Ranked Queries: An Application

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

Motivation

- UB Tech Group
- We are working on an augmentative communications device to help those who are unable to speak and may have lost some of the motor control in their arms and hands, therefore can only type with one finger.

Motivation(cont..)

- Our primary user is David.
- David suffered a stroke in his mid 20’s. For 20 years since that incident, David has been unable to speak. His only choice, to point to letters of the alphabet on a sheet of paper attached to his wheelchair.

UB Talker

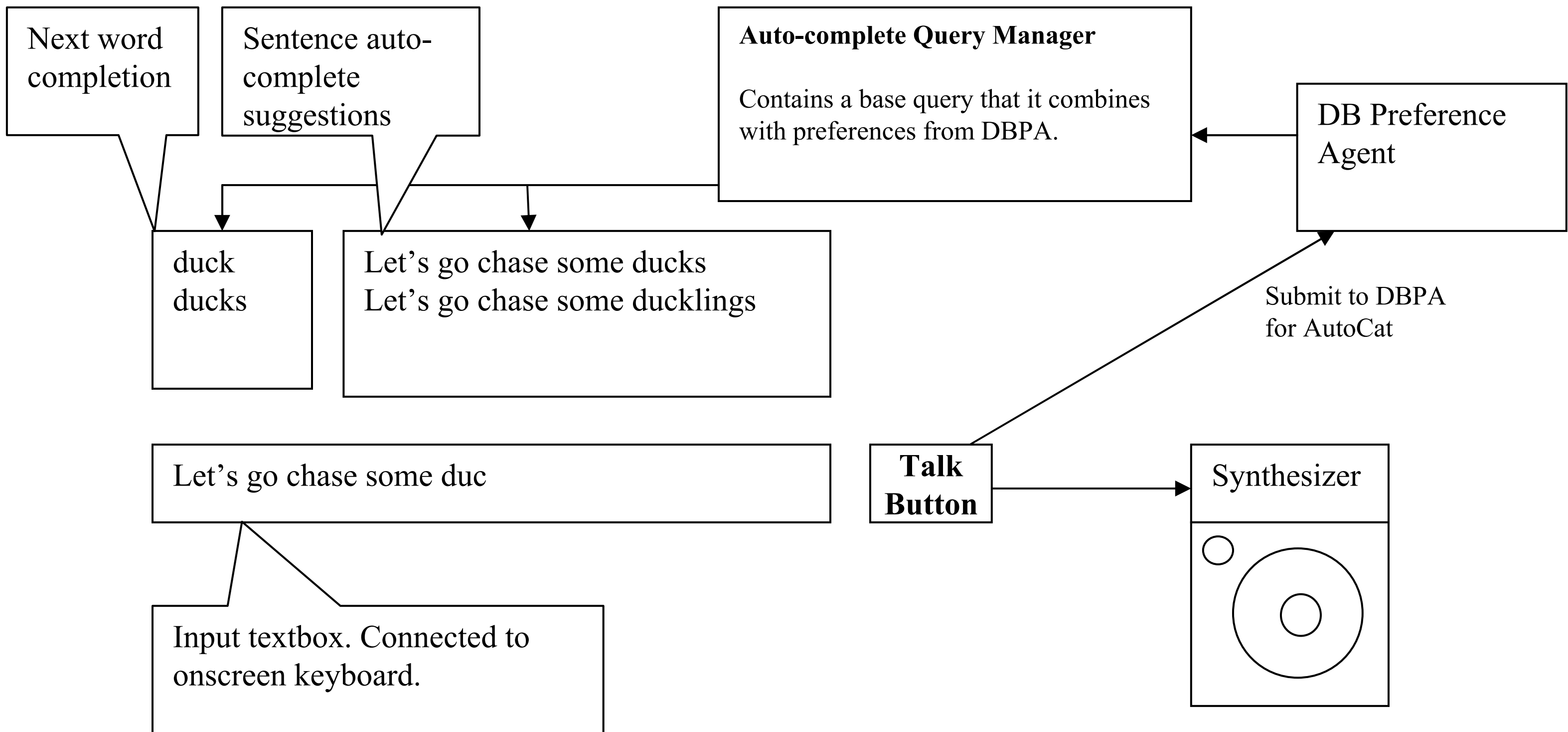
- The UB Talker is an I/O system for the input of text strings, namely words and phrases, and the output of synthesized speech.
- The engine is composed of a relational SQL database (mySQL), and an open source speech engine.

The Database

- The database keeps track of what has been outputted from the program, namely the phrases that have been spoken. These are stored in their own indexed table, where the primary key is an auto generated id and the string itself.
- We shall refer to the words and phrases as strings from now on.

The Problem

- The Talker is full user oriented software, in that it is something that the user depends on for ease of use and intelligence.
- Strings in the database have a number of attributes, namely statistics related to frequency of submission to the database, time and date of submission, as well as categories that either the user specified explicitly or the program generated automatically.


Next word completion
Sentence auto-complete suggestions
Auto-complete Query Manager
Contains a base query that it combines with preferences from DBPA.
DB Preference Agent
duck
ducks
Let's go chase some ducks
Let's go chase some ducklings
Submit to DBPA for AutoCat
Let's go chase some duc
Talk Button
Synthesizer
Input textbox. Connected to onscreen keyboard.

The Problem (cont..)

- We want to try and complete the string the user is trying to type or perhaps suggest similar strings.
- Each time we need to suggest, we have to query the database.
- When the database gets big, as it would, simply retrieving strings the complete the user's string or strings that contain some of the words is not enough.

The Problem (cont..)

- We want to automatically rank the strings in the database, using all the statistics that we store.
- In this sense the talker is a preference agent and has to figure out the user's preferences for each query. Figuring out the user's preferences is easy.

The Problem (cont..)

- The hard part is to match the user’s preferences to the information in the database, and return appropriately ordered results.
- Proposed solution: ranked queries.

Presentation outline

- Ranking functions
- A domain specific example
- IDF Similarity for categorical data
- IDF Similarity for numerical data
- QF Similarity
- Breaking ties
- Implementation

Ranking

- DB systems support Boolean queries

    i.e.: select * from …

        where A1 = a AND A2 = b …
- Problems:
    - Empty answers
    - Too many answers
    - Inappropriate ordering
- Solution: Ranking functions

Ranking Functions

- A ranking function is a value function that returns the rank of a tuple compared to another tuple.
- Ranking functions can help extend Boolean logic.
- Also referred to as utility functions.
- The general form can be simplified to MAUT function of the form: $U(I) = \sum_j W_j \,.\, U_j(F_j(I))$, 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.
- Most commercial database systems such as IBM's DB2 UDB allow for the definition of user defined functions (UDF).
- Problem of domain specific ranking functions.
  - OO ranking functions can help

Example: UDF for evaluating price

```
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);
        }
}
```

Evaluation function implementation as a JAVA UDF for the Price attribute

Example (cont..)

- **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 weigth.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
```

A Domain Specific Ranking example

Strings Table:

<u>sid</u>	<u>string</u>
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 (statistics).

<u>sid</u>	**lasttime**
22	343
23	600
24	33
25	433

<u>sid</u>	**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

- ***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 time zones A-H, 3 hours long each. E.g.: A = 12 – 3am
- f(A), f(B), … , f(H) refer to the frequency of the usage of the string referred to by sid during the time zones A-H.

A Domain Specific Ranking example

- Define $f'(x) = f(x)/[f(A)+f(B)+\ldots+f(H)]$ 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:

  $w(a,b) <= w(b,a) < w(a,a) = w(b,b)$ if $a != b$

  and the sum of the $w(x,x) + w(x,y) = 1.0$ for all y in {A-H}-{x}

A Domain Specific Ranking example

- select text, $\sum_{i \text{ in } \{A\text{-}G\}}$ 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. 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.
- To rank based on the lasttime timestamp we need to create another ranking function.

Need for generalization

- We can say that there are mainly numerical and categorical statistics that concern us.
- A general method for ranking can be refined in different situations to suit needs but it is hard to generalize disparate methods.
- Related work:
  - Automated Ranking of Database Query Results [1]
  - Toward Scalable Scoring for Preference-based Item recommendation.[2]

Model

- Assume a database with attributes $[A_1, A_2, \ldots, A_m]$ and tuples $[T_1, \ldots, T_n]$
- Queries of form:

  select … where C_1 AND C_2 … AND C_M

  where each C is of form $A_k = \text{value}_k$ or A_k in [range]

Cosine Similarity

- Given a set of tuples in the database and the query we want to retrieve the results in ranked order.
- The similarity of rank of a tuple compared to a query is defined as the normalized dot product of the term frequency vectors of the tuple and query. This is referred to Cosine Similarity of a tuple and a query*.

*Given a vocabulary of m words, a document is treated as an m-dimensional vector, where the ith component is the frequency of occurrence (item 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.

Cosine Similarity

- Can be further refined by scaling each component with the IDF (inverse document frequency) of corresponding word or term.

  where: $IDF(w) = \log(N/F(w))$,

  N = # of tuples

  F(w) = # of tuples in which w appear
- IDF assigns smaller weights to commonly occurring terms compared to rare ones.
- Eg: we want to assign a smaller importance to words like 'the' 'or' 'and'.

IDF Similarity

- IDF Similarity is a generic database ranking function that extends the Cosine Similarity metric from Information Retrieval to Database Systems containing a heterogeneous mix of categorical and numerical data.
- It can be combined with the workload of the database.

IDF Similarity for categorical data

- For each value t in the domain of attribute A_k, $IDF_k(t) = \log(n/F_k(t))$,

  where n = # of tuples in DB

  $F_k(t)$ = frequency of documents with attribute $A_k == t$
- For any pair of values u & v in A_k's domain let $S_k(u,v) = (u == v\ ?\ IDF_k(u) : 0)$
- Can be extended such that n is not the count of all tuples in the database, but a selection of tuples with specific properties.
- Eg: find all document with certain keywords and then rank them based on the frequency of words and other attributes.

IDF Similarity for categorical data

- For tuple $T = <t_1,t_2,\ldots,t_m>$ and query $Q = <q_1,q_2,\ldots,q_m>$,

  $SIM(T,Q) = \sum_{k=1..m} S_k(t_k,q_k)$ as the similarity, or rank of T compared to Q.

  Sum of corresponding similarity coefficients over all attributes.

IDF Similarity for numerical data

- Want a Gaussian like distribution
- The IDF must depend on nearby values.
- E.g.: 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.
- One solution is to categorize the numerical data into buckets.
- It may be hard to determine the number of buckets and can separate related data.

IDF Similarity for numerical data

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

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

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

- The denominator acts like the concept of frequency. The farther t_i from t the smaller the contribution.
- The denominator acts like the concept of frequency. The farther t_i from t the smaller the contribution.
- $h = 1.06\ \boldsymbol{\sigma}\ n^{-1/5}$ where $\boldsymbol{\sigma}$ is the standard deviation of $\{t_1,\ldots,t_n\}$.

IDF Similarity for numerical data: Example

- 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) = 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 <= s <= 1] . 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 = $value_k$
- 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

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

Missing attributes

- So far we assumed that Q contained a value for every attribute A.
- However, that is not always the case
- We only consider the projection of the database on the attributes that have been assigned a value in the query.
- We can use missing attributes to break ties.

Workload

- Can be helpful in ranking
- 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.
- Can help find the frequency with which database attributes and values are referenced.
- Can be combined with IDF to provide better results.
- E.g.: We can keep track of past queries to the database using a profiling module, and use the information in ranking the current query results.
- QF Similarity: Uses only workload information
- QFIDF Similarity: Uses both workload and data

QF Similarity

- Let RQF(q) = raw frequency of value q of attribute A in the query strings of the workload.
- RQFMax = raw frequency of the most frequent value in the workload.
- Define QF(q) = RQF(q)/RQFMax
- Thus S(t,q) = (t == q ? QF(q) : 0)

QF Similarity

- 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.
- Use of *Jackarand Coefficient* [1].
- 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.
- $J(W(t),W(q)) = |W(t) \cap W(q)| / |W(t) \cup W(q)|$
- Similarity between t and q: $S(t,q) = J(W(t),W(q))QF(q)$
- If $W(t)$ is very similar to $W(q)$, $S(t,q) = QF(q)$.
- QF Similarity is purely workload based.
- Workload may not have enough information
- A solution is to combine QF and IDF

QFIDF Similarity

- Redefine QF(q) = (RQF(q)+1)/(RQFMax+1)

  Therefore, if RQF(q) = 0 we still get a non-zero value.
- S(t,q) = (t == q ? QF(q)*IDF(q) : 0)
- Missing attributes can be used to break ties between results that have the same rank.

Breaking ties: too many results

- Sometimes many tuples will get the same rank, thus arbitrary ordered in the output
- Consider a query that does not specify a value for every attribute.
- Missing attributes can lead to result containing groups of tuples with the same rank.

Breaking ties

- We can use the workload information to determine the importance of missing attribute t_k as $\log(QF_k(t_k))$ and in case of IDF, $\log(IDF_k(t_k))$
- We can sum up these values for all missing attributes and use the result in 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.
- Sometimes a missing attribute with a low workload frequency can be of more importance than if it had high frequency. For instance rarely occurring attributes in the database. This could lead to incorrect ordering of results.

Implementation: Pre-processing

- Calculating QF, IDF, etc, for every time we rank a tuple can be very expensive.
- We can pre-calculate, or estimate smooth functional representations of the data we need in auxiliary tables.
- Data can be approximated as histograms in linear time by the WARPing method[4]

Implementation: Query Processing

- Top-K tuple retrieval
- For smaller databases, it may not be necessary to implement early terminating Top-K retrieval. It may be good enough to process everything.
- The advantage of early terminating Top-K is that not all tuples have to be processed.

Index-based Threshold Algorithm for Top-K (ITA)

- A Fagin's Threshold Algorithm derivative
- Works even when sorted access is not available for some attributes.
- Ranking or similarity functions seen so far satisfy the monotonic property.
  - If T & U are two tuples such that for all k $S_k(t_k,q_k) <= S_k(u_k,q_k)$, then $SIM(T,Q) <= SIM(U,Q)$
- This property allows for an early termination or stopping condition for Fagin's TA.

To implement ITA

- Need:
  a) Sorted access along any attribute A_k, in which TIDs of tuples can be retrieved in decreasing rank or similarity of the value their A_k attribute compared to q_k
  b) Random access to complete tuple content given a TID

ITA (cont..)

- Assume indices are present for columns $A_1,\ldots,A_p$ and not present on $A_{p+1},\ldots,A_m$.
- Want to do index seeks on orderings $L_1,\ldots,L_p$ where each L_k is ordered by the equality of $A_k=q_k$ preceding A_k != q_k.
- TableLookup(TID) – random access
  - Implement using relational database index facilities.
- IndexLookupGetNextTID(L_k) – sorted access
  - Acts like an iterator a seek pointer is kept for each L_k.
  - Can be interleaved or ordered in any way depending on the structure of the data

ITA: Early Stopping Condition

- Define tuple H as $<a_1,\ldots,a_p,q_{p+1},\ldots,p_m>$ where a_k is value of attribute A_k and A_k for each $L_1,\ldots,L_p$ is pointed to by the "current" seek pointer in the corresponding L_k. H is the best tuple we could see in the rest of unseen tuples.
- If SIM(H,Q) <= rank of lowest tuple in Top-K buffer, then terminate.
- Fewer indexed columns means more tuple lookups using TID.

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

Auxiliary table overhead

- If we use a ranking function that needs data like QF, IDF, etc, we are going to have one auxiliary table per attribute A_k. These tables are referred to as QFIDF tables.
- Each QFIDF table has two columns. <ColVal,QFIDFVal>, for each ColVal in the values of attribute A_k in the database.
- The cost of look up for these values, every time a tuple needs to be ranked, can be expensive.
- Since the QFIDF lookup is only dependant on the query and not the actual data tuples, we can read the data into a local data structure at initialization time.

ITA generalizations

- Can be extended to handle “nearby matches” for numerical columns
- Can be extended to handed WHERE clauses such as IN and range, as explained previously.

Future work

- General framework for domain specific ranking.
- General testing infrastructure for ranking.
- Keeping the balance of performance and accuracy of rankings is worth more research.

References

- [1] Sanjay Agrawal et. al, Automated Ranking of Database Query Results. CIDR 2003
- [2] Markus Stolze & Walid Rjaibi, Toward Scalable Scoring for Preference-based Item recommendation. IEEE 2001
- [3] Vagelis Hristidis et al, PREFER: A System for the Efficient Execution of Multiparametric Ranked Queries. ACM 2001
- [4] B. W. Silverman. Density Estimation. Chapman and Hall. 1986