

Ranked Queries: An Application

By: Amin Ghadersohi
Instructor: Dr. Jan Chomicki
State University of New York at Buffalo
Department of Computer Science and Engineering

* This report is based mainly on reference [1].

Ranking query results has many applications in the World Wide Web for searching structured and non-structured data for specific information. Areas such as Database Systems and Information Retrieval deal with the need to sort data based on specified preferences. The preferences can be user specified, or calculated per query, as needed, by a separate module, based on a number of factors dependant on the design of the system. Here we will propose an application for ranked queries to solve a problem more related to language processing than a database system problem.

String completion, suggestion and/or prediction, are areas of language processing that make great use of statistics and probability theory to achieve their goals. At the UB Tech group, one of our interests is to achieve good string completion using a relational database of strings, built up over time. Other models, such as probabilistic prefix trees, are also being researched and tested, but since the data is stored in a relational database, any other method will still have to resort to querying the database. Thus, even non-statistical based models will benefit from a "smarter" data storage/retrieval module that doesn't just store the data, but can also relate the to each other as to compare and rank the tuples in relation to other tuples as well as all the other relevant information that can be derived from the database.

The UB Talker

UB Talker is an I/O system that takes text strings, namely words, phrases and sentences as input, and outputs synthesized speech. Our primary user and motivation for the project, David, lost his ability to speak after suffering a stroke in his mid 20's. David also lost most of the motor control in his hands to the point where he can only use one finger to point to objects and type letters on a keyboard.

It should already be clear why we want to complete the sentences of such a user. For this purpose the database acts like the memory of the person the program is trying to assist. When a person says something, there are always properties that we can derive and store for use as statistics. Over time, we can use the statistics to relate the data to itself and other data. In other words, we can use the statistics to perform sophisticated and specific queries. We can take into consideration intrinsic properties of strings such as the number of times they were outputted to the synthesizer, as well as extrinsic features such as time of day, the emotional state of the user, etc. Since the nature of the Talker is such that it satisfies a primary need of a human being, namely communication, it is not hard to gather some useful information through the user's interaction with the program, over time. We want to also be able to add new types of information to the database "on the fly" and have the information play a role in ranking the results, without the need for modifying the query system.

Ranking

If the problem was to just retrieve data in any order, it would not make sense to rank the data. However, when we are trying to pinpoint the closest matching tuple for a query, in the case where exact matches don't exist, we need to have a way to compare

the results and sort them based on their similarity to the query. The query will contain constraints that the tuples need to satisfy.

Database systems support Boolean queries, where the condition is of form “where $A_1 = a$ AND $A_2 = b$ AND ...” Some of the problems that arise are 1) empty answers, 2) too many answers, and 3) inappropriate ordering of the answers [1]. The latter two cases combined is usually a problem with such systems. This is not to say that DB systems are flawed, just that they need to be extended or customized. It is indeed the case that most DB systems allow user extensions and customizations.

Ranking Functions

At the heart of ranking query results are ranking functions. Their job is to evaluate the rank or similarity of a tuple compared to another tuple. The general form of a ranking function can be simplified to a MAUT function of the form $U(I) = \sum_j W_j \cdot U_j(F_j(I))$, where $U_j(F_j(I))$ is the evaluation of attribute j of tuple I and W_j is the weight assigned to this attribute, i.e. the importance of this attribute in the total rank of the tuple compared to another tuple[2].

Today, most commercial DB systems, such as IBM's DB2 UDB, support the definition of user functions (UDF) [2]. The first and most important concern now is the problem of domain specific ranking needs. In other words, for every different type of data or attribute in the database, a new ranking function is required. An instance of such functions can be seen in Example 1.

```

import COM.ibm.db2.app.*;
class EvaluationUDF extends UDF
{
    public double PriceEval (double price, double val1, double val2)
    {
        /*
        * Get the slope and end point of the line segment that includes the
        * 2 points (val1, 1) and (val2, 0)
        * The line segment equation is y = a * x + b;
        */
        double a = 1 / (val1 - val2);
        double b = -val2 / (val1 - val2);

        /*
        * Return the evaluation for this price
        */

        double eval = a * price + b;
        return (eval);
    }
}

```

Function to generate the weight of each attribute:

```

CREATE FUNCTION weight (attribute-name char(30)) RETURNS integer
LANGUAGE SQL READS SQL DATA NO EXTERNAL ACTION DETERMINISTIC
RETURN
    SELECT weight
    FROM attribute_weight
    WHERE attribute weighth.attribute-name = weight.attribute-name

```

Select statement:

```

SELECT Product ID, Price, Speed, RAM, CDWriter, HDSIZE,
(
    weight("product.Price") * PriceEval (Price, 0,5600) +
    weight("product.Speed") * SpeedEval(Speed, 1600, 300) +
    weight("product.RAM") * RAMEval(RAM,512,32) +
    weight("product.CDWriter") * CDWriterEval(CDWriter) +
    weight("product.HDSIZE") * HDSIZEEval(HDSIZE)
) AS Score
FROM product
ORDER BY Score

```

Example 1: Evaluation function implementation as a JAVA UDF for the Price attribute

As we can see, there is still a problem of deciding the importance of each attribute in the final rank of the tuples. In some situations, such as searching an online product catalog, this may actually allow the user to set the weights manually. However, for a system like the Talker where the program is in charge of figuring out the user's

preferences, it would be better to have a way to automatically calculate the weights.

Moreover, both intrinsic and extrinsic properties of the data can be taken into consideration, and even combined with user defined preferences, in case where they are essential.

The following example demonstrates the process of formulating a ranking function for a specific attribute in the Talker database, namely frequency of speech, taking into consideration current time of day.

Strings Table:

sid	string
22	I want to chase ducks
23	Remember the time we went to the park to chase ducks
24	Lets have dinner
25	I need a shave

Time and frequency table.

sid	f(A)	f(B)	f(C)	f(D)	f(E)	f(F)	f(G)	f(H)
22	0	0	0	2	6	3	0	0
23	0	0	0	0	0	0	2	0
24	0	0	0	0	0	88	599	44
25	4	0	342	55	56	65	5	0

Note: *sid* refers to the primary key of the Strings table.
lasttime is a timestamp of the last time the string was used.
 A day is broken into eight timezones A-G, 3 hours long each. Eg: A = 12 – 3am
f(A), *f(B)*, ... , *f(G)* refer to the frequency of the usage of the string referred to by *sid* during the timezones A-G.

Define $f^*(x) = f(x)/[f(A)+f(B)+\dots+f(H)]$ where x is in $\{A-H\}$. Call this the normalized frequency of x . Define $w(x, y)$ as the weight of x in $\{A-H\}$ compared to the position of y in $\{A-H\}$ such that: $w(a,b) \leq w(b,a) < w(a,a) = w(b,b)$ if $a \neq b$ and the sum of the $w(x,x) + w(x,y) = 1.0$ for all y in $\{A-H\} - \{x\}$

Example 2: Ranking based on an attribute from the Talker database

We want to rank the tuples in the Strings table by considering the frequency of past usage of each tuple or string in the database. To accomplish this task, a day is broken into 8 time blocks, each 3 hours long, and whenever a string is spoken, the count for the proper time block is incremented. By doing this we have effectively categorized or bucketized a

numerical attribute. As later we will see, ranking of categorical data can be easier than numerical data [1].

We can formulate the queries of the Talker in the following way:

```
select text,  $\sum_{i \text{ in } \{A-G\}} w(i,T) f(i)$   
where text.contains(keyword1) AND text.contains(keyword2) and ...
```

The selection of keywords can be done as a separate task [1, 3, 4]. However, since we can't be sure that we are not omitting a relevant keyword, it would be nice if the ranking function could take into consideration the frequency of attributes.

As discussed before, the main problem here is repeating the same procedure every time a new data is added to the database. Therefore, we need to somehow generalize our methods. Typically, we can say that a database contains only numerical and categorical data. We can also take advantage of treating numerical data as categorical data to create a general framework for ranking database queries. Researchers at Microsoft have done just that [1]. Here we will take advantage of their work in an effort to create a better string completion module for the Talker. Moreover, it is easier to extend a general framework for specific needs, than to try and keep a collection of disparate frameworks and algorithms.

The Database Model

Assume a database R with attributes $[A_1, A_2, \dots, A_m]$ and tuples $[T_1, \dots, T_n]$. The conjunctive selection conditions of the queries will be of form "where C_1 AND C_2 ... AND C_m " where each C_k is of form $A_k = \text{value}_k$ or $A_k \text{ in } [\text{range}]$ [1].

Cosine Similarity

The method we are about to discuss is derived from a standard IR technique referred to as Cosine Similarity. Given a set of tuples in the database, we want to retrieve

the results in order of decreasing rank. Cosine Similarity defines the similarity or rank of a tuple compared to another tuple as the normalized dot product of the term frequency (TF) vectors of the two tuples [1]. In broader terms, given a vocabulary of m words, a document is treated as an m -dimensional vector, where the i_{th} component is the frequency of occurrence (item frequency, or TF) of the i_{th} vocabulary word in the document. Since a query is also a set of words, it too has a vector representation.

Cosine similarity can be further refined to handle more specific needs. For example, we can scale each component with the IDF (inverse document frequency) of the corresponding word or term where $IDF(w) = \log(N/F(w))$, $N = \#$ of tuples, and $F(w) = \#$ of tuples in which w appear [1].

IDF can be thought of as the weight multiplier for each attribute. It assigns smaller weights to commonly occurring terms in the database compared to rarely occurring ones. For example, we want to assign minimal weight to words such as ‘the’ ‘is’ ‘that’ ‘or’ and so on. Since these words are most frequent, they will count less in the total rank.

IDF Similarity

IDF similarity is a generic database ranking function that extends the Cosine similarity metric to DB systems containing a heterogeneous mix of categorical and numerical data. We will discuss how the two types are handled separately, and then we will try and relate them together.

IDF Similarity for categorical data

It is the case that many ranking techniques only take into consideration the intrinsic properties of tuples or documents. For example, only attribute values of tuples

are compared to other tuples. However, to ensure that the ranks of the tuples are distinct from others tuples, we need to somehow compare the tuple to the whole database, or an overall deduction of the database contents. As apparent from the name, IDF similarity uses the frequency of attribute values in the database to decide the importance of a factor.

For each value t in the domain of attribute A_k , define $IDF_k(t) = \log(n/F_k(t))$, where $n = \#$ of tuples in the database, and $F_k(t) =$ frequency of documents with attribute $A_k = t$. Then, for any pair of values u and v in A_k 's domain let $S_k(u,v) = (u = v ? IDF_k(u) : 0)$. S is the similarity of attribute value u compared to attribute value v , both in the same domain. This method can be extended such that n is not the count of all tuples in the database, but a selection of tuples with specific properties, i.e.: find all documents with certain keywords, by methods explained in [1, 3, 4], and then rank them based on the frequency of words, and possibly other attributes.

So far, we can compare values of the same attribute. We want to combine such results in the form of a summation to get the total similarity or rank of a tuple compared to another. As mentioned before, a query can also be thought of a tuple. For our purpose we can assume that the values of all attributes are specified in the query. Later, we will discuss how missing attributes can be used to break ties. For tuple $T = \langle t_1, t_2, \dots, t_m \rangle$ and query $Q = \langle q_1, q_2, \dots, q_m \rangle$, define $SIM(T, Q) = \sum_{k=1..m} S_k(t_k, q_k)$, the sum of corresponding similarity coefficients over all attributes, as the similarity, or rank of T compared to Q .

IDF Similarity for numerical data

As we saw with categorical data, we can use the frequency of tuples in the database to derive the weight or importance of an attributes values compared to the query. However, for numbers there is no such thing as frequency. The concept of

frequency only applies when values are treated like categories. For instance, when we specify a range of numerical values, we are effectively creating a group from numerical values, such that we can easily decide whether a value belongs to that range or not.

The task of finding such a range is not always an easy. Therefore, again we run into the problem of domain specificity. As we saw in our earlier example, a day was broken into 8 time zones. But this may or may not work for another database. To generalize, we want a Gaussian like distribution. The IDF must depend on nearby values, such that the similarity of values that are close to each other is high, i.e.: if we are completing a string at 10am, the exact time of day maybe less important than actually finding most of the keywords in the completion. In other words, we want to find timestamps near 10 am; the exact timestamp may not even exist in the database.

Let $\{t_1, \dots, t_n\}$ be the values of attribute A in the database. Define $IDF(t)$ as:

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

The denominator of the log acts like the concept of frequency. The farther t_i is from t , the smaller the contribution. Therefore, values that are closer to t will contribute more to the rank or similarity. Now we can redefine $S(t,q)$ as:

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

Where $h = 1.06 \sigma n^{-1/5}$ and σ is the standard deviation of $\{t_1, \dots, t_n\}$ [6].

We can use this approach to solve the domain specific ranking example from previous slides. Assume the timestamp is an integer value representing hour of day. If t is

equal to q , we get $S(t,q) = \text{IDF}(q)$. The denominator of the log in the IDF will act as a frequency by assigning highest contribution from the values that are the same or close to q . If t is far from q , $S(t,q) = [0 \leq s \leq 1] * \text{IDF}(q)$.

Values in a Range

So far we have considered queries such as: “where C_1 AND C_2 ... AND C_M ”, where each C is of form $A_k = \text{value}_k$. However, we may also want to also consider A_k IN Q_k , where Q_k is a range $[a, b]$. We can generalize the similarity between t_k and Q_k as the maximum similarity between t_k and all values in Q_k [1].

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

Database Workload

The queries that are submitted to the database (the workload) could also be of value when ranking query results. This maybe a poor choice since the final relevant tuples in not recorded. The UB Talker actually does record the final selected tuples by incrementing their frequency of usage. But we still don't know whether the order of tuples in the result set was correct or not. In other words, was the most relevant result the first tuple in the result set?

The workload can help find the frequency with which database attributes and values are referenced. It can be combined with IDF to provide better results. For example, we can keep track of past queries to the database using a profiling module, and use the information in ranking the current query results [1].

QF Similarity

Having defined workload, we can define an extension of IDF similarity that takes into consideration the frequency of query attribute values. Let $RQF(q) =$ raw frequency of

value q of attribute A in the query strings of the workload, and RQF_{Max} = raw frequency of the most frequent value in the workload. Then, define $QF(q) = RQF(q)/RQF_{Max}$ and $S(t,q) = (t == q ? QF(q) : 0)$.

Sometimes it may not be good to have $S(t,q) = 0$. It is more desirable to have a non-zero value. The similarity of two values in the database maybe zero according to just the data. We can make use of the *Jackarand Coefficient* [7]. Let $W(t)$ be the subset of queries in the workload W in which categorical value t occurs in an IN clause. The idea is that the values that appear in the same IN clause are more similar to each other than anything else.

$$J(W(t),W(q)) = |W(t) \cap W(q)| / |W(t) \cup W(q)|$$

Therefore, the similarity between t and q can be redefined as:

$$S(t,q) = J(W(t),W(q))QF(q)$$

If $W(t)$ is very similar to $W(q)$, $S(t,q) = QF(q)$. As we can see, this only takes into account the workload. We again combine this with IDF to get more useful results.

QFIDF Similarity

Redefine $QF(q) = (RQF(q)+1)/(RQF_{Max}+1)$, such that, if $RQF(q) = 0$ we still get a non-zero value. Now our similarity function for two tuples takes into consideration both data from the workload and the database: $S(t,q) = (t == q ? QF(q)*IDF(q) : 0)$. This maybe useful in the cases where the database does not contain enough information and workload does. The inverse of the situation can also be handled properly.

Breaking ties

Earlier, we assumed that the queries specify values for all the attributes. However, this is not a realistic assumption. Missing attributes can lead to clustering of the tuples in

the result set. In other words, groups of tuples will have the same rank and thus rendered incomparable with each other, if we were to ignore missing attributes. Consider a query that does not specify a value for every attribute. We can use the workload information to determine the importance of missing attribute t_k as $\log(QF_k(t_k))$, or in case of IDF, $\log(IDF_k(t_k))$. We can add the sum of these values, for all missing attributes, to the rank of the tuples in the results of IDF or QF similarity to break ties. Another alternative is to rank the tied tuples higher if their missing attribute values have small IDF or vice-versa, depending on the data [1]. However, the problem of domain specificity cannot be completely ignored, as each application may need to make adjustments.

Implementation Notes

To implement this method we can make use of dynamic programming techniques to prevent re-calculation of IDF, QF, etc values. We can pre-calculate, or estimate smooth functional representations of the data we need in auxiliary tables, loaded at initialization time. The data can also be approximated as histograms, in linear time, by the WARPing method [6].

In cases where Top-K tuple retrieval is required, we can use the Fagin's Threshold Algorithm derivative referred to as Index-Based Threshold Algorithm [8]. Further details on that are omitted from this paper.

Conclusions

For the purposes of the Talker, using this probabilistic model is appropriate because even most of the methods for pattern recognition and language processing are highly probabilistic and depend on large amounts of statistics and data. Today, such methodologies are used in handwriting recognitions, speech recognition, computer vision,

radar systems, data mining and information retrieval. Elements of dynamic programming can be used to expedite the process by pre-calculating values, or using a bottom-up approach to prevent exponential runtimes.

It would be a legitimate question to ask: why not use language definitions to accomplish the task of string completion? The answer would be that, even though we could use such a method to verify the results, we could not use them to make predictions without using any other data. Since we are essentially trying to predict the future, we need some information, and we can only get such information from the past and present. No matter what the algorithm, such a method is always going to be non-deterministic, in that there is no guarantee that a solution, or in our case a suggestion will exist in the current data.

Future Work

As we have seen, the biggest problem of ranking is domain specificity. To solve a problem it is always useful to be able to have a way to do quick experiments. A general framework for domain specific ranking needs to be defined so that we can see the problems in each case and incorporate solutions and fixes into our general ranking framework, such that it will automatically handle these cases also.

References & Bibliography:

- [1] S. Agawam, S. Chaudhuri, G. Das, A. Gionis. Automated Ranking of Database Query Results. CIDR 2003
- [2] M. Stolze & W. Rjaibi, Toward Scalable Scoring for Preference-based Item recommendation. IEEE 2001
- [3] S. Agrawal, S. Chaudhuri, G. Das. DBXplorer: A System for Keyword Based Search over Relational Databases. ICDE 2002.
- [4] K. Zechner. Automatic Generation of Concise Summaries of Spoken Dialogues in Unrestricted Domains. ACM 2001.
- [5] Vagelis Hristidis et al, PREFER: A System for the Efficient Execution of Multiparametric Ranked Queries. ACM 2001
- [6] B. W. Silverman. Density Estimation. Chapman and Hall. 1986
- [7] G. A. Watson. An Algorithm for the Single Facility Location Problem using the Jaccard Metric. SIAM J. Sci. Stat. Comput., 1983.
- [8] N. Bruno, L. Gravano, A. Marian. Evaluating Top-K Queries over Web-Accessible Databases. ICDE 2002