IQPI: An Incremental System for Answering Imprecise Queries Using Approximate Dependencies and Concept Similarities

S. M. Fakhr Ahmad, M. H. Sadreddini, M. Zolghadri Jahromi

***Abstract*—Most of the proposed systems to process queries over web databases require the user to provide some information regarding the relative importance of attributes and the similarities between nominal values. Recently, a new system called AIMQ has been proposed, which is based on measuring concept similarities. This system is end-user independent and can answer imprecise queries. The main drawback of this system is that it is not incremental. All computations must be repeated when a tuple is added to the database. As a solution to this problem, in this article, we propose an incremental and efficient system called IQPI, which can be considered as the incremental version of AIMQ. In IQPI, the set of approximate dependencies between attributes are mined, first (using our new efficient approach). Using this set of dependencies, the user's imprecise query is converted into some precise queries. Each of the precise queries is then fed into the system and the results are filtered (to obtain most relevant answers) using concept similarity graphs. These graphs are constructed in another part of the system and each edge in a graph represents the similarity between two nominal values. The structure of the similarity graphs are such that the least amount of computation is needed for them to be updated, when database is changed. In dependency mining part of the system, we present a new incremental algorithm that is based on logical operations over bit strings. It is crucial for a search system to be incremental, due to the dynamic nature of the world-wide web.**

***Index Terms*—Imprecise Query, Relational Database, Concept Similarity, Approximate Dependency, Incremental**

I. INTRODUCTION

By the fast expansion of the World Wide Web, a large number of web databases have been accessible to users from all over the world. The user submits a query containing a few constraints binding to different fields of the database and receives a set of tuples as the result of search process (which seem to be relevant to the query). Most of the Database query processing models assume users know what they want and how to formulate the query. As a matter of fact, the user is usually unable to express his need precisely. However, he can often tell which tuples are interesting when receiving a mixed set of results with different degrees of relevance to the query.

As an example, suppose a user searching for his interesting car through a car database. Assume that he wants a car costing about 10000$. If he knows a special case within this range of price (e.g., Toyota), a feasible query might be as follows: Q: (Make = 'Toyota' , Price <= 10000). Submitting this query to the database, a set of tuples including different models of *Toyota* and having prices not over 10000$ will be returned to him, i.e., the user only receives the answers which precisely satisfy the query conditions. However, he may also be interested in cars which are similar to *Toyota*. Moreover, a model of *Toyota* may exist which costs 10010$ (a bit over 10000$), having many advantages in comparison with the others, but is not shown to the user. In such cases, the need to a search system which can process imprecise queries (such as: Q: (Make *Like* 'Toyota' , Price *around* 10000)) is sensed.

Problem Statement: Given a conjunctive query Q over a relation R, find all tuples of R that satisfy Q above a threshold of relevance, *thresh*.

$$\text{Ans}(Q) = \{x \mid x \in R, \text{Sim}(Q, x) > \textit{thresh}\} \quad (1)$$

, where *thresh* is a real-value number in the unit interval [0, 1].

The rest of paper is organized as follows. Section 2 introduces some existing approaches for answering imprecise queries. In section 3, our proposed system is described and the main algorithms (similarity mining and dependency mining algorithms) are presented. Experimental results (containing two different types of experiments) on some benchmark data sets are shown in section 4. Finally, we give a conclusion at the end of the paper.

II. RELATED WORK

Many of the proposed systems to process queries over web databases require the user to provide some information regarding the relative importance of attributes and the similarities between nominal values. For example, in [1], the authors propose a method to provide ranked answers to queries over Web databases, but some additional guidance must be provided by the users in deciding the similarity. These approaches however are not applicable to existing databases as they require large amounts of domain specific information either pre-estimated or given by the user of the query. Many other approaches for retrieving answers to

S. M. Fakhr Ahmad is with the Department of Computer Engineering, School of Engineering, Islamic Azad University of Shiraz (and PhD student in Shiraz University), Shiraz, Iran
(e-mail: mfakhahmad@cse.shirazu.ac.ir)
M. H. Sadreddini is with the Department of Computer Science & Engineering, School of Engineering, Shiraz University, Shiraz, Iran (e-mail: sadredin@shirazu.ac.ir)
M. Zolghadri Jahromi is with the Department of Computer Science & Engineering, School of Engineering, Shiraz University, Shiraz, Iran (e-mail: zjahromi@shirazu.ac.ir)

imprecise queries are based on theory of fuzzy sets. Fuzzy systems [2] contain attributes with imprecise values, like height= “tall” and color=“blue or red”, and allow the retrieval with fuzzy query languages. In [3], Motro has added a *similar-to* operator using distance metrics over attribute values in order to interpret vague queries. These metrics must be provided by database designers. As the main challenge, for each problem, the optimal number of fuzzy sets, their type and their parameters must be specified in order to obtain the best results. Binderberger [4] investigates methods to extend database systems to support similarity search and query refinement over arbitrary abstract data types. Further [4] requires changing the data models and operators of the underlying database while [1] requires the database to be represented as a graph.

In [5, 6], authors explore methods to generate new queries related to the user’s original query by generalizing and refining the user queries. The abstraction and refinement rely on the database having explicit hierarchies of the relations and terms in the domain. In [7], Motro proposes allowing the user to select directions of relaxation, thereby indicating which answers may be of interest to the user.

Recently, a new system called AIMQ [8] has been proposed by Nambiar et al., which is based on measuring concept similarities. This system is end-user independent and can answer imprecise queries. The system assumes that tuples in the base set are all relevant to the imprecise query and creates new queries. The technique they use is similar to the pseudo-relevance feedback [9, 10] technique used in IR systems. The main drawback of this system is that it is not incremental. As a solution to this problem, in this article, we propose an incremental and efficient system called IQPI, which can be considered as the incremental version of AIMQ. The structures used in the proposed system are such that the least amount of computation is needed for them to be updated, when database is changed. In dependency mining part of the system, we present a new incremental algorithm that is based on logical operations over bit strings. Using these two incremental methods in main sections of the system makes the whole process be much more efficient than previous systems.

III. The proposed system (IQPI)

In this section, we propose an incremental and efficient system called IQPI, which can be considered as the incremental version of AIMQ. The system consists of three main parts. In the first part, the set of approximate dependencies between attributes are mined. In this part, we present a new incremental algorithm that is based on logical operations over bit strings. The second part of the system is the similarity miner, whose output is a set of similarity graphs. Each edge in a typical graph represents the similarity degree between two nominal values. The structure of the similarity graphs are such that the least amount of computation is needed for them to be updated, when database is changed. The third and central part of the system is the search engine which uses the results of the other two parts. In this part, using the set of dependencies, the user's imprecise query is converted into some precise queries. Each of the precise queries is then fed into the system and the results are filtered (to obtain most relevant answers) using concept similarity graphs. Finally, a set of tuples, having the relevance degree above a threshold, are returned to the user.

A. *Measuring Concept Similarities*

As mentioned before, in each step of expanding the set of answers, the tuples must be filtered according to their degree of relevance to the query. Thus we must have a factor to measure the similarity between a query and a tuple. Measuring the similarity between two vectors containing just numeric values is straight forward, using different factors such as Euclidian distance, etc. However, how can we measure the similarities between nominal attributes (i.e., *concept similarity*)?! Before illustrating the method, we present a definition for *concept*.

concept: An attribute coupled with an assigned value is called a concept, e.g., Make= 'Toyota' is a concept over the database CarDB.

Now, we describe the process of measuring the concept similarity between *Make = 'Ford'* and *Make = 'Toyota'* as an example:

1- Each concept is considered as a query and submitted to the database, separately. The result of running each query is a set of tuples which is called a *supertuple*.

2- Each supertuple is represented in a table which shows the number of each existing concept (in the supertuple). For example, consider the supertuple shown in Fig. 1, which is the result of the query *Q: Make = Toyota* over the database CarDB.

ST(Q:Make="Toyota")	
Model	Camry: 7, Corolla: 6
Year	2000:6, 1999:5 , 2001:2
Price	5995:4, 6500:3, 4000:6

Fig. 1. The supertuple obtained from running the query Make = " Toyota" over the database CarDB

The values within the supertuple of Fig. 1 indicate that There are totally 13 records in the database having Make = 'Toyota'. The first row in this figure shows that from these 13 records, in 7 cases the Model is 'Camry' and in the 6 others it is 'Corolla'. Similar information can be gain from other rows. The supertuple can be considered as a collection of 39 non-identical concepts. Similarly, we have such a structure for the concept Make = 'Ford'.

3- The union and the intersection of the two supertuples (considering the repeating items) are measured and fed into the *Jaccard* similarity formula (shown in equation (3)) to measure the similarity of the two concepts.

$$\mathrm{Sim}(A,B) = (S_A \cap S_B)/(S_A \cup S_B) \tag{2}$$

, where S_X is the supertuple of the concept X. After computing the concept similarities for each pair of concepts (related to each attribute), we construct a similarity graph for each attribute. Each node in this graph represents a nominal value, which is contained in the domain of the attribute. Each edge of the graph connecting two concepts is labeled with two values which are the union and the intersection of two supertuples related to the two concepts.

Fig. 2 is an example of similarity graphs for the concept Make = 'Ford'. The values of intersection and union are shown by *I* and *U*, respectively.

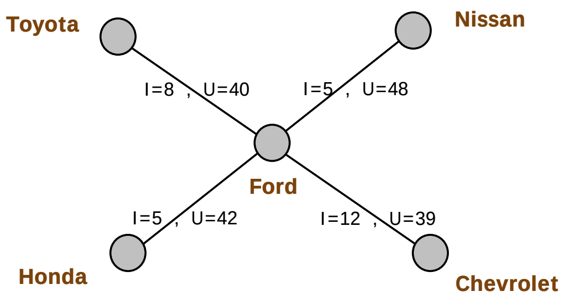

Fig. 2. Similarity graph for the concept Make = 'Ford'

As an example, suppose that we want to measure the similarity between two concepts Make = 'Ford' and Make = 'Toyota'. First, each concept should be submitted to the database as a query. Assume that the following supertuples are the results of the queries:

Q1 :− CarDB(Make = “Toyota”) $\longrightarrow$ S_A:

Model	Camry:3 , Corola:4
Price	10k-15k:4 , 15k-20k: 3
Color	Blue:1, Black:3 , White:
Year	2005 : 2 , 2006:3 ,

Q2 :− CarDB(Make = “Ford”) $\longrightarrow$ S_B:

Model	Focus:2 , F150:3
Price	10k-15k:3 , 1k-5k: 2
Color	Blue:2, Red:2 , White: 1
Year	2005 : 1 , 2006:4

Consequently, the similarity of these concepts is computed as follows:

$S_A \cap S_B = 8$, $S_A \cup S_B = 40$
$\Rightarrow$ Sim(Toyota , Ford) = 8/40 = 0.2

A1) Updating Concept Similarities

According to the dynamic nature of the world-wide web, the similarity mining algorithm must be designed such that when some change occurs in data, it can update similarity values without any need to re-scan all data and restart all computations. Such algorithms are called incremental algorithms.

In the proposed system, while constructing the graphs, we save some information about the co-occurrences[1] of different concepts in a table called co-occurrence table. This table is a symmetric matrix that indicates the co-occurrence times of each pair of concepts (within a tuple). A part of this table is shown in Fig. 3. For example, the value 5 shown in the first row and the first column of this figure implies that there are 5 cars having make = 'Ford' and Model = 'Focus'.

[1] Two concepts are called to have co-occurrence if they can be found in a tuple.

Suppose that the values for similarity elements (*I* and *U*) of these two concepts have been 8 and 40, respectively. Now, consider that a new tuple such as (Ford, Focus, 12k, black, 2006) is added to the database. This new record will make some affects on the previous results. However, it should not lead to running all similarity computations. A key point is that in similarity graphs, only the values related to the concepts which are present in the new record (e.g., Make = 'Ford'), have to be updated. To compute the similarity of Make = 'Ford' with other concepts such as Make = 'Toyota' the following steps must be performed:

1- Increment the co-occurrence values of the main concept (here, Make = 'Ford') and every other concept existing in the new tuple, by 1 (The first row in Fig. 3).
2- Count the number of values which have been incremented and still have a value not more than the similar value of the other concept (here, Make = 'Toyota'), denote it by k.
3- New values for *I* and *U* are computed from the following formulas:

$$I = I + k \qquad (3)$$

$$U = U + (|R|-1) - k \qquad (4)$$

, where $|R|$ is the number of database attributes.

Fig. 4 shows the new co-occurrence values for the two mentioned concepts after insertion of the new tuple. Considering the changed values in the first row and comparing them with the related values in the other row, the value of *k* is found to be 1. Using the equations (3) and (4), the similarity elements (*I* and *U*) are computed as follows:

$I = 8 + 1 = 9$
$U = 40 + (5 - 1) - 1 = 43$
new similarity value = 9/43 = 0.21

B. Mining Approximate Dependencies

Functional dependencies (FDs) are defined as relationships between attributes of a relational scheme R, and are presented in expressions of the form $X \rightarrow A$. In this expression *X* (referred to as the Left-Hand Side (LHS) of the dependency) is a subset of attributes belonging to R and *A* (referred to as the Right-Hand Side (RHS) of the dependency) is an attribute of R. A functional dependency is said to be valid in a given relation r over R , if for all pairs of tuples *t*, *u* belonging to r, we have

$$(t[X_i] = u[X_i]\ ,\ \text{for all } X_i \text{ in } X) \Rightarrow t[A] = u[A] \qquad (5)$$

Classical Functional dependencies are used in relational schema design in order to normalize relations to be free of redundancy and update anomalies. These dependencies don't allow for exceptions and are sensitive to noisy data. Approximate Dependencies (ADs) are dependencies which do not hold over a fraction of data and thus have a higher flexibility for exceptions and noisy data [11].

	Model= 'Focus'	Model= 'z13'	Model= 'Camry'	Price= '10-15k'	Price= '15-20k'	Color= 'blue'	Color= 'black'	Year= 2004	Year= 2005	Year= 2006
Make= 'Ford'	5	3	2	1	4	6	3	4	2	7
Make= 'Toyota'	3	4	6	2	5	4	3	5	3	1

Fig. 3. A part of co-occurrence table containing two concepts, Make = 'Ford' and Make = 'Toyota'

	Model= 'Focus'	Model= 'z13'	Model= 'Camry'	Price= '10-15k'	Price= '15-20k'	Color= 'blue'	Color= 'black'	Year= 2004	Year= 2005	Year= 2006
Make= 'Ford'	**6**	3	2	**2**	4	6	**4**	4	2	**8**
Make= 'Toyota'	3	4	6	2	5	4	3	5	3	1

Fig. 4. A part of co-occurrence table containing two concepts, Make = 'Ford' and Make = 'Toyota' after insertion of a new tuple

In dependency mining part of the system, we present a new incremental algorithm that is based on logical operations. It uses logical operations on binary strings to find the set of minimal dependencies (having an acceptable accuracy) between attributes. Many of the dependency mining approaches already proposed are not incremental and so have to re-scan all data and repeat the whole computations when a number of records are added to the database [12]–[16]. In this section, we first present some definitions, and then the incremental method for discovery of ADs is described.

B1) Definitions

Definition 1. Membership Binary String of attribute *A*, for the discrete value λ (denoted as: MBS(A,λ)) in a relation *r* over a relational scheme R, is a binary string having a length equal to the number of tuples of r. Each bit in this string is associated with a tuple of the relation and is set to 1 if attribute *A* has the value of λ and 0, otherwise.

Example 1. Consider a relational scheme R having four attributes *A*, B, *C* and *D*. A relation instance r over R is given in Fig. 5. Using definition 1, the bit string for MBS(C,'*H*'), MBS(*C*,'*L*') and MBS(*C*,'*M*') can be calculated as:
MBS(C,'*H*')= 001000001000
MBS(C,'*L*') = 000110110110
MBS(C,'*M*') = 110001000001

A	*B*	*C*	*D*
H	*L*	*M*	*L*
H	*L*	*M*	*L*
H	*L*	*H*	*H*
H	*H*	*L*	*L*
H	*H*	*L*	*L*
H	*M*	*M*	*L*
M	*H*	*L*	*M*
M	*L*	*L*	*M*
M	*L*	*H*	*L*
M	*L*	*L*	*M*
M	*L*	*L*	*M*
H	*H*	*M*	*M*

Fig. 5. A discrete-valued relation instance

Definition 2. Membership Binary Set of attribute *A* (denoted as MB-Set(*A*)) is a set having all MBSs of attribute *A* as its members.

Example 2. Using the above definition, the MB-Set(*C*) for the relation r shown in Fig. 5 is calculated as:
MB-Set(*C*) = {MBS(*C*,'*H*') , MBS(C,'*M*') , MBS(C,'*L*')}
= {001000001000 , 110001000001 , 000110110110}

C. Checking the Validity of an AD

For a relation instance, r, over the relational scheme R, we first compute the MBS of each attribute in R according to each discrete value it can take, and then construct the MB-Sets for all attributes. The complexity of this operation is O(|R|.|r|). All the information required to check the validity of a particular dependency is now available in MB-Sets and even database updates do not require rescanning of the old data.

In order to measure the validity degree of a particular dependency, we first construct the set *M* in the following way. Any member of *M* is constructed by performing logical *AND* operation on the bit-strings, each selected from an MB-Set of the LHS attributes. For example, if the LHS of a dependency has 3 attributes, each having 4 MBSs, then the *M* set will have 4^3 members. The result set, *F*, is then calculated using some logical operations given in the algorithm of Fig. 6. This set can then be used to calculate the accuracy of the dependency under investigation. For this purpose, we perform logical *OR* operation over all members of *F*. The number of 1-bits in the resulting string, *S*, which stands for the number of exceptions (denoted by *e*) is counted and inserted in the equation (6) to calculate the accuracy of the dependency.

$$Accuracy = \frac{|\mathrm{r}| - e}{|\mathrm{r}|} \tag{6}$$

Example 3. As an example, consider $AB \rightarrow C$ over the relation r (an instance of R(*A*,*B*,*C*,*D*)), shown in Fig. 5. By Using the four steps of the algorithm presented in Fig. 6, the accuracy of this dependency can be calculated as follows:

MB-Set(*A*) = {(111111000001), (000000111110), (0000000000000)}
MB-Set(*B*) = {(000110100001), (000001000000), (111000011110)}
MB-Set(C) = {(001000001000), (110001000001), (000110110110)}

M = {(000110000001), (000001000000), (111000000000), (000000100000), (00000000000), (0000000011110), (00000000000), (00000000000), (00000000000)}
F = {(000000000001), (000000000000), (001000000000), (000000000000), (000000001000)}
S = 001000001001

```
Algorithm: Measure-Accuracy
Input: A set of attributes as the LHS and a single attribute as the RHS of a dependency
Output: Accuracy of the dependency and indices of all tuples not satisfying the dependency
1. Construct a set by performing logical AND operation over all members of MB-Sets of the LHS. Denote the resulting set as M.
   From M, Omit those strings which do not contain any 1.
2. Construct the empty set F
   For each member m of M
     For each member $c_j$ of MB-Set of the RHS
        Find $f_j = m\ AND\ (NOT(c_j))$
        Store $f_j$ as temp if it has the minimum number of 1-bits already found for the $c_j$.
     End For
     Add temp to F.
   End For
3. Perform logical OR operation over all members of F, Denote the resulting string as S.
4. Count the number of 1-bits in S and insert it (as e) into the equation (6) to obtain the accuracy.
```

Fig. 6. Measure-Accuracy: The algorithm for computing the accuracy of a dependency

In the above example, the M set has been constructed using *AND* operation over MB-Set(A) and MB-Set(B), (i.e., MB-Sets of the LHS attributes). Since both MB-Sets have 3 members, the M set contains $3^2 = 9$ members. However, 4 of these members are zero-strings which do not have any effect on the final result and can be omitted from M to avoid unnecessary computations. Here, only the remaining members of M (i.e., 5 non-zero strings) have been used to construct the F set, and that's why the F set contains 5 members instead of 9.

In this example, S contains three 1-bits. These bits occur at indexes 3, 9 and 12 of the relation instance. One key feature of the method is that these indexes show the positions of those tuples that reduce the accuracy of the dependency (i.e., exceptions). Using equation (6), the accuracy of this dependency is calculated as:
(12-3)/12 = 75 %.

D. Search Engine

The process of answering an imprecise query involves the following steps:

1. Converting the query to a precise query

As the first step, the imprecise query is converted to a precise query. This task is accomplished by converting each '*like*' operator to '=' within the query statement. The resulting query is called Q_{pr}.

2. Running the Precise Query

The precise query is run and all tuples that exactly satisfy the query constraints, are retrieved. The result set of records is called the base set or A_{bs}.

3. Extending the Base Set

Using the base record set, A_{bs}, some other records which are similar to them, are retrieved and it leads to a more extended set of tuples, called A_{es}. The process is performed as follows:

Each record within A_{bs} is assumed as a query statement having $|R|$ conditions, where $|R|$ is the number of database attributes. Each constraint is removed from the query (this is called query relaxation) in turn and each time a number of tuples are retrieved. In each iteration, the retrieved tuples are filtered according to their similarity with the query and those having a similarity above a threshold are accepted. A challenge in this step is the order according to which the attributes are removed from the query. This order is determined according to the importance degrees of attributes and the least important attribute is the first one to be removed. If an attribute highly influences other attributes then it should be removed last.

D1) Measuring Weights of Attributes

We are now faced with the issue of which attribute to relax first. We make use of the ADs to decide the relaxation order within the set of attributes. For each attribute we determine a weight depending on how strongly it influences the other attributes. The importance of an attribute such as X is measured as follows: The accuracy degrees of all ADs which contain X on their LHSs are summated. If the LHS of an AD is the single attribute X, then the accuracy of the AD is counted, straightly. However, if the LHS is a set of attributes containing X, the accuracy of AD is divided into the length of the LHS and then summated. This issue can be more clarified through the following example.

Example 4. Consider a relation R(A,B,C,D,E). The set of ADs that hold over this relation, are given.

$$A \rightarrow B:\ 0.7,\quad A \rightarrow C:\ 0.85,\quad AD \rightarrow E:\ 0.92,$$
$$C \rightarrow A:\ 0.87,\quad BD \rightarrow C:\ 0.98,\quad BC \rightarrow E:\ 0.9$$

The weights (importance degrees) of attributes are computed as follows:

Weight(A) = 0.7 + 0.85 + (0.92/2) = 2.01
Weight(B) = (0.98/2) + (0.9/2) = 0.94
Weight(C) = 0.87 + (0.9/2) = 1.32
Weight(D) = (0.98/2) + (0.92/2) = 0.95

Since The attribute E can not be seen on the LHS of any AD, so it does not influence the other attributes at all and has an importance degree = 0. Thus, E is the first attribute that is selected to be removed from the query. After E, the attributes A, C, D, B are selected, respectively.

IV. Experimental results

A. Evaluating the Efficiency of Similarity Mining

As mentioned before, the main advantage of the proposed system in comparison with AIMQ is that it is incremental. In order to observe the time for updating the similarity graphs, when a new tuple is added to the database, the two systems (IQPI and AIMQ) were evaluated via a set of experiments over some real-life datasets from UCI repository. In each experiment, the similarity graphs were first constructed using 90% of data. Then, using the remaining data, the similarity values were updated. The times to compute the new similarity values are shown in table I.

Since AIMQ is not incremental, when a new tuple is added to the database, the whole process of constructing similarity graphs has to be repeated. However, IQPI just updates a number of values through the method described in section 3.1.1. The difference is sensible in the results of this experiment.

Table I. The time to update similarity values when a set of tuples are added to the database

Time (sec)					
Flare	Letter recognition	Adult	Nursery	CarDB	
6	34	42	27	24	IQPI
31	356	578	190	167	AIMQ

B. Evaluating the Relevance of the Results

The well-known factor used in IR systems to evaluate the relevance degree of query answers, is the Work/ReleventTuple formula, shown in equation (7).

$$\text{Work/ReleventTuple} = \frac{\text{number of all retrieved tuples}}{\text{number of relevant tuples retrieved}} \qquad (7)$$

This factor represents the average number of tuples a user must verify from the result set until he reaches his favorite answer. Our first experiment was run in order to compare the rank-based selection of attributes for query relaxation (used in this work) with the random selection approach. In this experiment, the similarity threshold was changed from 0.3 to 0.9. In each case, the two approaches were executed for 10 different queries over the CarDB database and the average value (through 10 queries) of *Work/ReleventTuple* was computed. The results show that the above factor is higher in case of random-selection. It can also be seen that the random-selection approach is very sensitive to the value of the similarity threshold, i.e., the number of tuples that are recognized as relevant and returned to the user increases more significantly as the value of the threshold decreases. This problem is not seen in rank-based selection.

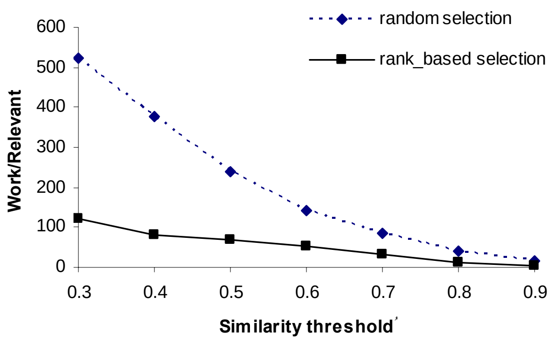

Fig. 7. comparing the rank-based selection of attributes for query relaxation with the random selection approach

V. Conclusion

In this article, we proposed an incremental and efficient system called IQPI, which is similar to another proposed system, AIMQ. The main drawback of AIMQ is that it is not incremental and all computations must be repeated when a tuple is added to the database. The proposed system in this article can be considered as the incremental version of AIMQ. In IQPI, the set of approximate dependencies between attributes are mined, first. Using this set of dependencies, the user's imprecise query is converted into some precise queries. Each of the precise queries is then fed into the system and the results are filtered (to obtain most relevant answers) using concept similarity graphs. These graphs are constructed in another part of the system and each edge in a graph represents the similarity between two nominal values. The structure of the similarity graphs are such that the least amount of computation is needed for them to be updated, when database is changed. In dependency mining part of the system, we present a new incremental algorithm that is based on logical operations over bit strings. In order to evaluate the relative efficiency of the system, a set of experiments over some real-life data sets, were conducted and we compared the time for updating the similarity graphs by IQPI and AIMQ, when a set of tuples are added to the database. The results show an acceptable relative efficiency for the proposed system.

Appendix: Illustrating Proofs for Equations (4) and (5)

a) Equation (4): Updating formula of the parameter I after inserting a new tuple to the database ($I = I + k$)

Consider having two collections A and B, each having a number of items (we denote them by *pseudo-sets*[1]). Let the intersection of the two pseudo-sets (considering repeating members, as well) be I. Assume that there exist m instances of a typical element x within the pseudo-set A, whereas the existing count of this element in B is equal to n (m and n are two positive integers). Depending on the relative values of m and n, two cases can occur. If $m < n$, adding a new instance of x to A (incrementing m) causes the count of x common between A and B, and consequently the whole intersection of A and B increase ($I = I + 1$). On the other hand, if $m \geq n$, the count of x instances that are common between A and B equals n. In this case, adding any number of x to A does not change this value from n, and thus, does not affect on the intersection of the pseudo-sets (I).
In general, when we add one new instance to a number of items of the pseudo-set A, some of them may not have any effect on the intersection value (of A and B). Among all increasing items, just items whose existing counts in A are less than their existing count in B take part in changing the value of I. The number of elements satisfying this constraint is denoted by k in the article. Thus, $I = I + k$.

[1] They have a close meaning to *sets*, also being allowed to contain members of repeating values

b) Equation (5): Updating formula of the parameter *U* after inserting a new tuple to the database ($U = U + (|R| - 1) - k$)

Consider a typical database R, and let $|R|$ be the number of attributes in R. As mentioned through the article, whenever a new record is added to the database (R), each row of the co-occurrence table, which is present within the new record, has to be updated. Since each record includes $|R|$ concepts, the number of rows in the co-occurrence table to be updated is equal to $|R|$. In this table, the co-occurrence values of each concept (each of $|R|$ concepts) with every other concepts existing in the new record (remaining $|R|$ -1 concepts) must be incremented (by 1). Considering each row of the co-occurrence table as a pseudo-set (containing concepts), the union count (U) of two pseudo-sets is increased by $|R|$-1, after updating their associating rows.

On the other hand, we know that the union count of two sets (or pseudo-sets), A and B, can be straightly found through the following formula:

$U = |A| + |B| - I$,

where |A|, |B| and I are the number of items in A, the number of items in B and the intersection count of A and B.

Adding a new record to R, adds a value equal to $|R|$-1 to $|A|+|B|$, and also the value of k to I (k was introduced in part *a* of the appendix). Let use U_0 and I_0 for the union and intersection counts of A and B before adding the new record, and denote these two values after the record insertion by U and I, respectively. Considering the above reasoning, we have:

$$U_0 = |A| + |B| - I_0$$
$$U = |A| + |B| + (|R| - 1) - I$$
$$I = I_0 + k$$

$$\Rightarrow \quad U - U_0 = (|R| - 1) - (I - I_0)$$

$$\Rightarrow \quad U = U_0 + (|R| - 1) - k$$

References

[1] R. Goldman, N .Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. *VLDB*, 1998.

[2] J.M. Morrissey. (1990, April). Imprecise information and uncertainty in information systems. *ACM Transactions on Information Systems*. 8. pp. 159–180.

[3] A. Motro. Vague. (1998). A user interface to relational databases that permits vague queries. *ACM Transactions on Office Information Systems*, 6(3). pp. 187–214.

[4] M. Ortega-Binderberger. Integrating Similarity Based Retrieval and Query Refinement in Databases. *PhD thesis*, UIUC, 2003.

[5] W.W. Chu, Q. Chen, and R. Lee. (1991). Cooperative query answering via type abstraction hierarchy. *Cooperative Knowledge Based Systems*. pp. 271–290.

[6] W.W. Chu, Q. Chen, and R. Lee. A structured approach for cooperative query answering. *IEEE TKDE*, 1992.

[7] A. Motro. Flex: A tolerant and cooperative user interface to database. *IEEE TKDE*, 1990, pp. 231–245.

[8] U. Nambiar, and S. Kambhampati, Answering Imprecise Queries over Autonomous Web Databases, *In: Proc. ICDE 2006, 22nd International Conference on Data Engineering*, 2006.

[9] C. Buckley, G. Salton, and J. Allan. Automatic Retrieval with Locality Information Using Smart. TREC-1, *National Institute of Standards and Technology*, Gaithersburg, MD, 1992.

[10] N.E. Efthimiadis. Query Expansion. *In Annual Review of Information Systems and Technology*, Vol. 31, 1996, pp. 121–187.

[11] P. Bosc, L. Lietard, and O. Pivert, Functional dependencies revisited under graduality and imprecision, *NAFIPS*, 1997, pp. 57–62.

[12] P. A. Flach and I. Savnik. (1999). Database dependency discovery: a machine learning approach, *AI communications*, 12(3). pp. 139–160.

[13] Y. Huhtala, J. Kärkkäinen, P. Porkka and H. Toivonen. (1999) TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *The Computer Journal*. 42(2). pp. 100–111.

[14] Y. Huhtala , J. Kärkkäinen , P. Porkka and H. Toivonen, Efficient Discovery of Functional and Approximate Dependencies Using Partitions, *In: Proc. the Fourteenth International Conference on Data Engineering,* (February 1998), pp. 392 - 401.

[15] S. Lopes , J.M. Petit , L. Lakhal, Efficient Discovery of Functional Dependencies and Armstrong Relations, in: Proc. ICDT 2000, the 7th International Conference on Extending Database Technology: Advances in Database Technology, vol 1777(March 27–31, 2000) 350 - 364.

[16] S.L. Wang, J.S. Tsai, B.C. Chang, "Mining Approximate Dependencies using partitions on Similarity-Relation-based Fuzzy databases", in : Proc. IEEE SMC'99, Vol. 6, pp. 871–875, 1999.