Automated Ranking of Database Query Results

Sanjay Agrawal
Microsoft Research

Surajit Chaudhuri
Microsoft Research

Gautam Das
Microsoft Research

Aristides Gionis
Computer Science Dept
Stanford University

Abstract

Ranking and returning the most relevant results of a query is a popular paradigm in Information Retrieval. We discuss challenges and investigate several approaches to enable ranking in databases, including adaptations of known techniques from information retrieval. We present results of preliminary experiments.

1. Introduction

Automated ranking of the results of a query is a popular aspect of the query model in Information Retrieval (IR) that we have grown to depend on. In contrast, database systems support only a Boolean query model. For example, a selection query on a SQL database returns all tuples that satisfy the conditions in the query. Therefore, the following two scenarios are not gracefully handled by a SQL system:

1. *Empty answers*: When the query is too selective, the answer may be empty. In that case, it is desirable to have the option of requesting a ranked list of approximately matching tuples without having to specify the ranking function that captures "proximity" to the query. An FBI agent or an analyst involved in data exploration will find such functionality appealing.
2. *Many answers*: When the query is not too selective, too many tuples may be in the answer. In such a case, it will be desirable to have the option of ordering the matches automatically that ranks more "globally important" answer tuples higher and returning only the best matches. A customer browsing a product catalog will find such functionality attractive.

Conceptually, the automated ranking of query results problem is really that of taking a user query (say, a conjunctive selection query) and mapping it to a Top-*K* query with a ranking function that depends on given conditions in the user query. The key questions are:

- How to derive such ranking functions automatically? How well do ranking functions from IR apply?
- Are the ranking techniques for handling empty answers and many answers problems different?
- How to execute such Top-*K* queries efficiently over large databases?

We will start off by asking ourselves how to make it possible for relational databases to adapt ranking functions from IR for handling the database ranking problem. When each attribute in the relation is a categorical attribute, we can "mimic" the IR solution by applying the TF-IDF idea that is based on the frequency of occurrence of attribute values in the database. However, unlike text documents, databases contain numeric as well as categorical information. Therefore, we need to extend TF-IDF concepts to numerical domains. We develop *IDF Similarity*, a database ranking function that extends TF-IDF concepts to databases containing a heterogeneous mix of categorical as well as numeric data.

While IDF Similarity works well for some database ranking applications, sometimes its effectiveness is quite limited. In certain instances the relevance of data values for ranking may be due to other factors in addition to their frequencies. This has been noted in the IR domain as well, where sometimes one has to go beyond TF-IDF weightings to derive accurate ranking functions. This begs the question: *what else could be the basis of generic ranking in databases?* We show that collecting the *workload* on the database can be quite useful for ranking. In a way, this may be viewed as a poor man's choice of relevance feedback and collaborative filtering where a user's final choice of relevant tuples is not recorded. Despite its primitive nature, such workload information can help determine the frequency with which database attributes and values are referenced. When used in conjunction with IDF, workload information boosts ranking quality. We develop *QF Similarity,* a ranking function that leverages such workload information.

Much of the discussion in this paper focuses on the empty answers problem. Solving the many answers problem poses additional challenges because a ranking function that only depends on the conditions in the user

Proceedings of the 2003 CIDR Conference

query is inadequate for this problem. We extend our ranking functions with additional query independent components that measure the "importance" of tuples in a global sense.

Finally, even if we get the ranking functions right, for large databases, we have to minimize their impact on query processing. Although inverted lists are popular data structures for efficient retrieval in IR, they are inadequate for our purposes as we seek imprecise matches involving categorical and numerical attributes. We study adaptations of some recent algorithms for Top-*K* query processing, which leads us to yet another contribution of this paper; an index-based Top-*K* query processing algorithm, *ITA* that exploits our ranking functions.

We have built a system in which our ranking algorithms have been implemented on a relational DBMS. The system has two major components, a pre-processing component and a query processing component. The preprocessing component is a ranking function extractor that leverages data and workload characteristics. The query processing component is a Top-*K* algorithm that uses the ranking function and exploits the physical database design. We have performed user experiments on our system to evaluate its effectiveness. However, despite our best efforts, our user experiments are preliminary. Unlike IR which relies on extensive available user studies and benchmarks, no infrastructure is available today for evaluating database ranking.

The rest of this paper is organized as follows. In Section 2 we discuss related work. In Sections 3 and 4 respectively, we describe two database ranking functions for the empty answers problem, IDF Similarity and QF Similarity. Section 5 discusses differences between the empty answers and the many answers ranking problem, and describes extensions to our ranking functions to solve the latter problem. Section 6 discusses key implementation details, especially choices among Top-*K* processing techniques and our ITA algorithm. We present experiments in Section 7, and conclude in Section 8.

2. Related work

Extracting ranking functions has been extensively investigated in areas outside database research such as Information Retrieval. The Cosine Similarity metric with TF-IDF weighting of the vector space model [4] is very successful in practice. We extend the TF-IDF weighting technique for database ranking to handle a heterogeneous mix of numeric and categorical data.

Ranking is an important component in collaborative filtering research [5]. These methods require training data using queries as well as their ranked results. In contrast, we require workloads containing queries only.

In database research, there has been some scattered work on the automatic extraction of similarity/ranking functions from a database. The early work of [21] considered vague/imprecise similarity-based querying of databases. The problem of integrating databases and information retrieval systems has been attempted in several works [12, 13, 17, 18]. Information retrieval based approaches have been extended to XML retrieval in [26]. The papers [10, 23, 24, 32] employ relevance-feedback techniques for learning similarity in multimedia and relational databases. A keyword-based retrieval system over databases is proposed in [1].

The distinguishing aspects of our work from the above are (a) we address the challenges that a heterogeneous mix of numeric as well as categorical attributes pose, and (b) we propose a novel and easy to implement ranking method based on query workload analysis. Although [22] describes a ranking application for a mix of categorical and numeric data, the similarity function is not automatically derived but rather is based on domain knowledge of the application. The paper [30] proposes distance functions for heterogeneous data, but the emphasis is on classification applications. In [19, 20], the authors propose SQL extensions in which users can specify soft constraints in the form of preferences. These extensions broaden the expressiveness of search criteria by a user, but do not relieve the user from the onus of having to specify suitable ranking functions.

A major concern of this paper is the query processing techniques for supporting ranking. Several techniques have been previously developed in database research for the Top-*K* problem [6, 7, 14, 15, 31]. We adopt the algorithm in [15] for our purposes, and discuss issues such as how the relational engine and indexes/materialized views can be leveraged for query performance.

3. IDF Similarity: generalizing IR methods

In this section, we develop *IDF Similarity*, a database ranking function based on information retrieval techniques. We consider a database table *R* with categorical and numerical attributes $\{A_1, \ldots, A_m\}$ and tuples $\{T_1, \ldots, T_n\}$. The selection conditions will be *conjunctive* conditions, i.e., of the form "WHERE C_1 AND … AND C_m", where each atomic C_k is of the form "$A_k = value_k$". (More general conditions are discussed in Section 3.3. Also, our ranking techniques can be extended for multi-table databases; see Section 6.2.2).

3.1 IDF Similarity for categorical data

If the database only had categorical attributes, a very simple solution can be employed by essentially "mimicking" the well-known IR technique of Cosine Similarity with TF-IDF weighting by treating each tuple (and query) as a small document and defining a similarity function between tuples and queries. We note that such approaches have been considered in several prior works on database ranking (see Section 2). Henceforth in this paper *ranking function* and *similarity function* will be used interchangeably.

We start by briefly reviewing this standard IR technique. Given a set of documents and a query (the latter specified as a set of keywords), the problem is to retrieve the Top-K documents most relevant, or most *similar* to the query. Similarity between a document and the query is formalized as follows. Given a vocabulary of m words, a document is treated as an m-dimensional vector, where the ith component is the frequency of occurrence (also known as *term frequency*, or TF) of the ith vocabulary word in the document. Since a query is a set of words, it too has a vector representation. The *Cosine Similarity* between a query and a document is defined as the normalized dot-product of the two corresponding vectors. The Cosine Similarity may be further refined by scaling each component with the *inverse document frequency* (IDF) of the corresponding word (IDF(w) of a word w is defined as log(N/F(w)) where N is the number of documents, and F(w) is the number of document in which w appears). IDF has been used in IR to suggest that commonly occurring words convey less information about user's needs than rarely occurring words, and thus should be weighted less.

We can also adopt these techniques for our problem. More formally, for every value t in the domain of attribute A_k, we define $IDF_k(t)$ as $\log(n/F_k(t))$, where n is the number of tuples in the database and $F_k(t)$ is the frequency of tuples in the database where $A_k = t$. For any pair of values u and v in A_k's domain, let the quantity $S_k(u,v)$ be defined as $IDF_k(u)$ if $u = v$, and 0 otherwise. Consider tuple $T = <t_1,\ldots,t_m>$ and query $Q = <q_1,\ldots,q_m>$ (i.e. the latter has a C-condition of the form "WHERE $A_1 = q_1$ AND … AND $A_m = q_m$"). The similarity between T and Q is defined in Equation (1). We refer to the quantities $S_k(u,v)$ as similarity coefficients; thus the similarity between T and Q is simply the sum of corresponding similarity coefficients over all attributes. (To improve readability in the rest of the paper, we shall omit the subscript k where ever possible. Thus $S(t,q)$ will refer to the similarity coefficient $S_k(t, q)$, while A will refer to the attribute A_k).

$$SIM\,(T,Q) = \sum_{k=1}^{m} S_k(t_k, q_k) \qquad (1)$$

This similarity function closely resembles the IR-like Cosine Similarity with TF-IDF weightings, except that the dot-product is un-normalized. Also note that in our case, the term frequency TF is irrelevant since each tuple is treated as a small document in which a word, i.e. a <attribute, value> pair can only occur once. Henceforth we refer to this similarity function as *IDF Similarity*.

IDF Similarity can be very effective in certain database ranking applications. For example, if we query an automobile database for a "CONVERTIBLE" made by "NISSAN", the system first returns all Nissan convertibles, followed by other convertibles, and followed by other Nissan cars. This is because "CONVERTIBLE" is a rare car type and consequently has higher IDF than "NISSAN", a common car manufacturer.

3.2 Generalizing IDF Similarity for numeric data

The following interesting research challenges arise when we try to extend IDF Similarity for more general database schemas containing a heterogeneous mix of categorical and numerical attributes. Intuitively, the similarity coefficient $S(u, v)$ between values u and v of a numeric attribute A should be a smooth function inversely related to the "distance" between u and v. Thus, for numeric data it is inappropriate to adopt the definition of similarity coefficients in Section 3.1 because of their binary nature (where if u and v are arbitrarily close to each other yet distinct, $S(u, v)$ will incorrectly evaluate to 0). Moreover, the "frequency" (and hence "IDF") of a numeric value should depend on nearby values. For example, if we request for a home in a realtor database with price \$300k and 10 bedrooms, the price is less important for ranking purposes (there may be many houses priced close to \$300k, even if few have exactly that price) than the number of bedrooms (relatively fewer homes have around 10 bedrooms).

A simple solution is to discretize the domain of numeric attribute A into buckets, effectively treating a numerical attribute as categorical. However, most bucketing approaches are problematic since (a) inappropriate bucket boundaries may separate two values that are actually close to each other, (b) determining the correct number of buckets is not easy, and (c) values in different buckets are treated as completely dissimilar, regardless of the actual distance separating the buckets.

Instead, we propose a more robust definition of similarity for numeric data that does not suffer from these shortcomings. Let $\{t_1, t_2, \ldots, t_n\}$ be the values of attribute A that occur in the database. For any value t, we define IDF(t) as shown in Equation (2) (where h is the *bandwidth* parameter, to be defined later).

$$IDF\,(t) = \log\left(\frac{n}{\sum_{i}^{n} e^{-\frac{1}{2}\left(\frac{t_i - t}{h}\right)^2}}\right) \qquad (2)$$

Intuitively, the denominator in Equation (2) represents a numeric extension of the concept of "frequency" of t, i.e. the sum of "contributions" to t from every the other point t_i in the database. These contributions are modeled as (scaled) Gaussian distributions, so that the further t is from t_i, the smaller is the contribution from t_i.

We then define the similarity between t and q as shown in Equation (3), i.e. as the density at t of a Gaussian distribution centered at q, scaled by IDF(q).

$$S(t,q) = e^{-\frac{1}{2}\left(\frac{t-q}{h}\right)^2} IDF(q) \qquad (3)$$

As an illustration, consider the scenario where the numeric data resembles categorical data: there are n_t tuples in the database with value t, and the remaining $n - n_t$ tuples have values far from t. If q belongs to the latter, then it is easy to see that $S(t, q)$ is almost 0. Whereas, if q also has the value t, then $S(t, q)$ degenerates to $\log(n/n_t)$, which is exactly the formula for categorical data.

The above numerical extensions to IDF have been derived using *kernel density estimation* techniques [25]. A popular estimate for the bandwidth is $h = 1.06\sigma n^{-1/5}$, where σ is the standard deviation of $\{t_1, t_2, \ldots, t_n\}$. For theoretical justification of these extensions, see [2].

3.3 Other generalizations of IDF Similarity

In Section 3.1 we had assumed a query model where C-conditions are conjunctions of atomic conditions such as "$A_k = q_k$". A useful generalization is the ability to specify a range/set of values for numerical/categorical attributes.

Let query Q have a C-condition "C_1 AND … AND C_m", where each C_k is generalized as "A_k IN Q_k", where Q_k is a set of values for categorical attributes, or a range $[lb,ub]$ for numeric attributes. For uniformity of notation, we use IN to also specify numeric ranges, e.g. "A_k IN $[lb,ub]$", instead of the more standard BETWEEN. Let $T = <t_1,\ldots,t_m>$ be any tuple. To generalize the similarity function SIM(T,Q) of Equation (1), we define similarity between t_k and Q_k as the maximum similarity coefficient between t_k and all values in Q_k. The generalized similarity function is shown in Equation (4).

$$SIM(T,Q) = \sum_{k=1}^{m} \max_{q \in Q_k} S_k(t_k, q) \qquad (4)$$

In defining Equation (4), we considered the alternative of using *avg* instead of *max*. However, this can lead to an unintuitive scenario where a tuple that completely satisfies the selection condition may be ranked lower than a tuple that only partially satisfies the selection condition. A more detailed discussion on this issue is omitted.

Thus far, our query model assumes that values for *all attributes* are specified in a query. In most real queries it is unlikely that all attributes are specified. We refer to these as *missing attributes*. Our approach is to restrict similarity calculations only to the attributes specified by the query, i.e., we only consider the projection of the database on the columns that are referenced in the query. This has parallels with approaches in IR, where similarity is calculated only using words that appear in the query. It is only when numerous tuples have the same similarity score that we use missing attributes to break ties. Details of this scenario are discussed in Section 5.

4. QF Similarity: leveraging workloads

While IDF Similarity can be very useful in many applications of database ranking, it nevertheless has several shortcomings that need to be addressed. In this section we first discuss these shortcomings, and then discuss *QF Similarity*, a ranking function that leverages workload information to overcome these shortcomings.

The following examples show that a data value may be important for ranking purposes irrespective of its frequency of occurrence in the database.

Example 1: *In a realtor database, more homes are built in recent years such as 2000 and 2001 as compared to earlier years such as 1980 and 1981. Thus recent years have smaller IDF. Yet the demand for newer homes is usually more than that for older homes.*

Example 2: *In a bookstore database, the demand for an author is due to factors other than the number of books she has written (such factors may include for example, number of favorable reviews).*

We note that the above problems can be solved by a domain expert who can define a more accurate similarity function (e.g. by giving more weight to later years in Example 1). However, this can be highly dependent on the application, so we do not attempt a general discussion here. Instead, we show how to derive the similarity function *automatically* by analyzing other more easily available knowledge sources, such as past usage patterns of the database (i.e. workload). An important point is that our techniques *do not require as inputs both workload queries and their correctly ranked results*; getting the latter information is tedious and involves user feedback, whereas gathering queries only is relatively easy since profiling tools exist on most commercial DBMS that can log each query string that executes on the system.

In the next subsection we describe a simple version of QF Similarity, in which the importance of attribute values is determined by the frequency of their occurrence in the workload. We follow this up in Section 4.2 with a more sophisticated variant of QF Similarity, in which similarity between pairs of different categorical attribute values can also be derived from the workload. In Section 4.3 we briefly discuss a hybrid strategy, *QFIDF Similarity*, where we combine information from the workload as well as the data to derive importance of attribute values.

4.1 Query frequencies of attribute values

The idea behind the simple variant of QF Similarity is that the importance of attribute values is directly related to the frequency of their occurrence in query strings in the workload. Consider the realtor database discussed in Example 1. It is reasonable to assume that there are more queries requesting for newer homes than for older homes. Thus the frequency of the year 2001 appearing in the workload will be more than of the year 1981. A simple idea that takes advantage of this observation is to record the frequency of attribute values appearing in the workload, and then let similarity coefficients depend on these frequencies. We make this precise as follows.

Assume for simplicity only categorical data; we discuss numeric data in Section 4.3. Let RQF(q) be the raw frequency of occurrence of value q of attribute A in the query strings of the workload. Let RQFMax be the raw frequency of the most frequently occurring value in the workload. Let the *query frequency*, QF(q) be defined as RQF(q)/ RQFMax. We define the similarity coefficient $S(t,q)$ as QF(q) if $q = t$, and 0 otherwise.

We note that QF(q) has resemblance with the classical term frequency TF(q), except that it is the frequency of q over the entire workload rather than in the specific query.

4.2 Similarity between different attribute values

In this section we discuss a more sophisticated variant of QF Similarity. While the simple QF Similarity discussed in Section 4.1 can resolve Examples 1 and 2, it cannot resolve the following example; in fact, none of the ranking functions discuss so far can resolve Example 3.

Example 3: *In an automobile database, a HONDA ACCORD and a TOYOTA CAMRY are very dissimilar as measured by any of the previous similarity functions, since the similarity coefficients S(TOYOTA, HONDA) and S(CAMRY, ACCORD) are both 0. However, intuitively we know that the two cars are quite similar, e.g. they are family sedans, of comparable quality, and targeted to the same market segment.*

To resolve this problem, we need similarity coefficients that are non-zero even when the pair of categorical values is different. For example, S(TOYOTA, HONDA) may be 0.8, while S(TOYOTA, FERRARI) may be 0.1.

We discuss an approach for deriving such similarity coefficients by leveraging workload information in further ways. The intuition is that if certain pairs of values $t <> u$ often "occur together" in the workload, they are similar. For example, there may be queries with C-conditions such as "MFR IN {TOYOTA, HONDA, NISSAN}". Such workloads suggest that these manufacturers are more similar to each other than to, say FERRARI.

Let $W(t)$ be the subset of queries in workload W in which categorical value t occurs in an IN clause. The *Jaccard coefficient* [29] measures the similarity between the two sets $W(t)$ and $W(q)$ as shown in Equation (5).

$$J(W(t),W(q)) = \frac{|W(t) \cap W(q)|}{|W(t) \cup W(q)|} \qquad (5)$$

The similarity coefficient between t and q is defined as this Jaccard coefficient, scaled by the QF factor as shown in Equation (6).

$$S(t,q) = J(W(t),W(q))QF(q) \qquad (6)$$

Note that in the limit when $W(t)$ is very similar to $W(q)$, $S(t, q)$ degenerates to QF(q), which is exactly the formula for $S(t, q)$ in Section 4.1.

4.3 Discussion

Pair-wise similarity between different attribute values can be determined by other techniques in addition to analyzing IN clauses of queries. For example, perhaps there have been several recent queries in the workload by a specific user who has repeatedly requested for TOYOTA and HONDA cars in succession. Finding such co-occurrence of values over sequences of queries by specific users is the subject of ongoing work.

Numerical values that occur in the workload can also benefit from query frequency analysis. For example, in the realtor database, if certain home prices are very frequently specified by workload queries, it is reasonable to treat them (and nearby values) as important values during similarity computations. Thus, as we did for IDF( ) in Section 3.2, we have to compute a *smooth* query frequency function QF( ).

QF Similarity is purely workload-based, i.e. it does not use the data at all. This may be a disadvantage in situations where we have insufficient or unreliable workloads. We experimented with a hybrid ranking function, *QFIDF Similarity*, where we combined IDF and QF weights by multiplying them, i.e., $S(t, q)$ = QF(q)*IDF(q) when $t = q$, and 0 otherwise. (In this formula we define QF(q) = (RQF(q)+1)/ (RQFMax+1) so that even if a value is never referenced in the workload, it gets a small non-zero QF). Using multiplication to combine the two factors is inspired by the TF*IDF factors in the original TF-IDF ranking function [4]. The resulting function noticeably improved ranking quality in certain cases (see Section 7).

5. The many answers problem: breaking ties

In the previous two sections we have focused mainly on the empty answers ranking problem. In this section we discuss differences between the empty answers and many answers problem, and describe how our ranking functions can be extended to handle the latter problem.

For ranking the results of a query that produces many answers, IDF Similarity and QF Similarity may sometimes run into the following problem: many tuples may tie for the same similarity score and thus get ordered arbitrarily. For example, consider a query Q with a selection condition of the form "$A_1 = q_1$ AND … AND $A_i = q_i$" where $i < m$ (i.e. some of the columns, $A_{i+1}, \ldots, A_m$ have not been specified by the query). Suppose many tuples in the database satisfy this selection condition. We note that the projection of each of these tuples along the attributes specified in the query is the same, i.e. $<q_1, \ldots, q_i>$. Thus SIM(T, Q,) for each answer tuple T will be the same, whether we use IDF Similarity or QF Similarity.

We observe that this problem can also arise in the empty answers problem: the top one or two tuples may have distinct similarity scores, followed by a large group of tuples that share the same similarity score. In general, if we only use the attributes specified in the query for ranking purposes, our similarity functions will partition the database into several equivalence classes, where tuples within each class have the same similarity score.

To break ties among the tuples in each class, it is thus necessary to look beyond the attributes specified in the query, i.e. *missing attributes*. Investigating attributes beyond what has been specified by the query is particularly tricky since the ranking function does not know what the user's preferences for the missing attributes are. The final ranking function could be a composition of weights of the missing attribute values. The problem thus is how do we in a principled manner determine these weights?

Our approach is to determine weights of missing attribute values that reflect their "global importance" for ranking purposes, since we cannot possibly relate them to the preferences of the specific user who has issued the query. For example, suppose we seek homes with four bedrooms in a realtor database. Since there are many homes satisfying this condition, we examine attributes other than number of bedrooms to rank the result set. If we knew that "BELLEVUE" is a more important location than "CARNATION" in a global sense, we would rank four bedroom homes in Bellevue higher than four bedroom homes in Carnation.

We use workload information to determine global importance of missing attribute values. The intuition is that if Bellevue is truly a popular neighborhood, the workload will contain many more queries requesting for Bellevue homes compared to Carnation homes. More formally, we define the global importance of missing attribute value t_k as $\log(QF_k(t_k))$, and extend QF Similarity to use the quantity $\sum\log(QF_k(t_k))$ to break ties in each equivalence class (larger this quantity[1], higher the rank of the tuple) where the summation is over missing attributes.

Extending IDF Similarity by using IDF values instead of QF values of missing attributes to break ties presents challenges. One possibility is to rank tied tuples higher if their missing attribute values have large IDF, i.e. occur infrequently in the database. But this gives rise to the undesirable scenario where, all else being equal, homes that occur in uncommon neighbourhoods are ranked before homes that occur in more common neighbourhoods. An alternative strategy is to rank tied tuples higher if their missing attribute values have *small* IDF, i.e. occur more frequently in the database. This will work well in the realtor example above, as homes in more popular neighbourhoods will be ranked higher than homes in strange neighbourhoods. Although more robust than the previous strategy, there are situations where this approach is also flawed. For example, suppose the database had a Boolean attribute "Deck". Since only a small fraction of homes have decks, this ranking function will rank higher homes that *do not* have decks, which is contrary to intuition since a deck is usually a desirable feature.

In Section 7 we discuss experiments which show that for ranking queries with numerous answers, the quality of QF Similarity is noticeably better than the quality of IDF Similarity (both functions extended as described above).

6. Implementation

In this section we discuss the implementation of the pre-processing and query processing components of our database ranking system.

6.1 Pre-processing component

The main task of the pre-processing component is to compute and store a representation of the similarity function in auxiliary database tables. Computing IDF(t) (resp. QF(t)) for all categorical values t involves scanning the database (resp. scanning/parsing the workload) to compute frequency of occurrences of values in the database (resp. workload), and storing the results in auxiliary tables. For a numeric attribute, since we do not know what value q will be specified by a query, we cannot pre-compute IDF(q) (resp. QF(q)); thus we have to store an approximate representation of the smooth function IDF( ) (resp. QF( )) so that the function value at any q can be retrieved at runtime. We mention that since kernel density estimation techniques have been used to smoothen these functions, they can be approximated as histograms in linear time by the WARPing method [25]; we omit further details from this paper. The approximated functions are stored in auxiliary tables.

For identifying similarity coefficients for QF Similarity between all pairs of values u and v of any attribute A (Section 4.2), we avoid space/time requirements quadratic in the size of A's domain by only storing similarity coefficients that are above a certain threshold. They can be efficiently computed using a *frequent itemset* algorithm [3].

6.2 Query processing component

The main task of the query processing component is, given a query Q and an integer K, to efficiently retrieve the Top-K tuples from the database using one of our ranking functions. We assume that the ranking function has already been extracted in a pre-processing phase (Section 6.1). We focus exclusively on the empty answers problem; the query processing challenges of the many answers problem is part of our ongoing work.

---

[1] If $QF_k(t_k)$ is viewed as the probability of occurrence of value t_k in a random query, the quantity $\sum\log(QF_k(t_k))$ represents the *log-likelihood* of a query that requests the remaining values of T, which can be construed as the "importance" of T for ranking purposes.

Our objective was to use the available functionality of a traditional SQL DBMS for solving this Top-K problem. Thus, we decided not to adopt techniques that build specialized *multi-dimensional indexes* for arbitrary similarity spaces (e.g. Fast-Map [16]). Another possible approach is to use *inverted lists*, a popular data structure in information retrieval. We discarded this approach from further consideration since (a) this requires the presence of indexes on *all* columns specified in a query, which may be impractical and (b) it does not work for numeric data.

6.2.1 Handling a simpler query processing problem

We first focus on a much simpler version of the query processing problem; the more general problem is discussed in Section 6.2.2.

- **Inputs**: (a) a database table R with m *categorical* columns, clustered on key column *TID*, where standard database indexes exist on a subset of columns, (b) A query expressed as a conjunction of m *single-valued* conditions of the form $A_k = q_k$., and (c) an integer K.
- **Similarity function**: We assume a very simple similarity function which we call *Overlap Similarity*. This function measures the number of values in the tuple that match the corresponding values in the query. In Section 6.2.2 we discuss implementations of the more general similarity functions developed earlier in this paper.
- **Output**: The Top-K tuples of R most similar to Q.

We discuss two solutions to this restricted problem.

Traditional implemention of Top-K operator: Many SQL database systems (e.g. Microsoft SQL Server) support Top-K query processing features. The SQL for the above restricted problem is shown in Figure 1.

```
SELECT   TOP K      R.*
FROM     R
ORDER BY
         ((CASE WHEN R.A1 = q1 THEN 1 ELSE 0 END) +
          (CASE WHEN R.A2 = q2 THEN 1 ELSE 0 END) +
                            …
          (CASE WHEN R.Am = qm THEN 1 ELSE 0 END))
DESC
```

Figure 1: Top-K query in SQL

Most database systems would create a computed column (created on the fly in a pipelined manner) corresponding to the ranking function (e.g., in the ORDER BY clause in Figure 1) and then use a *Sort_TopK* operator, i.e., sort the relation to get Top-K results. Recent papers have focused on how to efficiently implement a Sort_TopK operator [8, 9]. It is important to note that the assumed semantics of Top-K is nondeterministic, i.e., ties are broken arbitrarily.

An index-based Top-K implementation: In most SQL systems, the above algorithm cannot leverage any available indexes and has to scan every database tuple. However, we observe that the Overlap Similarity function (in fact, *all* similarity functions discussed in this paper) satisfies a useful *monotonic property*: if T and U are two tuples such that for all k, $S_k(t_k, q_k) \leq S_k(u_k, q_k)$, then SIM($T$, Q) $\leq$ SIM(U, Q). This enables us to adapt Fagin's *Threshold Algorithm* (*TA*) and its derivatives [7, 15] to retrieve the Top-K tuples without having to process all tuples of the database.

To adapt TA for our purposes, we have to implement two types of access methods: (a) *sorted access* along any attribute A_k, in which TIDs of tuples can be efficiently retrieved one-by-one in order of decreasing similarity of their A_k attribute value from q_k, and (b) *random access*, in which the entire tuple corresponding to any given TID can be efficiently retrieved. In brief, Fagin's algorithm performs sorted access along each attribute in "lock-step", retrieves the complete tuples corresponding to the TIDs seen using random access, and maintains a buffer of the Top-K tuples seen thus far. The monotonic property of the similarity function allows the use of an early *stopping condition*, by which the algorithm can detect that the final Top-K tuples have been retrieved before all tuples have been processed.

We leverage available database indexes such as B+ trees to efficiently implement these two access methods. Since it is unrealistic to assume that indexes are always present on all attributes specified by any query, we adapt a derivative of TA [7] that works even if sorted access is not available on some attributes. Our resulting adaptation, called the *Index-based Threshold Algorithm*, or *ITA*, is shown in Figure 2.

Assume that indexes are present on columns A_1, ..., A_p and not present on columns A_{p+1}, ..., A_m. The essence of ITA is to do index seeks on orderings L_1, ..., L_p where each L_k is defined as an ordering of tuples where tuples with $A_k = q_k$ precede the tuples with $A_k <> q_k$. We use the following terminology: (a) *TupleLookup*(TID), where the complete tuple for the given TID is retrieved from R, and (b) *IndexLookupGetNextTID*(L_k), where given an ordering L_k of a column A_k, the next matching TID of R is retrieved using the available index on that column. These operations are respectively equivalent to the random access and sorted access operations described earlier. TupleLookup(TID) can be implemented by traditional indexes in a relational databases. Efficient implementation of IndexLookupGetNextTID(L_k) using the indexing support in relational database engine requires more care; we omit further details from this paper.

Index seeks on L_1, ..., L_p may be interleaved or ordered in a variety of ways based on heuristics or data statistics. The most important step is the *stopping condition*, i.e. identifying that no more index seeks on any column will be needed. We discuss this next.

Stopping Condition: We define a *hypothetical* tuple by taking the "current" value $a_1, \ldots, a_p$ for $A_1, \ldots, A_p$ corresponding to index seeks on $L_1, \ldots, L_p$ and using the values $q_{p+1}, \ldots, q_m$ for the remaining columns. This creates the very best tuple we can hope to find in the data that is yet to be seen. If the similarity of this hypothetical tuple to the query is no more than the tuple in the Top-*K* buffer with the lowest similarity, the algorithm successfully terminates.

ITA: Index-based Threshold Algorithm

Initialize Top-*K* buffer to empty

REPEAT

FOR EACH k = 1 TO p DO

1. TID_k = IndexLookupGetNextTID(L_k)
2. T_k = TupleLookup(TID_i)
3. Compute value of ranking function for T_k
4. If rank of T_k is higher than the lowest ranking tuple in the Top-*K* buffer
then update Top-*K* buffer
5. If *stopping condition* has been reached then EXIT

END FOR

UNTIL indexLookupGetNextTID(L_1) … indexLookupGetNextTID(L_p)

are all completed

Figure 2: Index-based Threshold Algorithm for Top-*K* query processing

Although *ITA* does not require indexes on all columns referenced by the query, fewer indexes imply that the algorithm may need to do more tuple lookups using TIDs before it can terminate. We also note that the same tuple may be retrieved several times via TID lookup because its TID may be encountered multiple times during index lookups along different columns. The main disadvantage of this approach is that it introduces random accesses, and this can have an adverse affect on performance if too many index lookups are needed (see Section 7.3.2).

6.2.2 Handling more general query processing

Our basic framework for Top-*K* query processing extends to the more general similarity functions developed in the paper. These extensions are described next.

Presence of QFIDF tables: Let us consider query processing when we use one of the more general ranking functions described in Sections 3 and 4.1. In addition to the base table *R*, several small auxiliary tables, one per categorical column of *R*, have been created during preprocessing that contain information about the similarity function. We call these tables *QFIDF tables*. We assume that each such QFIDF table has two attributes <ColVal, QFIDFVal> and is clustered on the ColVal attribute. ColVal contains all distinct values of the specific database column that corresponds to this QFIDF table, while QFIDFVal contains the respective weights (for ranking purposes) of these distinct values. The specific QFIDFVal weights depend on the ranking function we adopt, e.g., IDF, QF or QF*IDF.

Let us consider the impact of this generality on the two Top-*K* implementations described in the previous subsection. First, to know the QFIDFVal weights, we need to look up the QFIDF tables. Since the QFIDF lookup is based only on the conditions in the query and is independent of the data tuple, this may be accomplished by an initialization step. The retrieved QFIDFVal weights are then used during subsequent processing in the traditional Top-*K* computation for the creation of the computed column based on the ranking function.

The above initialization step is also common to ITA. Subsequent computation in ITA remains unaffected, except that the ranking function computations have to take into account the retrieved QFIDFVal weights.

Numerical columns: We consider the important case when some of the database columns are numeric.

We adapt ITA for numeric conditions in a query as follows. Suppose the query has a condition $A_k = q_k$ for a numeric column A_k. Because A_k is numeric, unlike categorical cases, it is now possible to return "nearby matches" based on increasing value of $|A_k - q_k|$ once no more exact matches $A_i = q_k$ exist in the data. We perform two index scans on A_k: one that retrieves TIDs of tuples with values greater than q_k in increasing order, and another that retrieves TIDs of tuples with values lesser than q_k in decreasing order. We then pick the TIDs from the merged stream. Once we have ensured that each index on a numeric attribute can produce tuples in the order of decreasing similarity in the above fashion, the rest of the implementation is the same as what has been described for categorical attributes.

Handling numeric conditions in a query using traditional Top-*K* SQL is straightforward and is omitted from this paper.

Other generalizations: ITA can be extended to handle other generalizations, such as IN and range conditions in the query (Section 3.3), non-zero pair-wise similarity coefficients (Section 4.2), and for breaking ties among tuples (Section 5). Further details of these extensions to ITA may be found in [2].

When our ranking order is over the result of a relational query, defined over a set of tables, additional challenges arise. Appropriate materialized views can greatly enhance applicability of our techniques. Furthermore, indexes on base tables can be leveraged but the trade-off in query processing and optimization is increasingly more complex.

7. Experiments

We implemented the techniques described in this paper and conducted experiments to evaluate their effectiveness. All experiments were run on a machine with an x86 450 MHz processor with 256 MB RAM and an internal 5GB hard drive running Microsoft Windows 2000 and Microsoft SQL Server 2000.

We first tested the ranking quality as well as performance of the following similarity functions on queries with empty/few answers: *Overlap* (Section 6.2.1), *IDF* (Section 3), *QF* and *QFIDF* (Section 4). We then tested the extensions to IDF and QF for breaking ties among tuples (Section 5). Finally, we compared the query processing performance of the threshold algorithm using indexes (ITA) against SQL Server Top-*K* using these similarity functions (Section 6).

7.1 Summary of results

Quality results

- For queries with empty answers, QFIDF produced the best rankings, followed by QF, then IDF, and finally Overlap.
- For queries with empty answers, the ranking quality of QF improves with increasing workload size.
- For queries with numerous answers, QF produced better rankings than IDF.

Performance results

- The preprocessing time and space requirements of all our techniques scale linearly with data size.
- When all indexes are present, ITA is more efficient than SQL Server Top-*K* for all our similarity functions.
- Even when a subset of indexes is present, ITA can perform well; the performance is strongly determined by how effective the algorithm is in reducing the number of processed tuples.

7.2 Quality experiments

Evaluating and comparing the quality of different database ranking alternatives is challenging. Unlike Information Retrieval which relies on extensive user studies and available benchmarks (such as the TREC collection [28]), such infrastructure is not available today for evaluating database ranking. Nonetheless, we conducted user studies on several real databases.

In this paper we only report results for one real database, *Realtor*, which is part of a large real estate database from http://homeadvisor.msn.com. We first collected about 72,000 tuples representing homes for sale in Washington State. Of these, we retained 4099 tuples representing homes for sale in the Seattle Eastside. We chose a mixture of 10 categorical and numerical attributes for our experiments: City, Deck, Fenced, Culdesac, Price, Datebuilt, Bedrooms, Sqft. For building a workload, we requested eight people, some of them actual homeowners in Seattle Eastside, to provide us with queries that they would execute if they wanted to buy a home. An example of a typical query was: "SELECT * FROM homes WHERE Bedrooms > 3 AND Bathrooms > 2 AND Price < 350000"; the user commented he had in mind young families with not too much money, but have children and hence need space. We collected a total of 84 queries, each typically referencing 2-5 attributes. We used five people to provide test queries to evaluate the quality results. We selected a mix of 6-10 test queries similar to the ones provided by users during workload generation. We first describe a few sample results informally, and then present a formal evaluation of the ranking quality.

7.2.1 Informal quality results

All ranking functions produced rankings that were quite intuitive and reasonable. IDF was obviously superior to Overlap in several queries; for example when requesting for homes with price $300k located on a cul-de-sac, the latter attribute value was given more importance since only a small fraction of homes (around 15%) are located on cul-de-sacs, whereas a much larger fraction of homes have prices close to $300k.

However, there were several interesting examples where IDF was unable to obtain the rankings generated by the users. When requesting for a home located on a cul-de-sac and with a fenced yard, IDF was unable to distinguish between the importance of these two values, as both had approximately the same relative frequencies in the database (around 15% of homes also had fenced yards). But to the users a cul-de-sac location is more important than a fence (because fences can be easily constructed whereas a home location cannot be changed). QF Similarity obtained better rankings as even in our modest-sized workload there were many more queries that requested cul-de-sacs than fences.

7.2.2 Formal quality results

We now present a formal evaluation of the ranking quality produced by the ranking functions. Since it would have been very tedious to have users rank the entire database for each query, we used the following strategy. For each test query Q_i we generated a list H_i of 25 tuples likely to contain a good mix of "relevant" and "irrelevant" tuples to the query (we omit details from this paper, but we did this by ranking the entire database using these ranking functions and mixing a few highly ranked tuples with a few randomly selected tuples). Finally, we presented the queries along with the corresponding lists (with tuples randomly permuted) to each user in our study. Each user's responsibility was to mark each tuple in H_i as relevant or irrelevant to the query Q_i. We then applied our ranking functions against the test queries.

For formally comparing the ranking quality of the various ranking functions with the human responses, we used a standard *collaborative filtering metric R* to measure ranking quality (Equation (7)). In the equation, r_i is the subject's preference for the *i*th tuple in the ranked list returned by the ranking function (1 if it is marked relevant, and 0 otherwise). The intuition behind the *R* metric is that if relevant tuples are ranked low, they contribute less to the value of *R* with exponential decay (see [2] for further discussion on the *R* metric).

$$R = \sum_i \frac{r_i}{2^{\left(\frac{i-1}{9}\right)}} \qquad (7)$$

We next present the *R* metric values obtained in various quality experiments (*R* values are normalized by dividing by the maximum possible value for *R*).

Comparing quality of different ranking functions: In Figure 3 we present the average *R* metric for each ranking function on the test queries.

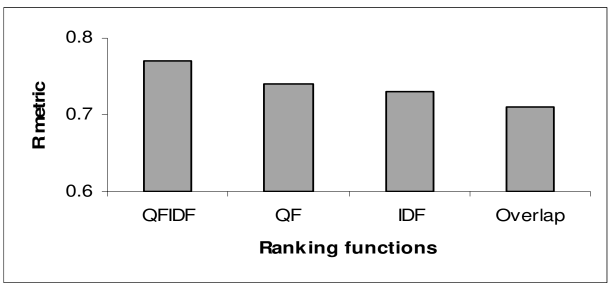

Figure 3: Quality of various ranking functions on Realtor database

The best ranking function in average ranking quality was QFIDF, followed by QF, then IDF, and finally Overlap. All ranking functions did better than a naïve ranking function that retrieves *K* random tuples (this naïve function's average *R* value is 0.66, not shown in the chart). We mention that the differences in quality are likely to have been more significant if our users were able to score many more than 25 tuples per query.

Quality versus workload size:

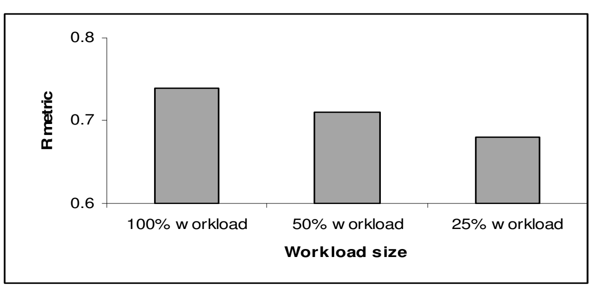

Figure 4: Ranking quality of QF Similarity on Realtor database as workload size varies

We explored the dependence of quality to workload size in QF Similarity by training it on randomly sampled fractions of the entire workload. The results (Figure 4) indicate that larger workloads lead to better quality, because they are likely to contain more accurate QF values.

Comparing quality on queries with many answers: We compared the quality of IDF Similarity with QF Similarity, both extended to use missing attributes to break ties as discussed in Section 5. For this experiment, our users especially created 6 test queries whose selection conditions were satisfied by many tuples (order of hundreds). QF has better ranking quality (R = 0.76) than IDF (R = 0.68). Again, we emphasize that the difference in quality is likely to have been more significant if users were able to score many more than 25 tuples per query.

7.3 Performance experiments

We evaluated the pre-processing and query processing performance of our ranking algorithms. We used the Realtor database for Washington State with 72,000 tuples (Section 7.2), as well as synthetic databases generated by using the publicly available program [11] for generating the popular TPC-H databases [27] with differing data skew. For our experiments we generated the *lineitem* fact table with 600,000 rows and varying skew parameter *z*. Here we report results for $z = 2.0$ (similar results occurred for values of *z* from 0.5 to 3). We treated all 17 attributes as categorical. There are 6 attributes with less than 10 distinct values, 3 attributes with order of tens distinct values, 5 attributes with hundreds, and 3 with thousands.

Note that although we use TPC-H databases, the workloads used in our experiments are quite different from standard TPC-H benchmarks. Thus, our results do not reflect the TPC-H benchmark numbers.

7.3.1 Preprocessing performance experiments

We omit reporting results as the preprocessing was very efficient: a scan of the table *R* in case of IDF Similarity, a scan/parse of the workload in case of QF Similarity (and variants), accompanied by the creation of the appropriate small auxiliary tables.

7.3.2 Query processing performance experiments

We report query processing experiments for the ranking functions developed in Sections 3 and 4. We do not report query processing performance experiments for the many answers problem (Section 5) as it is part of ongoing work.

We implemented three versions of our index-based threshold algorithm ITA: ITA-OL that uses Overlap Similarity, ITA-IDF that uses IDF Similarity and ITA-QF that uses QF Similarity. (Performance results for ITA-QFIDF are essentially the same as for ITA-QF and have been omitted). For comparison, we used the SQL Server's Top-*K* mechanism to retrieve the Top-*K* tuples for all of our similarity functions. For the first two parts of the experiment, non-clustered indexes were available on all columns referenced in the queries.

Varying number of attributes in query: We used the TPC-H database and generated 5 workloads W_1 through W_5 of 100 queries each (W_i is a workload containing queries each referencing i attributes). The attributes and values in a query were randomly selected from the underlying database.

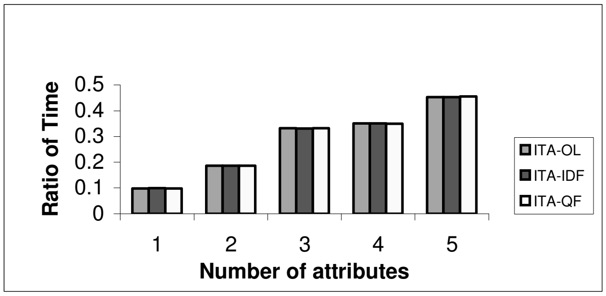

Figure 5: Time taken by ITA compared to SQL Server's Top-*K* processing as number of attributes varies

As Figure 5 shows, the running times (as a ratio of time taken by SQL Server's Top-*K* processing) increased as the number of attributes increased, which was expected. We also observed the query performance of all the three techniques to be almost identical to each other, but significantly better than SQL Server's Top-*K* processing (as the number of tuples processed was orders of magnitude less than SQL Server's Top-*K* processing).

Varying *K* in Top-*K*: Here we used the TPC-H database and a workload with 100 queries. The number of attributes in a query was randomly selected between 1 and 5. Figure 6 shows that all the techniques had almost identical performance (ITA-OL was slightly faster than both ITA-IDF and ITA-QF as it involves the least processing during querying) and outperformed SQL Server's Top-*K* processing by almost a factor of 5.

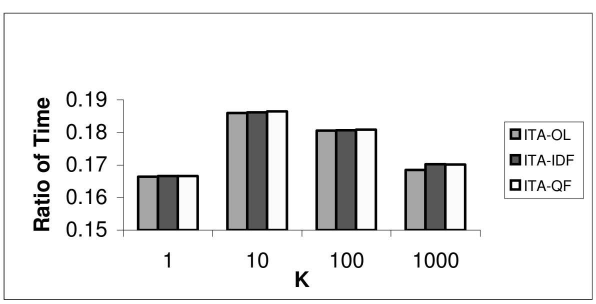

Figure 6: Time taken by ITA compared to SQL Server's Top-*K* processing as *K* varies

Note the decrease in time when *K* is increased from 10 to 100; this is because the time taken for SQL Server's Top-*K* increased as well (extra time was spent in maintaining the larger Top-*K* buffer).

Varying number of indexes in database: We investigated the performance when only some of the columns specified in a query have indexes. For a given number of available indexes *N* for a query *Q* we used two strategies: (a) ITA-QF-Exhaustive where the best running time was selected from amongst all possible subsets of *N* column indexes relevant for *Q* and (b) ITA-QF-Random where the *N* indexes to be retained for *Q* were randomly selected from amongst all relevant indexes for *Q*. We report results on the Realtor database with 72,000 tuples for this experiment. We generated a workload of 100 queries (each query referenced 4 attributes; the specific attributes and values were selected randomly from the underlying database). We fixed $K = 10$ and varied the number of available indexes *N* for each query from 4 down to 1.

Number of available Indexes *N*	Ratio of Time for ITA-QF-Exhaustive	Ratio of Time for ITA-QF-Random
4	0.13	0.13
3	0.10	2.90
2	0.12	4.30
1	2.76	7.80

Figure 7: Time taken by ITA compared to SQL Server Top-*K* processing as indexes are dropped

Figure 7 shows the running time of ITA-QF-Exhaustive and ITA-QF-Random for different values of *N*, expressed as a ratio of the time taken by SQL Server's Top-*K* processing. We observed that as the number of available indexes was decreased from 4 to 2, the running time of ITA-QF-Exhaustive remained almost the same, yet significantly (an order of magnitude) better than SQL Server's Top-*K* processing. This is due to the fact that the available indexes can still be used to answer the Top-*K* queries efficiently. At $N = 1$ there was a steep increase in running time (outperformed by SQL Server's Top-*K* processing) even though the number of tuples processed was still about 30% of the total tuples. This is due to the significantly higher cost of random access in databases compare to sequential access. We observed that the running time of ITA-QF-Random was much (3-8 times) worse than SQL Server's Top-*K* processing for $N = 3$, 2 and 1. ITA-QF-Random could not leverage the stopping criteria effectively; it accessed a large number of tuples (more than 30% of total data).

These experiments demonstrate that ITA-QF can be efficient even when a subset of indexes is available, but the performance is strongly tied with the nature of subset. The choice of determining such an optimal subset of indexes is a part of ongoing work.

8. Conclusions

In this paper, we have presented our experience in attempting to build a generic automated ranking infrastructure for SQL databases. This is consistent with our research philosophy of seeding the relational database management infrastructure with functionality necessary and useful for data exploration.

Our attempt was to extend TF-IDF based techniques from information retrieval to numerical and mixed data, as well as develop techniques of workload tracking as a weak form of collaborative filtering. Our approaches have shown promise, and are worthy of further investigation, especially more conclusive user studies. Equally important is to develop benchmarks. While TREC has served the IR community wonderfully well, there is no such infrastructure to move forward this nascent field.

We were also aware that a meaningful solution has to take into account the impact on query processing. Our proposals lead to an implementation of the ranking function that exploits indexed access by drawing on insights from Fagin's Threshold Algorithm.

Are we trying to solve too hard a problem? One could argue that ranking is extremely domain and/or user specific and we cannot hope to automate such a difficult task. We remind the readers that one could have raised similar concerns about IR ranking as well. To explore what information a database system can intelligently bring to bear at a modest cost to solve database ranking to reduce the burden of an application designer or user is a dream worth pursuing.

Acknowlegment

We thank Nico Bruno for his insightful comments on the algorithms and presentation of the paper.

References

[1] S. Agrawal, S. Chaudhuri and G. Das. DBXplorer: A System for Keyword Based Search over Relational Databases. ICDE 2002.
[2] S. Agrawal, S. Chaudhuri, G. Das and A. Gionis. Automated Ranking of Database Query Results. Technical Report, Microsoft Research, in preparation.
[3] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen and A. I. Verkamo. Fast Discovery of Association Rules. Advances in Knowledge Discovery and Data Mining, 1995.
[4] R. Baeza-Yates and B. Ribeiro-Neto. Modern Information Retrieval. ACM Press, 1999.
[5] J. Breese, D. Heckerman and C. Kadie. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. 14th Conference on Uncertainty in Artificial Intelligence, 1998.
[6] N. Bruno, L. Gravano, and S. Chaudhuri. Top-K Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. ACM Transactions on Database Systems (TODS), vol. 27, no. 2, June 2002.
[7] N. Bruno, L. Gravano, A. Marian. Evaluating Top-K Queries over Web-Accessible Databases. ICDE 2002.
[8] M. J. Carey and D. Kossmann. On Saying "Enough Already!" in SQL. SIGMOD 1997, 219-230.
[9] M. J. Carey and D. Kossmann. Reducing the Braking Distance of an SQL Query Engine. VLDB 1998. 158-169.
[10] K. Chakrabarti, K. Porkaew and S. Mehrotra. Efficient Query Ref. in Multimedia Databases. ICDE 2000.
[11] S. Chaudhuri and V. Narasayya. Program for TPC-D Data Generation with Skew. http://research.microsoft.com/dmx/AutoAdmin.
[12] W. Cohen. Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity. SIGMOD, 1998.
[13] W. Cohen. Providing Database-like Access to the Web Using Queries Based on Textual Similarity. SIGMOD 1998.
[14] R. Fagin. Fuzzy Queries in Multimedia Database Systems. PODS 1998.
[15] R. Fagin, A. Lotem and M. Naor. Optimal Aggregation Algorithms for Middleware. PODS 2001.
[16] C. Faloutsos and K-I. Lin. Fastmap: A Fast Algorithm for Indexing, Data mining and Visualization of Traditional and Multimedia Datasets. SIGMOD 1995.
[17] N. Fuhr. A Probabilistic Framework for Vague Queries and Imprecise Information in Databases. VLDB 1990.
[18] N. Fuhr. A Probabilistic Relational Model for the Integration of IR and Databases. ACM SIGIR Conference on Research and Development in Information Retrieval, 1993.
[19] W. Kießling and G. Köstler. Preference SQL - Design, Implementation, Experiences. VLDB 2002.
[20] W. Kießling. Foundations of Preferences in Database Systems. VLDB 2002.
[21] A. Motro. VAGUE: A User Interface to Relational Databases that Permits Vague Queries. TOIS 6(3) 1988, 187-214.
[22] Z. Nazeri, E. Bloedorn and P. Ostwald. Experiences in Mining Aviation Safety Data. SIGMOD 2001.
[23] M. Ortega-Binderberger, K. Chakrabarti and S. Mehrotra. An Approach to Integrating Query Refinement in SQL, EDBT 2002, 15-33.
[24] Y. Rui, T. S. Huang and S. Merhotra. Content-based Image Retrieval with Relevance Feedback in MARS. IEEE Conf. on Image Processing, 1997.
[25] B. W. Silverman. Density Estimation. Chapman and Hall. 1986.
[26] A. Theobald and G. Weikum. The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. EDBT 2002, 477-495.
[27] TPC Benchmark H. Decision Support. http://www.tpc. org
[28] TREC: http://trec.nist.gov/.
[29] G. A. Watson. An Algorithm for the Single Facility Location Problem using the Jaccard Metric. SIAM J. Sci. Stat. Comput., 1983.
[30] D. Wilson and T. Martinez. Improved Heterogeneous Distance Functions. Journal of AI Research, 1997.
[31] L. Wimmers, L. M. Haas , M T. Roth and C. Braendli. Using Fagin's Algorithm for Merging Ranked Results in Multimedia Middleware. CoopIS 1999.
[32] L. Wu, C. Faloutsos, K. Sycara and T. Payne. FALCON: Feedback Adaptive Loop for Content-Based Retrieval. VLDB 2000.