisfy the selection condition, smaller prefixes of the lists need to be read and merged before the stopping condition is reached.

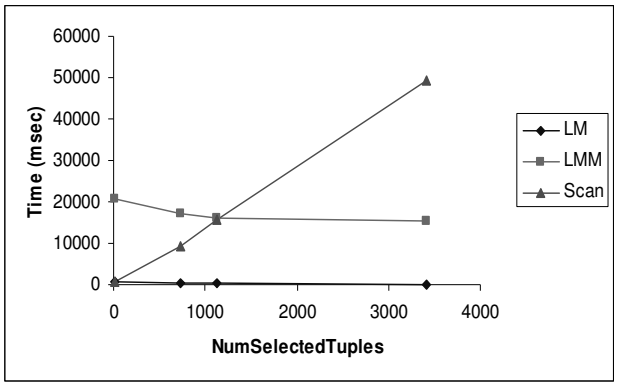

Figure 10: Execution Times of Different Variations of List Merge and Scan for Seattle Homes Dataset

Thus, List Merge and its variations are preferable if the number of tuples satisfying the query condition is large (which is exactly the situation we are interested in, i.e., the Many-Answers problem). This conclusion was reconfirmed when we repeated the experiment with LM and Scan on the much larger US Homes dataset with queries satisfying many more tuples (see Figure 11).

NumSelected Tuples	LM Time (msec)	Scan Time (msec)
350	800	6515
2000	700	39234
5000	600	115282
30000	550	566516
80000	500	3806531

Figure 11: Execution Times of List Merge and Scan for US Homes Dataset

Varying Number of Specified Attributes: Figure 12 shows how the query processing performance of the algorithms varies with the number of attributes specified in the selection conditions of the queries over the US Homes database (the results for the other databases are similar). The times are averaged over 10 Top-10 queries. Note that the times increase sharply for both algorithms with the number of specified attributes. The LM algorithm becomes slower because more lists need to be merged, which delays the termination condition. The Scan algorithm becomes slower because the selection time

increases with the number of specified attributes. This experiment demonstrates the criticality of keeping the number of sorted streams small in our adaptation of TA.

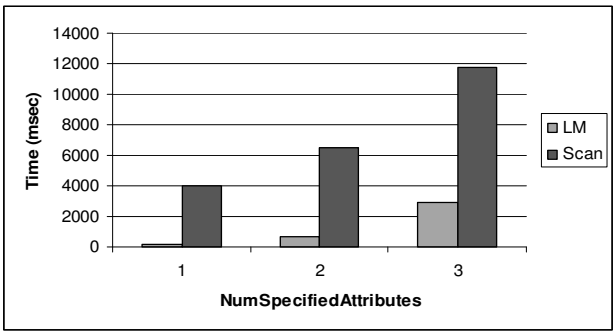

Figure 12: Varying Number of Specified Attributes for US Homes Dataset

7. Conclusions

We proposed a completely automated approach for the Many-Answers Problem which leverages data and workload statistics and correlations. Our ranking functions are based upon the probabilistic IR models, judiciously adapted for structured data. We presented results of preliminary experiments which demonstrate the efficiency as well as the quality of our ranking system.

Our work brings forth several intriguing open problems. For example, many relational databases contain text columns in addition to numeric and categorical columns. It would be interesting to see whether correlations between text and non-text data can be leveraged in a meaningful way for ranking. Finally, comprehensive quality benchmarks for database ranking need to be established. This would provide future researchers with a more unified and systematic basis for evaluating their retrieval algorithms.

References

[1] S. Agrawal, S. Chaudhuri G. Das. DBXplorer: A System for Keyword Based Search over Relational Databases. ICDE 2002.
[2] S. Agrawal, S. Chaudhuri, G. Das and A. Gionis. Automated Ranking of Database Query Results. CIDR, 2003.
[3] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen and A. I. Verkamo. Fast Discovery of Association Rules. Advances in Knowledge Discovery and Data Mining, 1995.
[4] R. Baeza-Yates and B. Ribeiro-Neto. Modern Information Retrieval. ACM Press, 1999.
[5] G. Bhalotia, C. Nakhe, A. Hulgeri, S. Chakrabarti and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. ICDE 2002.
[6] H. M. Blanken, T. Grabs, H.-J. Schek, R. Schenkel, G. Weikum (Eds.): Intelligent Search on XML Data: Applications, Languages, Models, Implementations, and Benchmarks. LNCS 2818 Springer 2003.
[7] J. Breese, D. Heckerman and C. Kadie. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. 14th Conference on Uncertainty in Artificial Intelligence, 1998.
[8] N. Bruno, L. Gravano, and S. Chaudhuri.Top-K Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. ACM TODS, 2002.
[9] N. Bruno, L. Gravano, A. Marian. Evaluating Top-K Queries over Web-Accessible Databases. ICDE 2002.
10] B. Cestnik. Estimating Probabilities: A Crucial Task in Machine Learning, European Conf. in AI, 1990.
[11] K. Chakrabarti, K. Porkaew and S. Mehrotra. Efficient Query Ref. in Multimedia Databases. ICDE 2000.
[12] W. Cohen. Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity. SIGMOD, 1998.
[13] W. Cohen. Providing Database-like Access to the Web Using Queries Based on Textual Similarity. SIGMOD 1998.
[14] W.B. Croft, J. Lafferty. Language Modeling for Information Retrieval. Kluwer 2003.
[15] R. Fagin. Fuzzy Queries in Multimedia Database Systems. PODS 1998.
[16] R. Fagin, A. Lotem and M. Naor. Optimal Aggregation Algorithms for Middleware. PODS 2001.
[17] N. Fuhr. A Probabilistic Framework for Vague Queries and Imprecise Information in Databases. VLDB 1990.
[18] N. Fuhr. A Probabilistic Relational Model for the Integration of IR and Databases. ACM SIGIR Conference on Research and Development in Information Retrieval, 1993.
[19] U. Güntzer, W.-T. Balke, W. Kießling: Optimizing Multi-Feature Queries for Image Databases. VLDB 2000.
[20] V. Hristidis,Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. VLDB 2002.
[21] W. Kießling. Foundations of Preferences in Database Systems. VLDB 2002.
[22] D. Kossmann, F. Ramsak, S. Rost: Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. VLDB 2002.
[23] A. Motro. VAGUE: A User Interface to Relational Databases that Permits Vague Queries. TOIS 1988, 187-214.
[24] Z. Nazeri, E. Bloedorn and P. Ostwald. Experiences in Mining Aviation Safety Data. SIGMOD 2001.
[25] S. Nepal, M. V. Ramakrishna: Query Processing Issues in Image (Multimedia) Databases. ICDE 1999.
[26] M. Ortega-Binderberger, K. Chakrabarti and S. Mehrotra. An Approach to Integrating Query Refinement in SQL, EDBT 2002, 15-33.
[27] Y. Rui, T. S. Huang and S. Merhotra. Content-Based Image Retrieval with Relevance Feedback in MARS. IEEE Conf. on Image Processing, 1997.
[28] K. Sparck Jones, S. Walker, S. E. Robertson: A Probabilistic Model of Information Retrieval: Development and Comparative Experiments - Part 1. Inf. Process. Manage. 36(6): 779-808, 2000.
[29] K. Sparck Jones, S. Walker, S. E. Robertson: A Probabilistic Model of Information Retrieval: Development and Comparative Experiments - Part 2. Inf. Process. Manage. 36(6): 809-840, 2000.
[30] J. Whittaker. Graphical Models in Applied Multivariate Statistics. Wiley, 1990.
[31] L. Wimmers, L. M. Haas , M T. Roth and C. Braendli. Using Fagin's Algorithm for Merging Ranked Results in Multimedia Middleware. CoopIS 1999.
[32] L. Wu, C. Faloutsos, K. Sycara and T. Payne. FALCON: Feedback Adaptive Loop for Content-Based Retrieval. VLDB 2000.