Probabilistic Information Retrieval Approach for Ranking of Database Query Results

SURAJIT CHAUDHURI
Microsoft Research
GAUTAM DAS
University of Texas at Arlington
VAGELIS HRISTIDIS
Florida International University
and
GERHARD WEIKUM
Max Planck Institut fur Informatik

We investigate the problem of ranking the answers to a database query when many tuples are returned. In particular, we present methodologies to tackle the problem for conjunctive and range queries, by adapting and applying principles of probabilistic models from information retrieval for structured data. Our solution is domain independent and leverages data and workload statistics and correlations. We evaluate the quality of our approach with a user survey on a real database. Furthermore, we present and experimentally evaluate algorithms to efficiently retrieve the top ranked results, which demonstrate the feasibility of our ranking system.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; H.2.4 [**Database Management**]: Systems

General Terms: Experimentation, Performance, Theory

Additional Key Words and Phrases: Probabilistic information retrieval, user survey, experimentation, indexing, automatic ranking, relational queries, workload

V. Hristidis has been partially supported by NSF grant IIS-0534530.
Part of this work was performed while G. Das was a researcher, V. Hristidis was an intern, and G. Weikum was a visitor at Microsoft Research.
A conference version of this article titled "Probabilistic Ranking of Database Query Results." appeared in *Proceedings of VLDB 2004*.
Authors' current addresses: S. Chaudhuri, Microsoft Research, One Microsoft Way, Redmond, WA 98052; email: surajitc@microsoft.com; G. Das, Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX 76019; email: gdas@cse.uta.edu; V. Hristidis, School of Computing and Information Sciences, Florida International University, Miami, FL 33199; email: vageli@cis.fiu.edu; G. Weikum, Max Planck Institut für Informatik, Building 46-1, Stuhlsatzbrücken 85, 66123 Saarbrücken, Germany, email: weikum@mpi-sb.mpg.de.

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 article:

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 also been investigated 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 article 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 article 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 (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" (we also consider IN queries, e.g., City IN (Seattle, Redmond)), 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"

[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.

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 boat docks. We extend our framework to also support numeric attributes (e.g., age), in addition to categorical, by exploiting state-of-the-art bucketing methods based on histograms.

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 alternative IR ranking models such as the Vector-Space model. We note that correlations are sometimes ignored in IR data—important exceptions are relevance feedback-based IR systems—because they are very difficult to capture in the very high-dimensional and sparsely populated feature spaces of text 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. Furthermore, we exploit possible functional dependencies in the database to improve the quality of the ranking.

The architecture of our ranking has a preprocessing 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 interesting precomputation techniques which reduce this complex problem to a problem efficiently solvable using top-*k* algorithms.

The rest of this article is organized as follows. In Section 2 we discuss related work. In Section 3 we define the problem. In Section 4 we discuss our approach to ranking based on probabilistic models from information retrieval, along with various extensions and special cases. 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

A preliminary version of this article appeared in Chaudhuri et al. [2004], where we presented the basic principles of using probabilistic information retrieval models to answer database queries. However, our earlier article only handled point queries (see Section 3). In this work, we show how IN and range queries can be handled and how this makes the algorithms to produce efficiently the top results more challenging (Sections 4.4.1 and 5.4). Furthermore Chaudhuri et al. [2004] focused on only categorical attributes, whereas we have a complete study of numerical attributes as well (Section 4.4.2). Chaudhuri et al. [2004] also ignored functional dependencies, which as we show can improve the quality of the results (Section 4.2.2). In this work, we also present specialized solutions for cases where no workload is available (Section 4.3.1), and no dependencies exist between attributes (Section 4.3.2). We also generalize to the case where the data resides on multiple tables (Section 4.4.3). Finally, we extend Chaudhuri et al. [2004] with a richer set of quality and performance experiments. On the quality level, we show results for IN queries and also compare them to the results of a "random" algorithm. On the performance level, we include experiments on how the number k of requested results affects the performance of the algorithms.

Ranking functions have been extensively investigated in information retrieval. The vector space model as well as probabilistic information retrieval (PIR) models [Baeza-Yates and Ribeiro-Neto 1999; Grossman and Frieder 2004; Sparck Jones et al. 2000a, 2000b] and statistical language models [Croft and Lafferty 2003; Grossman and Frieder 2004] are very successful in practice. Feedback-based IR systems (e.g., relevance feedback [Harper and Van Rijsbergen 1978], pseudorelevance feedback [Xu and Croft 1996]) are based on inferring term correlations and modeling term dependencies, which are related to our approach of inferring correlations within workloads and data. While our approach has been inspired by PIR models, we have adapted and extended them in ways unique to our situation, for example, by leveraging the structure as well as correlations present in the structured data and the database workload.

In database research, there has been significant work on ranked retrieval from a database. The early work of Motro [1988] considered vague/imprecise similarity-based querying of databases. Probabilistic databases have been addressed in Barbara et al. [1992], Cavallo and Pittarelli [1987], Dalvi and Suciu [2005], and Lakshmanan et al. [1997]. Recently, a broader view of the needs for managing uncertain data has been evolving (see, e.g., Widom [2005]).

The challenging problem of integrating databases and information retrieval systems has been addressed in a number of seminal papers [Cohen 1998a, 1998b; Fuhr 1990, 1993; Fuhr and Roelleke 1997, 1998] and has gained much attention lately Amer-Yahia et al. [2005a]. More recently, information retrieval-based approaches have been extended to XML retrieval [Amer-Yahia et al. 2005b; Chinenyanga and Kushmerick 2002; Carmel et al. 2003; Fuhr and Grossjohann 2004; Guo et al. 2003; Hristidis et al. 2003b; Lalmas and Roelleke 2004; Theobald and Weikum 2002; Theobald et al. 2005]. The articles Chakrabarti et al.[2002], Ortega-Binderberger et al. [2002], Rui et al. [1997], and Wu et al. [2000] employed relevance-feedback techniques for learning similarity in multimedia and relational databases. Our approach of leveraging workloads is motivated by and related to IR models that aim to leverage query-log information (e.g., see Radlinski and Joachims [2005] and Shen et al. [2005]). Keyword-query-based retrieval systems over databases have been proposed in Agrawal et al. [2002], Bhalotia et al. [2002], Hristidis and Papakonstantinou [2002], and Hristidis et al. [2003a]. In Kiessling [2002] and Nazeri et al. [2001], the authors proposed 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 article is Agrawal et al. [2003], 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, our article 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 Agrawal et al. [2003] only proposed a form of global score for ranking.

Ranking is also an important component in collaborative filtering research [Breese et al. 1998]. 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 article is the query processing techniques for supporting ranking. Several techniques have been previously developed in database research for the top-k problem [Bruno et al. 2002a, 2002b; Fagin 1998; Fagin et al. 2001; Wimmers et al. 1999]. We adopt the Threshold Algorithm of Fagin et al. [2001] Güntzer et al. [2000], and Nepal and Ramakrishna [1999] for our purposes, and develop interesting precomputation techniques 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 Agrawal et al. [2003].

3. PROBLEM DEFINITION

In this section, we formally define the Many-Answers Problem in ranking database query results and its different variants. We start by defining the simplest problem instance, which we later extend to more complex scenarios.

3.1 The Many-Answers Problem

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 $\cdots$ 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 $S \subseteq \{\mathrm{t}_1, \ldots, \mathrm{t}_\mathrm{n}\}$ be the answer set of Q. The *Many-Answers Problem* occurs when the query is not too selective, resulting in a large S. The focus in this article is on automatically deriving an appropriate ranking function such that only a few (say top-k) tuples can be efficiently retrieved.

3.2 The Empty-Answers Problem

If the selection condition of a query is very restrictive, it may happen that very few tuples, or even no tuples, will satisfy the condition—that is, S is empty or very small. This is known as the *Empty-Answers Problem*. In such cases, it is of interest to derive an appropriate ranking function that can also retrieve tuples that closely (though not completely) match the query condition. We do not consider the Empty-Answers Problem any further in this article.

3.3 Point Queries Versus Range/IN Queries and other Generalizations

The scenario in Section 3.1 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. *IN queries* contain selection conditions of the form " X_1 IN $(x_{1,1} \cdots x_{1,r1})$ AND $\cdots$ AND X_s IN $(x_{s,1} \cdots x_{s,rs})$." Such queries are a very convenient way of expressing alternatives in desired attribute values which are not possible to express using point queries.

Also, databases may be multitabled, and may contain a mix of categorical and numeric data. In this article, we develop techniques to handle the ranking problem for all these generalizations, though for the sake of simplicity of exposition, our focus in the earlier part of the article is on point queries over a single categorical table.

3.4 Evaluation Measures

We evaluate our ranking functions both in terms of *quality* as well as *performance*. Quality of the results produced is measured using the standard IR measures of precision and recall. We also evaluate the performance of our ranking functions, especially what time and space is necessary for preprocessing as well as for query processing.

4. RANKING FUNCTIONS: ADAPTATION OF PIR MODELS FOR STRUCTURED DATA

In this section we first review probabilistic information retrieval (PIR) techniques in IR (Section 4.1). We then show in Section 4.2 how they can be adapted for structured data for the special case of ranking the results of point queries over a single categorical table. In Section 4.3 we present two interesting special cases of these ranking functions, while in Section 4.4 we extend our techniques to handle IN queries, numeric attributes, and other generalizations.

4.1 Review of Probabilistic Information Retrieval

Much of the material of this subsection can be found in textbooks on information retrieval, such as those by Baeza-Yates and Ribeiro-Neto [1999] (see also Sparck Jones et al. [2000a; 2000b]). Probabilistic Information Retrieval (PIR) makes use of the following basic formulae 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 $\bar{R}=D-R$ 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), that is, $p(R|t)$.More formally, in probabilistic information retrieval, documents are ranked by decreasing order of their odds of relevance, defined as the following *score*:

$$Score(t) = \frac{p(R|t)}{p(\bar{R}|t)} = \frac{\frac{p(t|R)p(R)}{p(t)}}{\frac{p(t|\bar{R})p(\bar{R})}{p(t)}} \propto \frac{p(t|R)}{p(t|\bar{R})}.$$

The final simplification in the above equation follows from the fact that $p(R)$and $p(\bar{R})$are the same for every document t and thus mere constants that do not influence the ranking of documents. The main issue now is: how are these probabilities computed, given that R and $\bar{R}$are unknown at query time? The usual techniques in IR are to make some simplifying assumptions, such as estimating R through user feedback, approximating $\bar{R}$ 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,* the*Linked Dependence Model*, or the*Tree Dependence Model* [Yu and Meng 1998; Baeza-Yates and Ribeiro-Neto 1999; Grossman and Frieder 2004]).

In the next subsection we show how we adapt PIR models for structured databases, in particular for conjunctive queries over a single categorical table. Whereas the Binary Independence Model makes an independence assumption over all terms, we apply in the following a limited independence assumption, that is, we consider two dependent conjuncts, and view the atomic events of each conjunction to be independent.

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, 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 $\bar{R}$ as D, as mentioned in Section 4.1, is commonly done in IR), we get

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

where the last equality is obtained by applying Bayes' Theorem. Then, because $R \subseteq X$ (i.e., all relevant tuples have the same X values specified in the query), we obtain $P(X|Y, R) = 1$ which leads to

$$Score(t) \propto \frac{p(Y|R)}{p(Y|D)} \cdot \frac{1}{p(X|Y, D)}. \tag{1}$$

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. 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 $p(Y|X)$ 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 [Whittaker 1990]), and derive ranking functions based on these models. However, our preliminary investigations suggested that such approaches have unacceptable preprocessing and query processing costs.

Consequently, in this article 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, that is, $p(X|C)$ (respectively $p(Y|C)$) may be written as ($\prod_{x\in X} p(x|C)$respectively $\prod_{y\in Y} p(y|C)$), where C is any condition that only involves Y values (respectively X values), R, or D.

While this assumption is patently false in many cases (for instance, in the example early 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, that is, 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 a tuple's score under the above assumptions and obtain

$$\begin{aligned} Score(t) &\propto \frac{p(Y|R)}{p(Y|D)} \cdot \frac{1}{p(X|Y,D)} \\ &= \prod_{y\in Y} \frac{p(y|R)}{p(y|D)} \cdot \prod_{x\in X}\prod_{y\in Y} \frac{1}{p(x|y,D)}. \end{aligned} \tag{2}$$

4.2.2 *Presence of Functional Dependencies.* To reach Equation (2), we assumed limited conditional independence. In certain special cases such as for attributes related through *functional dependencies*, we can derive the equation without having to make this assumption. In the realtor database, an example of a functional dependency may be "Zipcode → City." Note that functional dependencies only apply to the data, since the workload does not have to satisfy them. For example, a query Q of the workload that specifies a requested zipcode may not have specified the city, and vice versa. Thus functional dependencies affect the denominator but not the numerator of Equation (2). The key property used to remove the independence assumption between attributes connected through functional dependencies is the following.

We first consider functional dependencies between attributes in Y. Assume that $y_i \to y_j$ is a functional dependency between a pair of attributes y_i, y_j in Y. This means that $\{\mathrm{t} \mid \mathrm{t}.y_i = a_i \wedge \mathrm{t}.y_j = a_j\} = \{\mathrm{t}|\mathrm{t}.y_i = a_i\}$ for all attribute values a_i, a_j. In this case an expression such as $p(y_i, y_j \mid D)$ can be simplified as$p(y_i|D)p(y_j|y_i, D) = p(y_i|D)$. More generally, the expression in Equation (1) may be simplified $\prod_{y\in Y'} \frac{1}{p(y|D)}$, where $Y' = \{y \in Y | \neg\exists y' \in Y, FD : y' \to y\}$.

Functional dependencies may also exist between attributes in X. Thus, the expression $\frac{1}{p(X|Y,D)}$ in Equation (1) may be simplified to $\prod_{y\in Y'}\prod_{x\in X'} \frac{1}{p(x|y,D)}$, where $X' = \{x \in X | \neg\exists x' \in X, FD : x' \to x\}$.

Applying these derivations to Equation (1), we get the following modification to Equation (2) (where X' and Y' are defined as above):

$$Score(t) \propto \prod_{y\in Y} p(y|R) \prod_{y\in Y'} \frac{1}{p(y|D)} \prod_{y\in Y'}\prod_{x\in X'} \frac{1}{p(x|y,D)}. \tag{3}$$

Notice that before applying the above formula, we need to first compute the transitive closure of functional dependencies, for the following reason.

Assume there are functional dependencies $x' \rightarrow y$ and $y \rightarrow x$ where $x, x' \in X$ and $y \in Y$. Then, if we do not calculate the closure of functional dependencies, there would be no $x' \in X$ with functional dependency $x' \rightarrow x$, and hence Equation (3) would be the same as Equation (2). Notice that Equations (2) and (3) are equivalent if there are no functional dependencies or the only functional dependencies (in the closure) are of the form $x \rightarrow y$ or $y \rightarrow x$, where $x \in X$ and $y \in Y$.

Although Equations (2) and (3) represent simplifications over Equation (1), they are still not directly computable, as R is unknown. We discuss how to estimate the quantities $p(y|R)$ next.

4.2.3 *Workload-Based Estimation of p(y|R).* Estimating the quantities $p(y|R)$ 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$p(y|R)$. Our approach is motivated by and related to IR models that aim to leverage query-log information (e.g., see Radlinski and Joachims [2005] and Shen et al. [2005]). For example, if the multikeyword queries "a b c d," "a b," and "a b c" constitute a (short) query log, then we could estimate p(a |c, queries) = 2/3.

We assume that we have at our disposal a workload W, that is, 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. The IR equivalent would be to have many past queries including all of the terms "Kirkland," "high-priced," and "waterfront view," and a new query "Kirkland high-priced" arrives.

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 article, that is, 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, $p(y|R)$ as $p(y|X, W)$. Making this substitution in Equation (2), we get

$$Score(X, Y) \propto \frac{P(Y|X, W)}{P(Y|D)} \cdot \frac{1}{P(X|Y, D)}.$$

Applying Bayes' rule for $P(Y|X, W)$ we get

$$P(Y|X, W) = \frac{P(X, W, Y)}{P(X, W)} = \frac{P(W) \cdot P(Y|W) \cdot P(X|Y, W)}{P(X, W)}.$$

Then by dropping the constant $\frac{P(W)}{P(X,W)}$ we get

$$Score(X, Y) \propto \frac{P(Y|W)}{P(Y|D)} \cdot \frac{P(X|Y, W)}{P(X|Y, D)} = \prod_{y\in Y} \frac{p(y|W)}{p(y|D)} \prod_{y\in Y} \prod_{x\in X} \frac{p(x|y, W)}{p(x|y, D)}. \quad (4)$$

Equation (4) is the final ranking formula, assuming no functional dependencies. If we also consider functional dependencies then we have

$$Score(X, Y) \propto \prod_{y\in Y} p(y|W) \prod_{y\in Y'} \frac{1}{p(y|D)} \prod_{y\in Y} \prod_{x\in X} p(x|y, W) \prod_{y\in Y'} \prod_{x\in X'} \frac{1}{p(x|y, D)}, \quad (5)$$

where X', Y' are defined as in Equation (3).

Note that unlike Equations (2) and (3), we have effectively eliminated R from the formulas in Equations (4) and (5), and are only left with having to compute quantities such as $p(y|W)$,$p(x|y, W)$,$p(y|D)$, and$p(x|y, D)$. In fact, these are the "atomic" numerical quantities referred to at various places earlier in this article. Also, note that Equations (4) and (5) have been derived for point queries; the formulas get more involved when we allow IN/range conditions, as discussed in Section 4.4.1.

Also note that the score in Equations (4) and (5) is composed of two large factors. The first factor (first product in Equations (4) and two first products in Equation (5)) 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 the specified values "City=Kirkland" and "Price=High."

4.2.4 *Computing the Atomic Probabilities.* This section explains how to calculate the atomic probabilities for categorical attributes. Section 4.4.2 explains how numerical attributes can be split into ranges which are then effectively treated as categorical attributes. Our strategy is to precompute each of the atomic quantities for all distinct values in the database. The quantities $p(y|W)$and $p(y|D)$ 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 $p(x|y, W)$ and $p(x|y, D)$ may be estimated by computing the confidences of pairwise *association rules* [Agrawal et al. 1995] in the workload and database, respectively.

Once this precomputation 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. That is, our framework allows both informative (e.g., set by domain expert) as well as noninformative (e.g., inferred by query workload) prior probability distributions to be used in the preference function. In this article, we focus on noninformative priors, which are inferred by the query workload and the data.

4.3 Special Cases

In this subsection we present two important special cases for which our ranking function can be further simplified: (a) ranking in the absence of workloads, and (b) ranking assuming no dependencies between attributes.

4.3.1 *Ranking Function in the Absence of a Workload.* We first consider Equation (4), which describes our ranking function assuming no functional dependencies—we shall consider Equation (5) later. So far we have assumed that there exists a workload, which is used to approximate the set R of relevant tuples. If no workload is available, then we can assume that $p(x|W)$ is the same for all distinct values x,and correspondingly$p(x \mid y, W)$ is the same for all pairs of distinct values x and y. Hence, as constants, they do not affect the ranking. Thus, Equation (4) reduces to

$$Score(t) \propto \frac{1}{p(Y|D)} \cdot \frac{1}{p(X|Y, D)} = \prod_{y \in Y} \frac{1}{p(y|D)} \prod_{y \in Y} \prod_{x \in X} \frac{1}{p(x|y, D)}. \tag{6}$$

The intuitive explanation of Equation (6) is similar to the idea of inverse document frequency (IDF) in information retrieval. In particular, the first product assigns a higher score to tuples whose unspecified attribute values y are infrequent in the database. The second product is similar to a "conditional" version of the IDF concept. That is, tuples with low correlations between the specified and the unspecified attribute values are ranked higher. This means, that tuples with infrequent combinations of values are ranked higher. For example, if the user searches for low-priced houses, then a house with high square footage is ranked high since this combination of values (low price and high square footage) is infrequent. Of course this ranking can potentially also lead to unintuitive results, for example, looking for high-priced houses may return low-square-footage ones.

Equation (6) can be extended in a straightforward manner to account for the presence of functional dependencies (similarley to the way Equation (4) was extended to Equation (5)).

4.3.2 *Ranking Function Assuming No Dependencies Between Attributes.* As mentioned in Section 4.2.1, a simpler approach to the ranking problem would be to make independence assumptions between all attributes (e.g., as is done in the binary independence model in IR). Whereas, in Section 4.2, we viewed X and Y as dependent events, we show here the special case of viewing X and Y as independent events. Then the linked independence assumption holds for both, the workload W and the database D. We obtain

$$Score(t) = \frac{p(Y\,|W)}{p(Y\,|D)} \cdot \frac{p(X\,|Y,W)}{p(X\,|Y,D)} = \frac{p(Y\,|W)}{p(Y\,|D)} \cdot \frac{p(X\,|W)}{p(X\,|D)}.$$

Here, the fraction p(X|W)/p(X|D) is constant for all query result tuples; hence:

$$Score(t) \propto \frac{p(Y\,|W)}{p(Y\,|D)} = \prod_{y\in Y} \frac{p(y\,|W)}{p(y\,|D)}. \tag{7}$$

Intuitively, the numerator describes the absolute importance of the unspecified attribute values in the workload, while the denominator resembles the IDF concept in IR. This formula is similar to the ranking formula for the Many-Answers problem developed in Agrawal et al. [2003] based on the vector-space model. The main difference between this formula and the corresponding formula in Agrawal et al. [2003] is that the latter did not have the denominator quantities, and also expressed the score in terms of logarithms. This provides formal credibility to the intuition behind the development of the algorithm in Agrawal et al. [2003].

4.4 Generalizations

In this subsection we present several important generalizations of our ranking techniques. In particular, we show how our techniques can be extended to handle IN queries, numeric attributes, and multitable databases.

4.4.1 *IN Queries.* *IN queries* are a generalization of point queries, in which selection conditions have the form "X_1 IN ($x_{1,1}\ldots x_{1,r1}$) AND ... AND X_s IN ($x_{s,1}\cdots x_{s,rs}$)". As an example, consider a query with a selection condition such as "City IN (Kirkland, Redmond) AND Price IN (High, Moderate)." This might represent the desire of a homebuyer who is interested in either moderate or high-priced homes in either Kirkland or Redmond. Such queries are a very convenient way of expressing alternatives in desired attribute values which are not possible to express using point queries.

Accommodating *IN queries* in our ranking infrastructure presents the challenge of automatically determining which of the alternatives are more relevant to the user—this knowledge can then be incorporated into a suitable ranking function. (This concept is related to work on vague/fuzzy predicates [Fuhr 1990, 1993; Fuhr and Roelleke 1997, 1998]. In our case, the objective is essentially to determine the probability function that can assign different weights to the different alternative values.)

First the ranking function derived in Equation (4) (and Equation (5)) have to be modified to allow IN conditions in the specified attributes. The complication stems from the fact that two tuples that satisfy the query condition may differ in their specific X values. In the above example, a moderate-priced home in Redmond will satisfy the query, as will an expensive home in Kirkland. However, since the specific X values of the two homes are different, this prevents us from factoring out the X as we so successfully did in the derivation of Equation (4). This requires nontrivial extensions to the execution algorithms, as shown in Section 5. Second, the existence of IN queries complicates the generation of the association rules in the workload, as we discuss later in this subsection.

4.4.1.1 *IN Conditions in the Query.* For simplicity, let us assume the case where there are no functional dependencies and the workload has point queries, but the query may have IN conditions. Later we will extend the discussion to the case where the workload also has IN conditions.

Consider a query that specifies conditions C, where C is a conjunction of IN conditions such as "City IN (Bellevue, Carnation) AND SchoolDistrict IN(Good, Excellent)." Note that we distinguish C from X; the latter are atomic values of specified attributes in a specific tuple, whereas the former refers to the query and contains a set of values for each specified attribute. Recall from Section 4.2 that

$$\begin{aligned} Score(t) &\propto \frac{p(t|R)}{p(t|D)} = \frac{p(X,Y|R)}{p(X,Y|D)} \\ &\propto \frac{p(X|R)}{p(X|D)} \frac{p(Y|X,R)}{p(Y|X,D)}. \end{aligned}$$

In what follows, we shall assume that $R = C, W$, that is, R is the set of tuples in W that specify C. This is in tune with the corresponding assumption in Section 4.2.3 for the case of point queries, and intuitively means that R is represented by all queries in the workload that also request for C. Of course, since here we are assuming that the workload only has point queries, we need to figure out how to evaluate this in a reasonable manner.

Consider the second part of the above formula for $Score(t)$, that is, $p(Y|X, R)/p(Y|X, D)$. This can be rewritten as $p(Y|X, C,W)/p(Y|X,C, D)$. Since we are considering the Many-Answers problem, if X is true, C is also true (recall that X is the set of attribute values of a result-tuple for the query-specified attributes). Thus this part of the formula can be simplified as $p(Y|X, W)/p(Y|X, D)$. Consequently, it can be further simplified in exactly the same way as the derivations described earlier for point queries, that is, in Equations (1) through (4).

Now consider the first part of the formula, $p(X|R)/p(X|D)$. Unlike the point query case, however, we cannot assume $p(X|R)/p(X|D)$ is a constant for all tuples. In what follows, we shall assume that x is a variable that varies over the set X, and c is a variable that varies over the set C. When x and c refer to the same attribute, it is clear that, if x is true, then c is also true. We have the following sequence of derivations:

$$\begin{aligned} \frac{p(X|R)}{p(X|D)} &= \frac{p(X|C,W)}{p(X|D)} = \prod_{x\in X}\frac{p(x|C,W)}{p(x|D)} \propto \prod_{x\in X}\frac{p(C,W|x)p(x)}{p(x|D)} \\ &= \prod_{x\in X}\frac{p(W|x)p(x)p(C|x,W)}{p(x|D)} \propto \prod_{x\in X}\frac{p(x|W)}{p(x|D)}\prod_{c\in C}p(c|x,W). \end{aligned}$$

Recall that we assume limited conditional independence, that is, that dependency exists only between the X and Y attributes, and not within the X attributes (recall that X and C specify the same set of attributes). Let $A(x)$ (respectively $A(c)$) refer to the attribute of x (respectively c). Then $p(c|x, W)$ is equal to $p(c|W)$ when $A(x){<>}A(c)$, and is equal to 1 otherwise. Let $c(x)$ represent the IN condition in C corresponding the attribute of x, that is, $A(c(x)) = A(x)$. Consequently, we have

$$\prod_{c\in C} p(c|x, W) = \frac{\prod_{c\in C} p(c|W)}{p(c(x)|W)}.$$

Hence, continuing with the above derivation, we have $p(X|R)/p(X|D)$ proportional to

$$\prod_{x\in X} \frac{p(x|W)\prod_{c\in C} p(c|W)}{p(x|D)p(c(x)|W)} = \left(\prod_{x\in X}\frac{p(x|W)}{p(x|D)}\right)\left(\prod_{x\in X}\frac{\prod_{c\in C} p(c|W)}{p(c(x)|W)}\right) \propto \prod_{x\in X}\frac{p(x|W)}{p(x|D)}.$$

This is the extra factor that needs to be multiplied to the score derived in Equation (4). Hence, the equivalent of Equation (4) for IN queries is

$$Score(t) \propto \prod_{z\in t}\frac{p(z|W)}{p(z|D)}\prod_{y\in Y}\prod_{x\in X}\frac{p(x|y, W)}{p(x|y, D)}. \tag{8}$$

Equation (8) differs from Equation (4) in the global part. In particular, we now need to consider *all* attribute values of each result-tuple t, because they may be different, whereas, in Equation (4), only the unspecified values of t were used for the global part. Notice that Equation (8) can be used for point queries as well since in this case the specified values of t are common for all result-tuples and hence would only multiply the score by a common factor. However, as we explain in Section 5.4, it is more complicated to efficiently evaluate Equation (8) for IN queries than for point queries because of the fact that all result-tuples share the same specified (X) values in point queries.

We note that Equation (8) can be generalized in a straightforward manner to allow for the presence of functional dependencies.

4.1.1.2 *IN Conditions in the Workload.* We had assumed above that the query at runtime was allowed to have IN conditions, but that the workload only had point queries. We now tackle the problem of exploiting IN queries in the workload as well. This is reduced to the problem of precomputing atomic probabilities such as $p(z\,|W)$ and $p(x\,|y, W)$ from such a workload. These atomic probabilities are necessary for computing the ranking function derived in Equation (8).

Our approach is to "conceptually expand" the workload by *splitting* each IN query into sets of appropriately weighted point queries. For example, a query with IN conditions such as "City IN (Bellevue, Redmond, Carnation) AND Price IN (High, Moderate)" may be split into $3 \times 2 = 6$ point queries, each representing specific combinations of values from the IN conditions. In this example, each such point query is given a weight of 1/6; this weighting is necessary to make sure that queries with large IN conditions do not dominate the calculations of the atomic probabilities.

Atomic probabilities may now be computed as follows: $p(z|W)$ is the (weighted) fraction of the queries in the expanded workload that refer to z, while $p(x|y, W)$ is the (weighted) fraction of all queries that refer to x from all queries that refer to y in the expanded workload. Of course, the workload is not literally expanded; these probabilities can be easily computed from the original workload that contain the IN queries.

4.4.2 *Numeric Attributes.* Thus far in the article we have only been considering categorical data. We now extend our results to the case when the data also has numeric attributes. For example, in the homes database, we may have numeric attributes such as square footage, age, etc. Queries may now have range conditions, such as "Age BETWEEN (5, 10) AND Sqft BETWEEN (2500, 3000)."

One obvious way of handling numeric data and queries is to simply treat them as categorical data—to consider every distinct numerical value in the database as a categorical value. Queries with range conditions can be then converted to queries with corresponding IN conditions, and we can then apply the methods outlined in Section 4.4.1. However, the main problem arising with such an approach is that the sheer size of the numeric domain ensures that many, in fact most, distinct values are not adequately represented in the workload. For example, perhaps numerous workload queries have requested for homes between 3000 and 4000 sqft. However, there may be one or two 2995-sqft homes in the database, but unfortunately these homes would be considered far less popular by the ranking algorithm.

A simple strategy for overcoming this problem is to discretize the numerical domain into *buckets*, which can then be treated as categorical data. However, most simple bucketing techniques are errorprone because inappropriate choices of bucket boundaries may separate two values that are otherwise close to each other. In fact, complex bucketing techniques for numeric data have been extensively studied in other domains, such as in the construction of histograms for approximating data distributions (see Poosala et al. [1996; Jagadish et al. 1998]) and in earlier database ranking algorithms (see Agrawal et al. [2003]), as well as in discretization methods in classification studies (see Martinez et al. [2004]). In this article too, we investigate the bucketing problem that arises in our context in a systematic manner, and present principled solutions that are adaptations of well-known methods for histogram construction.

Let us consider where exactly the problem of numeric attributes arises in our case. Given a query Q, the problem arises when we attempt to compute the score of a tuple t based on the ranking formula in Equation (8). We need accurate estimations of the atomic probabilities $p(z \mid W)$, $p(z \mid D)$, $p(x \mid y, W)$, and $p(x \mid y, D)$ when some of these values are numeric. What is really needed is a way of "smoothening" the computations of these atomic probabilities, so that, for example, if $p(z \mid W)$ is high for a numeric value (i.e., z has been referenced many times in the workload), $p(z+\varepsilon \mid W)$ should also be high for nearby values $z+\varepsilon$. Similar smoothening techniques should be applied to the other types of atomic probabilities, $p(z \mid D)$, $p(x \mid y, W)$ and $p(x \mid y, D)$. Furthermore, these probabilities have to be precomputed earlier, and should only be "looked up" at query time. In the following we discuss our solutions in more detail.

4.4.2.1 *Estimating $p(z \mid D)$ and $p(x \mid y, D)$.* We first discuss how to estimate $p(z \mid D)$. Let z be a value of some numeric attribute, say A. As mentioned earlier, the naïve but inaccurate way of estimating $p(z \mid D)$ would be to simply treat A as a categorical attribute—thus $p(z \mid D)$ would be the relative frequency of the occurrence of z in the database. Instead, our approach is to assume that $p(z \mid D)$ is the density, at point z, of a *continuous* probability density function (pdf) $p(z \mid D)$ over the domain of A. We therefore use standard density estimation techniques—in our case, histograms—to approximate this pdf using the values of A occurring in the database. There are a wide variety of histogram techniques for density estimation, such as equiwidth histograms, equidepth histograms, and even "optimal" histograms where bucket boundaries are set such that the squared error between the actual data distribution and the distribution represented by the histogram is minimized (see Poosala et al. [1996]; Jagadish et al. [1998] for relevant results on histogram construction). In our case, we use the popular and efficient technique of *equidepth histograms*, where the range is divided into a set of nonoverlapping buckets such that each bucket contains the same number of values.[2] Once this histogram has been precomputed, the density $p(z \mid D)$ at any point z is looked up at runtime by determining the bucket to which z belongs.

We next discuss how to estimate $p(x \mid y, D)$. Intuitively, our approach is to compute a two-dimensional histogram that represents the distribution of all (x, y) pairs that occur in the database. At runtime, we look up this histogram to determine the density, at point x, of the marginal distribution $p(x \mid y, D)$.

Consider first the case where the attribute A of x is numeric, but the attribute B of y is categorical. Our approach for this problem is to compute, for each distinct value y of B, the histogram over all values of A that cooccur with y in the database. Each such histogram represents the marginal probability density function $p(x \mid y, D)$. One issue that arises is if there are numerous distinct values for B, which may result in too many histograms. We circumvent this problem by only building histograms for those y values for which the corresponding number of A values occurring in the database is larger than a given threshold.

We next consider the case where A is categorical whereas B is numeric. We first compute the histogram of the distribution $p(y \mid D)$ as explained above. We then compute pairwise association rules of the form $b \to x$ where b is any bucket of $p(y \mid D)$ and x is any value of A. Then the density $p(x \mid y, D)$ is approximated as the confidence of the association rule $b \to x$ where b is the bucket to which y belongs.

Finally, consider the case where A and B are both numeric. As above, we first compute the histogram for $p(y \mid D)$. Then, for each bucket b of the histogram corresponding to $p(y \mid D)$, we compute the histogram over all values of A that cooccur with b in the database. Each such histogram represents the marginal

[2]In our approach, we set the number of buckets to 50.

probability density function $p(x \mid y, D)$. As before, if there are numerous buckets of $p(y \mid D)$, this may result in too many histograms, so we only build histograms for those buckets for which the corresponding number of A values occurring in the database is larger than a given threshold.

4.4.2.2 *Estimating $p(z \mid W)$ and $p(x \mid y, W)$.* The estimation of these quantities is similar to the corresponding methods outlined above, except that the various histograms have to be built using the workload rather than the database. The further complication is that, unlike the database where histograms are built over sets of point data, the workload contains range queries, and thus the histograms have to be built over *sets of ranges*. We outline the extensions necessary for the estimation of $p(z \mid W)$; the extensions for estimating $p(x \mid y, W)$ are straightforward and omitted.

Let z be a value of a numeric attribute A. As before, our approach is to assume that $p(z \mid W)$ is the density, at point z, of a *continuous* probability density function $p(z \mid W)$ over the domain of A. However, we cannot directly use standard density estimation techniques such as histograms because, unlike the database, the workload specifies a set of ranges over the domain of A, rather than a set of points over the domain of A.

We extend the concept of equidepth histograms to sets of ranges as follows. Let query Q_i in the workload specify the range (zL_i, zR_i). If this is the only query in the workload, we can view this as a probability density function over the domain of A, where the density is $1/(zR_i - zL_i)$ for all points $zL_i \leq z \leq zR_i$, and 0 for all other points. The pdf for the entire workload is computed by averaging these individual distributions at all points over all queries—thus the pdf for the workload will resemble a histogram with a potentially large number of buckets (proportional to the number of queries in the workload).

We now have to approximate this "raw" histogram using an equidepth histogram with far fewer buckets. The bucket boundaries of the equidepth histogram should be selected such that the probability mass within each bucket is the same. Construction of this equidepth histogram is straightforward and is omitted. At runtime, given a value z, the density can be easily looked up by determining the bucket to which z belongs.

4.4.3 *Multitable Databases.* Another aspect to consider is when the database spans across more than one table. Important multitable scenarios are star/snowflake schemas where fact tables are logically connected to dimension tables via foreign key joins. For example, while the actual homes for sale may be recorded in a fact table, various properties of each home, such as demographics of neighborhood, builder characteristics, etc., may be found in corresponding dimension tables. In this case, we create a logical view representing the join of all these tables—thus this view contains all the attributes of interest—and apply our ranking methodology on this view. As shall be evident later, if we follow the precomputation method of Section 5.2, then there is no need to materialize the logical view, since the execution is then based on the precomputed lists and the logical view would only be accessed at the final stage to output the top results.

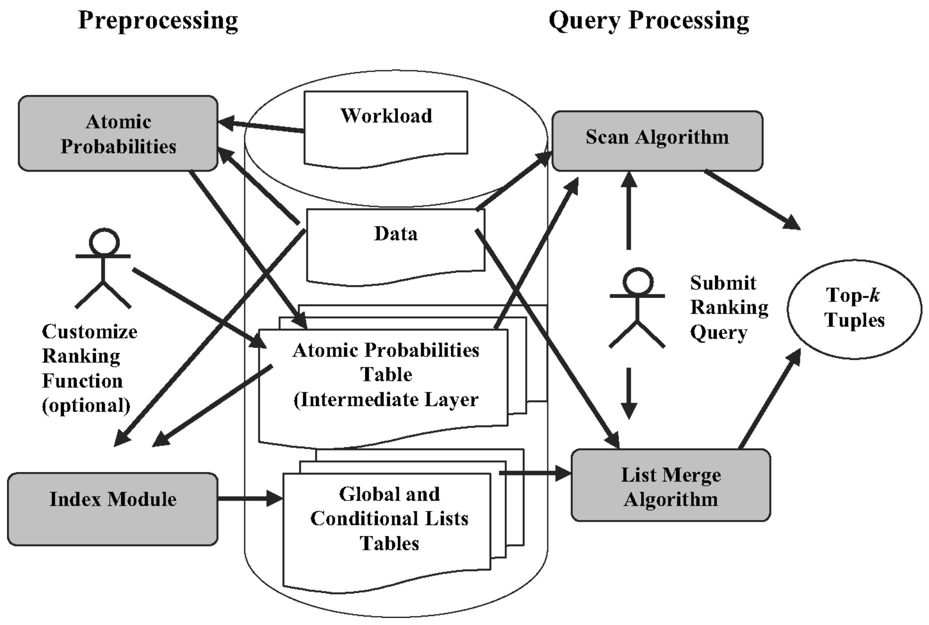


Fig. 1. Architecture of ranking system.

5. IMPLEMENTATION

In this section we discuss the architecture and the implementation of our database ranking system.

5.1 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.

5.2 Preprocessing

This component is divided into several modules. First, the *Atomic Probabilities Module* computes the quantities $p(y\,|\,W)$, $p(y\,|\,D)$, $p(x\,|\,y, W)$, and$p(x\,|\,y, D)$ for all distinct values x and y. These quantities are computed by scanning the workload and data, respectively. While the latter two quantities for categorical data can be computed by running a general association rule mining algorithm such as that given in Agrawal et al. [1995] on the workload and data, we instead chose to directly compute all pairwise cooccurrence frequencies by a single scan of the workload and data, respectively. The observed probabilities are then smoothened using the Bayesian *m-estimate* method [Cestnik 1990]. (We note that more sophisticated Bayesian methods that use an informative prior may be employed instead.) For numeric attributes, we compute

$p(y|W)$, $p(y|D)$, $p(x|y, W)$, and$p(x|y, D)$ as histograms, as described in Section 4.4.2.

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). For numeric quantities, attribute values are essentially the ranges of the corresponding buckets. 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 precompute 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 preprocessing and query processing, that is, what additional yet reasonable precomputations are possible that can enable faster query-processing algorithms than Scan? (We note that tradeoffs between preprocessing and query processing techniques are common in IR systems [Grossman and Frieder 2004].)

The high-level intuition behind our approach to the above problem is as follows. Instead of precomputing the top-k tuples for all possible queries, we precompute 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 $\{x\}$. For example, "SELECT $*$ FROM HOMES WHERE CITY=Kirkland" is an atomic query. 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.

This high-level idea is conceptually related to the merging of inverted lists in IR. However, our 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 [Bruno et al. 2002b; Fagin 1998; Fagin et al. 2001; Güntzer et al. 2000; Nepal and Ramakrishna 1999] 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 (4) (which involves *all* attribute values of the tuple, and not

```
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
```

Fig. 2. The Index Module.

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 precompute 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., of 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). 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 precomputed by the *Index Module* of the preprocessing component. This module (see Figure 2 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 $s+1$ multiplications and results in a score[3] 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, that is, the tuple-ids in the lists 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, that is, 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 tables 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 5.3.

5.2.1 *Presence of Functional Dependencies.* If we consider functional dependencies, then the content of the conditional and global lists is changed as follows.

$$CondScore = \begin{cases} \prod_{z \in t} p(x|z,W) \prod_{z \in t'} \frac{1}{p(x|z,D)}, & x \in A', \\ \prod_{z \in t} p(x|z,W), & otherwise, \end{cases}$$

and

$$GlobScore = \prod_{z \in t} p(z|W) \prod_{z \in t'} \frac{1}{p(z|D)},$$

where$A' = \{A_i \in A | \neg\exists A_j \in A, FD : A_j \rightarrow A_i\}$and t' is the subset of the attribute values of t that belong to A'.

[3]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.

```
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 $B$
```

Fig. 3. The List Merge Algorithm.

5.3 Query Processing

In this subsection we describe the query processing component. The naïve Scan algorithm has already been described in Section 5.2, so our focus here is on the alternate List Merge algorithm (see Figure 3). 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 5.2, is proportional to the actual score derived in Equation 4). 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.

5.3.1 *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 preprocessing time, at the expense of an increase in the query processing time.

Determining which lists to retain/omit at preprocessing 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 Bruno et al. [2002b], where not all sources have sorted stream access.

5.4 Evaluating IN and Range Queries

As mentioned in Section 4.4.1, executing IN queries is more involved because each result tuple has possibly different specified values. This makes the application of the List Merge algorithm more challenging, since the Scan algorithm computes the score of each result tuple from the information in this intermediate layer. In particular, List Merge is complicated in two ways:

(a) We cannot use a single conditional list for a specified attribute with an IN condition, since a single conditional list only contains tuples containing a single attribute values. For example, for the query "City IN (Redmond, Bellevue)" we must merge the conditional lists C_{Redmond} and C_{Bellevue}.

(b) More seriously, we can no longer use a single conditional C_x list for a specified attribute X_i (with or without an IN condition), if there is another specified attribute X_j with an IN condition. The reason is that the product $\prod_{z \in t} \frac{p(x|z,W)}{p(x|z,D)}$ stored in C$_x$ (x is an attribute value for attribute X_i) spans across all attribute values of t and not only across the unspecified attribute values Y as required by Equation (8). This was not a problem for the case of point queries (Equations (4) and (5)) because the factors $\frac{p(x|z,W)}{p(x|z,D)}$, where $z \in X$ of the above product, are common for all result-tuples, and hence the scores are multiplied by a common constant. On the other hand, if there is an attribute X_j with IN condition, then the factor $\frac{p(x|z,W)}{p(x|z,D)}$, where z is an attribute value for X_j, is not common and hence cannot be ignored.

To overcome these challenges, we split each IN query to a set of point queries, which are evaluated as usual and then their results are merged. In particular, suppose we have the IN query Q: "X_1 IN ($x_{1,1} \ldots x_{1,r1}$) and … and X_s IN ($x_{s,1} \cdots x_{s,rs}$)." First we split Q into $r1 \cdot r2 \cdot \ldots \cdot rs$ point queries, one for each combination of selecting a single value from each specified attribute. Then these point queries are evaluated separately and their results (along with their scores) are merged. To see that such a splitting approach yields the correct results, note that the first (global) part of the ranking function in Equation (8) is the same for both the point and the IN query and is equal to the scores in the Global Lists. The conditional part of Equation (8) only depends on the values of the tuple *t* and the set of specified attributes but not on the particular conditions of the query. Hence, the point queries will assign the same scores as the IN query. Finally, it should be clear that the same set of tuples is returned as results in both cases.

The splitting method is efficient only if a relatively small number of point queries results from the split, that is, if $r1 \cdot r2 \cdot \ldots \cdot rs$ is small. The key advantage of this approach is that no additional conditional lists need to be created to support IN queries. An alternate approach described next is preferable when the IN conditions frequently involve the same small set of attributes. We illustrate this idea through an example. Suppose queries specifying IN condition only on the City attribute are popular. Then, we create a new conditional list $C_x^{\neg City}$ for every attribute value x not in the *City* attribute, using the formula $CondScore = \prod_{z \in \{t - t.City\}} \frac{p(x|z,W)}{p(x|z,D)}$, and use these conditional lists whenever a query with an IN condition only on *City* is submitted.

Finally, note that *range queries*—that is, queries with ranges on numeric attributes—may be evaluated using techniques similar to queries with IN conditions. For example, if a condition such as "A BETWEEN (x_1, x_2)" is specified, then this condition is discretized into an IN condition by replacing the range with buckets from the precomputed histogram $p(x|W)$ that overlap with the range. In case the range only partially overlaps with the leading/trailing buckets, the retrieved tuples that do not satisfy the query condition are discarded in a final filtering phase.

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 used 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 U.S., with a mix of categorical as well as numeric attributes such as Price, Year, City, Bedrooms, Bathrooms, Sqft, Garage, etc. 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

Table I. Sizes of Datasets

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

as Title, Year, Genre, Director, FirstActor, SecondActor, Certificate, Sound, Color, etc. 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 Table I. The quality experiments were conducted on the Seattle Homes and Movies tables, while the performance experiments were conducted on the Seattle Homes and the U.S. 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.

6.1 Quality Experiments

We evaluated the quality of three different ranking methods: (a) our ranking method, henceforth referred to as *Conditional*; (b) the ranking method described in Agrawal et al. [2003], henceforth known as *Global*; and (c) a baseline *Random* algorithm, which simply ranks and returns the top-k tuples in arbitrary order. 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, for example, 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 queries against the database—for example, 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 two to four 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, for example, teenage males wishing to see action thrillers, people interested in comedies from the 1980s, 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.

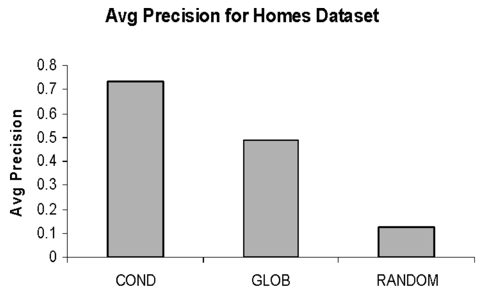
Likewise, for the Movies dataset, Conditional often produced rankings that were superior to Global. For example, for a query such as "Year=1980s and Genre=Thriller," Conditional ranked movies such as *Indiana Jones and the Temple of Doom* higher than *Commando*, because the workload indicated that Harrison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

As for Random, it produced quite irrelevant results in most cases.

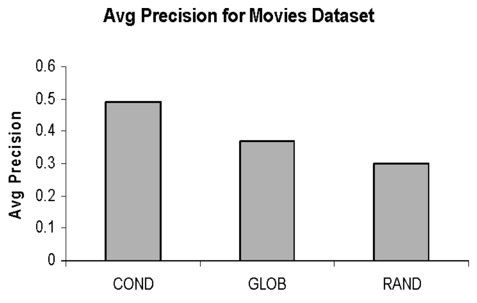
6.1.2 *Ranking Evaluation.* We now present a more formal evaluation of the ranking quality produced by the ranking algorithms. We conducted two surveys; the first compared the rankings against user rankings using standard precision/recall metrics, while the second was a simpler survey that asked users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

6.1.2.1 *First Survey.* Since requiring users to rank the entire database for each query for the first survey would have been extremely tedious, we used the following strategy. For each dataset, 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.

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



(a) Homes dataset



(b) Movies dataset

Fig. 4. Average p2recision.

number of retrieved tuples, while Recall is the ratio of the fraction of the number of retrieved tuples that are relevant to the total number of relevant tuples (see Baeza-Yates and Ribeiro-Neto [1999]). In our case, the total number of relevant tuples was 10, so Precision and Recall were equal. (We reiterate that this is only an artefact of our experimental setu—the "true" Recall can be measured only if the user is able to mark the entire dataset, which was unfeasible in our case).

We experimented with several sets of queries in this survey. We first present the results for the following four IN/Range queries for the Seattle Homes dataset:

Q1: Bedrooms=4 AND City IN{Redmond, Kirkland, Bellevue};
Q2: City IN {Redmond, Kirkland, Bellevue} AND Price BETWEEN ($700K, $1000K);
Q3: Price BETWEEN ($700K, $1000K);
Q4: School=1 AND Price BETWEEN ($100K, $200K).

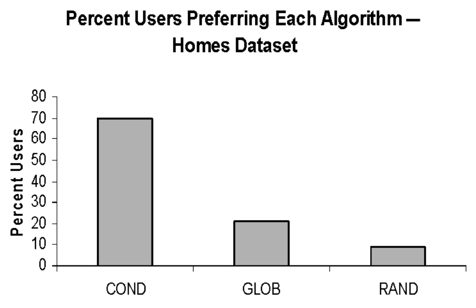
The precision (averaged over these queries) of the different ranking methods is shown in Figure 4 (a). As can be seen, the quality of Conditional ranking was superior to Global, while Random was significantly worse than either.

We next present our survey results for the following five point queries for the Movies dataset (where precision was measured as described above for the Seattle Homes dataset):

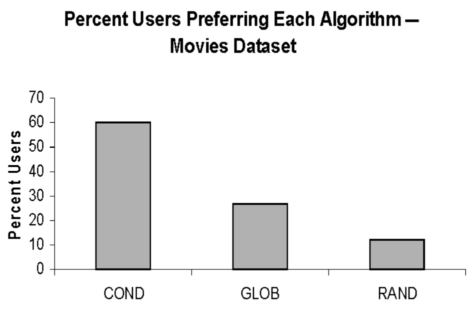
Q1: Genre=thriller AND Certificate=PG-13;
Q2: YearMade=1980 AND Certificate=PG-13;
Q3: Certificate=G AND Sound=Mono;
Q4: Actor1=Dreyfuss, Richard;
Q5: Genre=Sci-Fi.

The results are shown in Figure 4 (b). The quality of Conditional ranking was superior to Global, while Random was worse than either.

6.1.2.2 *Second Survey.* In addition to the above precision/recall experiments, we also conducted a simpler survey in which users were



(a) Homes dataset



(b) Movies dataset

Fig. 5. Percent of users preferring each algorithm.

given the top-5 results of the three ranking methods for five queries (different from the previous survey), and were asked to choose which rankings they preferred.

We used the following IN/Range queries for the Seattle Homes dataset:

Q1: Bedrooms=4 AND City IN (Redmond, Kirkland, Bellevue);
Q2: City IN (Bellevue, Kirkland) AND Price BETWEEN ($700K, $1000K);
Q3: Price BETWEEN ($500K, $700K) AND Bedrooms=4 AND Year > 1990;
Q4: City=Seattle AND Year > 1990;
Q5: City=Seattle AND Bedrooms=2 AND Price=500K.

We also used the following point queries for the Movies dataset:

Q1: YearMade=1980 AND Genre=Thriller;
Q2: Actor1=De Niro, Robert;
Q3: YearMade=1990 AND Genre=Thriller;
Q4: YearMade=1995 AND Genre=Comedy;
Q5: YearMade=1970 AND Genre=Western.

Figure 5 shows the percent of users that prefer the results of each 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.

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 U.S. Homes datasets for these experiments. We report performance results of our algorithms on point queries—we do not report results for IN/range queries, as each such query is split into a collection of point

Table II. Time and Space Consumed by Index Module

Datasets	List Building Time	List Size
Seattle Homes	1500 ms	7.8 MB
U.S. Homes	80,000 ms	457.6 MB

queries whose results are then merged in a straightforward manner as described in Section 5.4.

6.2.1 *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. Table II 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.

If space is a critical issue, we can adopt the space-saving variation of the List Merge algorithm as discussed in Section 5.3. We report on this next.

6.2.2 *Space-Saving Variations.* In this experiment, we showed how the performance of the algorithms changes when only a subset of the set of global and conditional lists are stored. Recall from Section 5.3 that we only retain lists for the values of the frequently occurring attributes in the workload. For this experiment, we considered 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 measured their execution times. The compared algorithms were

—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 6 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 1 s for each run, and consequently its graph is almost along the x-axis). This is to be expected as most of the computations were accomplished at preprocessing time. The performance of Scan degraded when the total number of selected tuples increased, because the scores of more tuples need to be calculated at runtime. In contrast, the performance of LM and LMM actually improved slightly. This interesting phenomenon occurred 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.

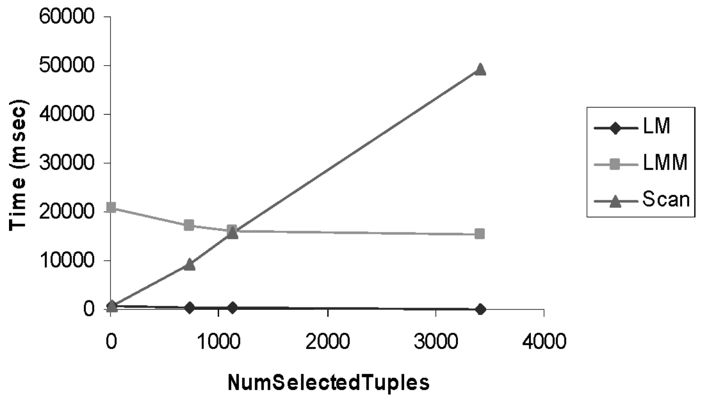


Fig. 6. Execution times of different variations of list merge and scan for seattle homes dataset.

Table III. Execution Times of List Merge for U.S. Homes Dataset

NumSelected Tuples	LM Time (ms)	Scan Time (ms)
350	800	6515
2000	700	39,234
5000	600	11,5282
30000	550	56,6516
80000	500	3,806,531

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 U.S. Homes dataset with queries satisfying many more tuples (see Table III).

6.2.3 *Varying Number of Specified Attributes.* Figure 7 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 U.S. 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 increased with the number of specified attributes. This experiment demonstrates the criticality of keeping the number of sorted streams small in our adaptation of TA.

6.2.4 *Varying K in Top-k.* This experiment showed how the performance of the algorithms decreases with the number K of requested results. The graphs are shown in Figures 8(a) and 8(b) for the Seattle and the U.S. databases respectively. For both datasets, we selected queries with two attributes, which returned about 500 results. Notice that the performance of Scan was not affected by K, since it is not a top-k algorithm. In contrast, LM degraded with K because a longer prefix of the lists needed to be processed. Also notice that Scan

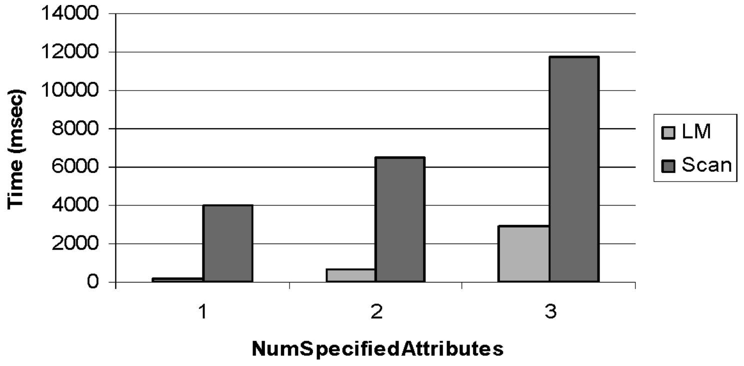


Fig. 7. Varying number of specified atributes for U.S. Homes dataset.

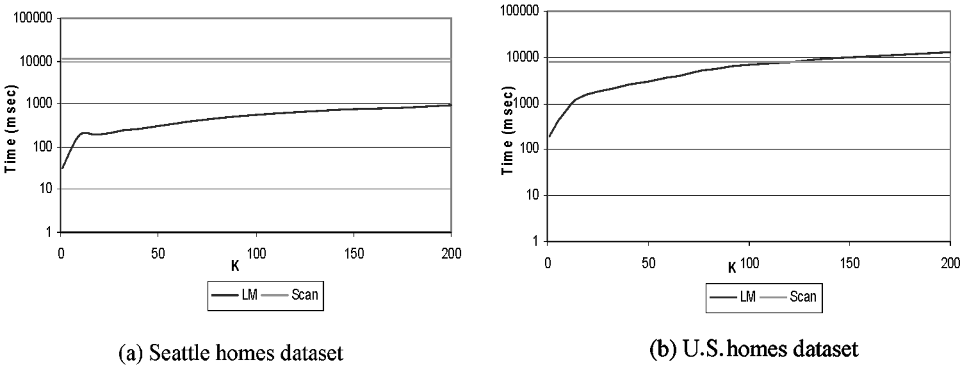


Fig. 8. Varying number K of requested results.

took about the same time for both datasets because the number of the results returned by the selection was the same (500).

7. CONCLUSIONS AND FUTURE WORK

We propose 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 nontext data can be leveraged in a meaningful way for ranking. Second, rather than just query strings present in the workload, can more comprehensive user interactions be leveraged in ranking algorithms—for example, tracking the actual tuples that the users select in response to query results? 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.

ACKNOWLEDGMENTS

We thank the anonymous referees for their extremely useful comments on an earlier draft of this article.

REFERENCES

Agrawal, S., Chaudhuri, S., and Das, G. 2002. DBXplorer: A system for keyword based search over relational databases. In *proceedings of ICDE*.

Agrawal, S., Chaudhuri, S., Das, G., and Gionis, A. 2003. Automated ranking of database query results. In *proceedings of CIDR*.

Amer-Yahia, S., Case, P., Roelleke, T., Shanmugasundaram, J., and Weikum. G. 2005a. Report on the DB/IR panel at SIGMOD 2005. *ACM SIGMOD Rec. 34*, 4, 71–74.

Amer-Yahia, S., Koudas, N., Marian, A., Srivastava, D., and Toman, D. 2005b. Structure and content scoring for XML. In *proceedings of VLDB*.

Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., and Verkamo, A. I. 1995. Fast discovery of association rules. In *proceedings of KDD*.

Barbara, D., Garcia-Molina, H., and Porter, D. 1992. The management of probabilistic data. *IEEE Trans. Knoual. Data Eng. 4*, 5, 487–502.

Bruno, N., Gravano, L., and Chaudhuri, S. 2002a. Top-*k* selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst*.

Bruno, N., Gravano, L., and Marian, A. 2002b. Evaluating top-*k* queries over Web-accessible databases. In *proceedings of ICDE*.

Breese, J., Heckerman, D., and Kadie, C. 1998. Empirical analysis of predictive algorithms for collaborative filtering. In *proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*.

Bhalotia, G., Nakhe, C., Hulgeri, A., Chakrabarti, S., and Sudarshan, S. 2002. Keyword searching and browsing in databases using BANKS. In *Proceedings of ICDE*.

Baeza-Yates, R. and Ribeiro-Neto, B. 1999. *Modern Information Retrieval*, 1st ed. Addison-Wesley, Reading, MA.

Cestnik, B. 1990. Estimating probabilities: A crucial task in machine learning. In *Proceedings of the European Conference on artificial Intelligence*.

Cavallo, R. and Pittarelli, M. 1987. The theory of probabilistic databases. In *Proceedings of VLDB*.

Chaudhuri, S., Das, G., Hristidis, V., and Weikum, G. 2004. Probabilistic ranking of database query results. In *Proceedings of VLDB*.

Chinenyanga, T. T. and Kushmerick, N. 2002. An expressive and efficient language for XML information retrieval. *J. Amer. Soc. Inform. Sci. Tech. 53*, 6, 438–453.

Croft, W. B. and Lafferty, J. 2003. *Language Modeling for Information Retrieval*. Kluwer, Norwell, MA.

Carmel, D, Maarek, Y. S. , Mandelbrod, M., Mass, Y., and Soffer, A. 2003. Searching XML documents via XML fragments. In *Proceedings of SIGIR*.

Cohen, W. 1998a. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of SIGMOD*.

Cohen, W. 1998b. Providing database-like access to the Web using queries based on textual similarity. In *Proceedings of SIGMOD*.

Chakrabarti, K., Porkaew, K., and Mehrotra, S. 2000. Efficient query references in multimedia databases. In *Proceedings of ICDE*.

Dalvi, N. N. and Suciu, D. 2005. Answering queries from statistics and probabilistic Views. In *Proceedings of VLDB*.

Fagin, R. 1998. Fuzzy queries in multimedia database systems. In *Proceedings of PODS*.

Fagin, R., Lotem, A., and Naor, M. 2001. Optimal aggregation algorithms for middleware. In *Proceedings of PODS*.

Fuhr, N. 1990. A probabilistic framework for vague queries and imprecise information in databases. In *Proceedings of VLDB*.

Fuhr, N. 1993. A probabilistic relational model for the integration of IR and databases. In *Proceedings of ACM SIGIR Conference on Research and Development in Information Retrieval*.

FUHR, N. AND GROSSJOHANN, K. 2004. XIRQL: An XML query language based on information retrieval concepts. *ACM Trans. Inform. Syst. 22*, 2, 313–356.

FUHR, N. AND ROELLEKE, T. 1997. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inform. Syst. 15*, 1, 32–66.

FUHR, N. AND ROELLEKE, T. 1998. HySpirit—a probabilistic inference engine for hypermedia retrieval in large databases. In *Proceedings of EDBT*.

GROSSMAN, D. AND FRIEDER, O. 2004. *Information Retrieval—Algorithms and Heuristics*. Springer, Berlin, Germany.

GÜNTZER, U., BALKE, W.-T., AND KIESSLING, W. 2000. Optimizing multi-feature queries for image databases. In *Proceedings of VLDB*.

GUO, L., SHAO, F., BOTEV, C., AND SHANMUGASUNDARAM. J. 2003. XRANK: Ranked keyword search over XML documents. In *Proceedings of SIGMOD*.

HARPER, D. AND VAN RIJSBERGEN, C. J. 1978. An evaluation of feedback in document retrieval using co-occurrence data. *J. Document. 34*, 3, 189–216.

HRISTIDIS, V. AND PAPAKONSTANTINOU, Y. 2002. DISCOVER: Keyword search in relational databases. In *Proceedings of VLDB*.

HRISTIDIS, V., GRAVANO, L., AND PAPAKONSTANTINOU, Y. 2003a. Efficient IR-style keyword search over relational databases. In *Proceedings of VLDB*.

HRISTIDIS, V., PAPAKONSTANTINOU, Y., AND BALMIN, A. 2003b. Keyword proximity search on XML graphs. In *Proceedings of ICDE*.

JAGADISH, H. V., POOSALA, V., KOUDAS, N., SEVCIK, K., MUTHUKRISHNAN, S., AND SUEL, T. 1998. Optimal histograms with quality guarantees. In *Proceedings of VLDB*.

KIESSLING, W. 2002. Foundations of preferences in database systems. In *Proceedings of VLDB*.

LAKSHMANAN, L. V. S., LEONE, N., ROSS, R., AND SUBRAHMANIAN, V. S. 1997. ProbView: A flexible probabilistic database system. *ACM Trans. Database Syst. 22*, 3, 419–469.

LALMAS, M. AND ROELLEKE, T. 2004. Modeling vague content and structure querying in XML retrieval with a probabilistic object-relational framework. In *Proceedings of FQAS*.

MARTINEZ, W., MARTINEZ, A., AND WEGMAN, E. 2004. Document classification and clustering using weighted text proximity matrices. In *Proceedings of Interface*.

MOTRO, A. 1988. VAGUE: A user interface to relational databases that permits vague queries. *ACM Trans. Informat. Syst. 6*, 3 (July), 187–214.

NAZERI, Z., BLOEDORN, E., AND OSTWALD, P. 2001. Experiences in mining aviation safety data. In *Proceedings of SIGMOD*.

NEPAL, S. AND RAMAKRISHNA, M. V. 1999. Query processing issues in image (multimedia) databases. In *Proceedings of ICDE*.

ORTEGA-BINDERBERGER, M., CHAKRABARTI, K., AND MEHROTRA, S. 2002. An approach to integrating query refinement in SQL. In *Proceedings of EDBT*. 15–33.

POOSALA, V., IOANNIDIS, Y. E., HAAS, P. J., AND SHEKITA, E. J. 1996. Improved histograms for selectivity estimation of range predicates. In *Proceedings of SIGMOD*. 294–305.

RADLINSKI, F. AND JOACHIMS, T. 2005. Query chains: Learning to rank from implicit feedback. In *Proceedings of KDD*.

RUI, Y., HUANG, T. S., AND MEHROTRA, S. 1997. Content-based image retrieval with relevance feedback in MARS. In *Proceedings of the IEEE Conference on Image Processing*.

SHEN, X., TAN, B. AND ZHAI, C. 2005. Context-sensitive information retrieval using implicit feedback. In *Proceedings of SIGIR*.

SPARCK JONES, K., WALKER, S., AND ROBERTSON, S. E. 2000a. A probabilistic model of information retrieval: Development and comparative experiments—Part 1. *Inf. Process. Man. 36*, 6, 779–808.

SPARCK JONES, K., WALKER, S., AND ROBERTSON, S. E. 2000a. A probabilistic model of information retrieval: Development and comparative experiments—Part 2. *Inf. Process. Man. 36*, 6, 809–840.

THEOBALD, A. AND WEIKUM, G. 2002. The index-based XXL search engine for querying XML data with relevance ranking. In *Proceedings of EDBT*.

THEOBALD, M., SCHENKEL, R., AND WEIKUM, G. 2005. An efficient and versatile query engine for topX search. In *Proceedings of VLDB*.

WU, L., FALOUTSOS, C., SYCARA, K., AND PAYNE, T. 2000. FALCON: Feedback adaptive loop for content-based retrieval. In *Proceedings of VLDB*.

WHITTAKER, J. 1990. *Graphical Models in Applied Multivariate Statistics*. Wiley, New York, NY.
WIDOM, J. 2005. Trio: A system for integrated management of data, accuracy, and lineage. *CIDR*.
WIMMERS, L., HAAS, L. M. , ROTH, M . T., AND BRAENDLI, C. 1999. Using Fagin's algorithm for merging ranked results in multimedia middleware. In *Proceedings of CoopIS*.
XU, J. AND CROFT, W. B. 1996. Query expansion using local and global document analysis, In *Proceedings of* SIGIR. 4–11.
YU, C.T. AND MENG, W. 1998. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann, San Francisco, CA.

Received November 2005; revised June 2006; accepted June 2006