Determining Attributes to Maximize Visibility of Objects

Muhammed Miah, Gautam Das, Vagelis Hristidis and Heikki Mannila

Abstract—In recent years, there has been significant interest in the development of ranking functions and efficient top-*k* retrieval algorithms to help users in ad-hoc search and retrieval in databases (e.g., buyers searching for products in a catalog). We introduce a complementary problem: how to guide a seller in selecting the best attributes of a new tuple (e.g., a new product) to highlight so that it stands out in the crowd of existing competitive products and is widely visible to the pool of potential buyers. We develop several formulations of this problem. Although the problems are NP-complete, we give several exact and approximation algorithms that work well in practice. One type of exact algorithms is based on Integer Programming (IP) formulations of the problems. Another class of exact methods is based on maximal frequent itemset mining algorithms. The approximation algorithms are based on greedy heuristics. A detailed performance study illustrates the benefits of our methods on real and synthetic data.

Index Terms—Data mining, knowledge and data engineering tools and techniques, marketing, mining methods and algorithms, retrieval models.

1 Introduction

In recent years, there has been significant interest in developing effective techniques for ad-hoc search and retrieval in unstructured as well as structured data repositories, such as text collections and relational databases. In particular, a large number of emerging applications require exploratory querying on such databases; examples include users wishing to search databases and catalogs of products such as homes, cars, cameras, restaurants, or articles such as news and job ads. Users browsing these databases typically execute search queries via public front-end interfaces to these databases. Typical queries may specify sets of keywords in case of text databases, or the desired values of certain attributes in case of structured relational databases. The query-answering system answers such queries by either returning all data objects that satisfy the query conditions, or may rank and return the top-*k* data objects, or return the results that are on the query's skyline. If ranking is employed, the ranking may either be simplistic – e.g., objects are ranked by an attribute such as Price; or more sophisticated – e.g., objects may be ranked by the degree of "relevance" to the query. While unranked retrieval (also known as *Boolean Retrieval*) is more common in traditional SQL-based database systems, ranked retrieval (also known as *Top-k Retrieval*) is more common in text databases, e.g. tf-idf ranking [28]. Recently there has been widespread interest in developing suitable top-*k* retrieval techniques even for structured databases [1, 7, 30]. Skyline retrieval semantics is also investigated where a data point is retrieved by a query if it is not dominated by any other data point in all dimensions [4, 19, 22, 25, 29, 31].

In this paper we do *not* address new search and retrieval techniques that will aid users in effective exploration of such databases. Rather, the focus is on the complementary novel problem of selecting the data to be shown, elaborated as follows.

Selecting Attributes for Maximum Visibility We distinguish between two types of users of these databases: users who search such databases trying to locate objects of interest, and users who insert new objects into these databases in the hope that they will be easily discovered by the first type of users. For example, in a database representing an e-marketplace (such as Craigslist.org, or the classified ads section of newspapers), the former type of users are potential *buyers* of products, while the latter type of users are *sellers* of products. Products could range from apartments for rent to job advertisements to automobiles for sale. Almost all of the prior research efforts on effective search and retrieval techniques – such as new top-*k* algorithms, new relevance measures, and so on – have been designed with the first kind of user in mind (i.e., the buyer). In contrast, less research has been addressing techniques to help a seller/manufacturer insert a new product for sale in such databases that markets it in the best possible manner – i.e., such that it stands out in a crowd of competitive products and is widely visible to the pool of potential buyers.

It is this latter problem that is the main focus of this paper. To understand it a little better, consider the following scenario: assume that we wish to insert a classified ad in an online newspaper to advertise an apartment for rent. Our apartment may have numerous attributes (it has two

- *M. Miah is a PhD student of CSE at the University of Texas at Arlington, Arlington, TX. E-mail: md.miah@mavs.uta.edu.*
- *G. Das is with the Department of Computer Science & Engineering, University of Texas at Arlington, Arlington, TX. E-mail: gdas@uta.edu.*
- *V. Hristidis is with the School of Computing and Information Sciences, Florida International University, Miami, FL. E-mail: vagelis@cis.fiu.edu.*
- *H. Mannila is with HIIT, Helsinki University of Technology and University of Helsinki, Finland. E-mail: heikki.mannila@tkk.fi.*

Manuscript received (insert date of submission if desired). Please note that all acknowledgments should be placed at the end of the paper, before the bibliography.

bedrooms, electricity will be paid by the owner, it is near a train station, etc). However, due to the ad costs involved, it is not possible for us to describe all attributes in the ad. So we have to select, say the ten best attributes. Which ones should we select? Thus, one may view our effort as an attempt to build a recommendation system for *sellers*, unlike the more traditional recommendation systems for buyers. It may also be viewed as *inverting* a ranking function, i.e., determining the argument of a ranking function that will lead to high ranking scores.

This general problem also arises in domains beyond e-commerce applications. For example, in the design of a new product, a manufacturer may be interested in selecting the ten best features from a large wish-list of possible features – e.g., a homebuilder can find out that adding a swimming pool really increases visibility of a new home in a certain neighborhood. Likewise, we may be interested in developing a catchy title, or selecting a few important indexing keywords, for a scientific article.

To define our problem more formally, we need to develop a few abstractions. Let D be the database of products already being advertised in the marketplace (i.e., the "competition"). Based on the problem variant that we are considering, this database could be either a traditional relational table where tuples are products and columns are attributes (e.g., a Boolean database), or a collection of short text documents, where each text document is an ad for a product. In addition, let Q be the set of search queries that have been executed against this database in the recent past – thus Q is the "workload" or "query log". The query log is our primary model of what past potential buyers have been interested in. Based on the problem variant, the queries could be SQL-like selection queries or keyword queries that request for certain tuples to be returned from D. For a new product that needs to be inserted into this database, we assume that the seller has a complete "ideal" description of the product (e.g., a long list of all possible attributes and their values, or a detailed text description covering all possible features of the product). But due to budget constraints, there is a limit, say m, on the number of attributes/keywords that can be selected for entry into the database. Our problem can now be defined as follows:

PROBLEM: *Given a database D, a query log Q, a new tuple t, and an integer m, determine the best (i.e., top-m) attributes of t to retain such that if the shortened version of t is inserted into the database, the number of queries of Q that retrieve t is maximized.*

In this paper we initiate an investigation of this novel optimization problem. We consider several variants, including Boolean, categorical, text and numeric data, and conjunctive and disjunctive query semantics. We also consider variants in which the "budget", i.e., m, is not specified; in this case our objective is to determine the value of m such that the number of satisfied queries *divided by m* is maximized. Thus we seek to maximize the "per dollar" benefit. A special case of this no-budget variant is when ranking is performed using functions that are non-monotone on the number of specified attributes (keywords), such as the BM25 [26] scoring function used in Information Retrieval.

We analyze the computational complexity of these problems, and show that most variants are NP-complete. Nevertheless, we develop principled optimal algorithms for several of these problem variants that work well in practice. We develop two types of methods yielding optimal solutions: (a) techniques based on Integer Programming (IP) and Integer Linear Programming (ILP) methods, which work well for moderate-sized problem instances, and (b) more scalable solutions based on novel adaptations of maximal frequent set algorithms that also allow us to leverage several preprocessing opportunities. We also develop fast greedy approximation algorithms that work well for all problem variants, and present a thorough experimental study of all methods on real as well as synthetic data.

Main Contributions The main contributions of this paper may be summarized as follows:

1. We introduce the problem of *selecting attributes of a tuple for maximum visibility* as a new data exploration problem. We consider several interesting variants of the problem as well as diverse application scenarios.
2. We analyze the computational complexity of the different variants of the problem and show that most of them are NP-complete.
3. We develop optimal Integer Programming (*IP*) and Integer Linear Programming (*ILP*) based algorithms to solve certain variants of the problem. These algorithms are effective for moderate-sized problem instances.
4. For certain problem variants, we also develop more scalable optimal solutions based on novel adaptations of maximal frequent itemset algorithms. Furthermore, in contrast to *ILP*-based solutions, we can leverage preprocessing opportunities in these approaches.
5. We also develop fast greedy approximation algorithms that work well for all problem variants.
6. We perform detailed performance evaluations on both real as well as synthetic data to demonstrate the effectiveness of our developed algorithms.

The rest of the paper is organized as follows. In Section 2 we provide some preliminary definitions. In Section 3 we give a formal definition with computational complexity analysis of our main problem for Boolean data. The corresponding algorithms are given in Section 4. In Section 5 we discuss the problem variant for text data and provide algorithms. We present the result of extensive experiments in Section 6. In Section 7 we discuss other problem variants for Boolean, categorical and numeric data, analyze their computational complexity, and give algorithms for the problems. In Section 8 we discuss related work, and Section 9 is a short conclusion.

2 PRELIMINARIES

First we provide some useful definitions.

Boolean Database: Let $D = \{t_1 \ldots t_N\}$ be a collection of Boolean tuples over the attribute set $A = \{a_1 \ldots a_M\}$, where each tuple t is a bit-vector where a 0 implies the absence of a feature and a 1 implies the presence of a feature. A tuple t

may also be considered as a subset of A, where an attribute belongs to t if its value in the bit-vector is 1.

Tuple Domination: Let t_1 and t_2 be two tuples such that for all attributes for which tuple t_1 has value 1, tuple t_2 also has value 1. In this case we say that t_2 *dominates* t_1.

Tuple Compression: Let t be a tuple and let t' be a subset of t with m attributes. Thus t' represents a compressed representation of t. Equivalently, in the bit-vector representation of t, we retain only m 1's and convert the rest to 0's.

Query Log: Let $Q = \{q_1 \ldots q_S\}$ be collection of queries where each query q defines a subset of attributes.

The following running example will be used throughout the paper to illustrate various concepts.

EXAMPLE 1: *Consider an inventory database of an auto dealer, which contains a single database table D with N rows and M attributes where each tuple represents a car for sale. The table has numerous attributes that describe details of the car: Boolean attributes such as AC, Four Door, Turbo, Power Doors, Auto Trans, Power Brakes, etc; categorical attributes such as Make, Color, Engine Type, Zip Code, etc; numeric attributes such as Price, Age, Fuel Mileage, etc; and text attributes such as Reviews, Accident History, and so on. Fig 1 illustrates such a database (where only the Boolean attributes are shown) of seven cars already advertised for sale. The figure also illustrates a query log of five queries, and a new car t that needs to be advertised, i.e., inserted into this database.* □

3 Main Problem Variant: Conjunctive Boolean With Query Log

In this section we formally define the main problem for Boolean data. As we discuss in Section 7, many other variants can be reduced to this problem. In Section 3.1 we formally define the problem and in Section 3.2 we present our complexity results. In Section 4 we provide algorithms for this problem variant. We first defined the query semantics to be used in the beginning.

Conjunctive Boolean Retrieval: We view each query as a conjunctive query. A tuple t satisfies a query q if q is a subset of t. For example, a query such as $\{a_1, a_3\}$ is equivalent to "return all tuples such that $a_1 = 1$ and $a_3 = 1$". Alternatively, if we view q as a special type of "tuple", then t dominates q. The set of returned tuples $R(q)$ is the set of all tuples that satisfy q.

In this problem variant as well as in most of the variants defined later, our task is to compress a new tuple t by retaining the best set of m attributes (i.e., top-m attributes) such that some criterion is optimized.

3.1 Problem Definition

Conjunctive Boolean - Query Log (*CB-QL*): *Given a query log Q with Conjunctive Boolean Retrieval semantics, a new tuple t, and an integer m, compute a compressed tuple t' having m attributes such that the number of queries that retrieve t' is maximized.*

Intuitively, for buyers interested in browsing products of interest, we wish to ensure that the compressed version of the new product is visible to as many buyers as possible.

EXAMPLE 1 (cont'd): *To illustrate this problem variant, consider the example in Fig 1, which shows a new tuple t that needs to be inserted into the database. Suppose we are required to retain m = 3 attributes. It is not hard to see that if we retain the attributes AC, Four Door, and Power Doors (i.e., t' = [1, 1, 0, 1, 0, 0]), we can satisfy a maximum of three queries (q_1, q_2, and q_3). No other selection of three attributes of the new tuple will satisfy more queries.* □

This problem also has a *per-attribute* version where m is not specified; in this case we may wish to determine t' such that the number of satisfied queries divided by $|t'|$ is maximized. Intuitively, if the number of attributes retained is a measure of the cost of advertising the new product, this problem seeks to maximize the number of potential buyers per advertising dollar.

Notice that in ***CB-QL***, it is the query log Q that needs to be analyzed in solving the problem; the actual database D (i.e., the "competing products") is irrelevant. We consider variants that involve the products database in Section 7.

Car ID	*AC*	*Four Door*	*Turbo*	*Power Doors*	*Auto Trans*	*Power Brakes*
t_1	0	1	0	1	0	0
t_2	0	1	1	0	0	0
t_3	1	0	0	1	1	1
t_4	1	1	0	1	0	1
t_5	1	1	0	0	0	0
t_6	0	1	0	1	0	0
t_7	0	0	1	1	0	0

Database *D*

Query ID	*AC*	*Four Door*	*Turbo*	*Power Doors*	*Auto Trans*	*Power Brakes*
q_1	1	1	0	0	0	0
q_2	1	0	0	1	0	0
q_3	0	1	0	1	0	0
q_4	0	0	0	1	0	1
q_5	0	0	1	0	1	0

Query Log *Q*

New Car	*AC*	*Four Door*	*Turbo*	*Power Doors*	*Auto Trans*	*Power Brakes*
t	1	1	0	1	1	1

New tuple *t* to be inserted

Fig. 1. Illustrating EXAMPLE 1

3.2 NP-Completeness Results

Theorem 1: The decision version of *CB-QL* problem is NP-hard.

Proof: An instance of decision version of CB-QL is similar to an instance of CB-QL, except that it has in addition a target parameter X, and is satisfied if there is a compressed tuple t' having m attributes such that the number of queries that retrieve t' is at least X. Clearly, the decision version of CB-QL is in NP. To prove that it is NP-complete, we reduce from the Clique problem. Given a graph G = (V, E) and an integer r, the task in the Clique problem is to check if there is a clique of size r in G. We transform this to an instance of decision version of CB-QL

as follows. The attribute set A will be V, and the query log will contain one row for each edge. If e = (u, v) is an edge, then the query log Q contains the conjunctive query {u, v}, i.e., the query retrieving all tuples with u = 1 and v = 1. The new tuple t has all the attributes in V set to 1. Let m = r and X = m(m-1)/2. Now t has a compressed representation with m attributes that satisfies X number of queries if and only if the graph has a clique of size r. □

4 Algorithms for Conjunctive Boolean With Query Log

In this section we discuss our main algorithmic results for the main problem variant discussed in Section 3.

4.1 Optimal Brute Force Algorithm

Clearly, since *CB-QL* is NP-hard, it is unlikely that any optimal algorithm will run in polynomial time in the worst case. The problem can be obviously solved by a simple brute force algorithm (henceforth called *Brute-Force-CB-QL*), which simply considers all combinations of *m*-attributes of the new tuple *t* and determines the combination that will satisfy the maximum number of queries in the query log *Q*. However, we are interested in developing optimal algorithms that work much better for typical problem instances. We discuss such algorithms next.

4.2 Optimal Algorithm based on Integer Linear Programming

We next show how *CB-QL* can be described in an integer linear programming (*ILP*) framework. Let the new tuple be the Boolean vector $t = \{a_1(t),...,a_M(t)\}$, S be the total number of queries in the query log, and let $x_1,...,x_M$ be integer variables such that if $a_i(t) = 1$ then $x_i \in \{0, 1\}$, else $x_i = 0$. Consider the task:

$$\text{Maximize} \sum_{i=1}^{S} \prod_{a_j q_i = 1} x_j \text{ subject to } \sum_{j=1}^{M} x_j \le m$$

It is easy to see that the maximum gives exactly the solution to *CB-QL*. The objective function is not linear, however, and thus we next show how this can be achieved.

We introduce additional 0-1 integer variables $y_1,...,y_S$, i.e., one variable for each query in *Q*. The formulation is

$$\text{Maximize} \sum_{i=1}^{S} y_i \text{ subject to}$$

$$\sum_{j=1}^{M} x_j \le m \text{ and } y_i \le x_j \text{ for each } j \text{ \& } i \text{ such that } a_j q_i = 1$$

$y_i \le x_j$ actually means $y_i = x_j = 1$ if $y_i = 1$, that is, if q_i is satisfied. Thus, the variable y_i corresponding to a query can be 1 only if all the variables x_j corresponding to the attributes in the query are 1. This implies that the maximum remains the same. We refer to the above algorithm as *ILP-CB-QL*. The integer *linear* formulation is particularly attractive as unlike more general IP solvers, ILP solvers are usually more efficient in practice.

4.3 Optimal Algorithm based on Maximal Frequent Itemsets

The algorithm based on Integer Linear Programming described in the previous subsection has certain limitations; it is impractical for problem instances beyond a few hundred queries in the query log. The reason is that it is a very generic method for solving arbitrary integer linear programming formulations, and consequently fails to leverage the specific nature of our problem. In this subsection we develop an alternate approach that scales very well to large query logs. This algorithm, called *MaxFreqItemSets-CB-QL*, is based on an interesting adaptation of an algorithm for mining *Maximal Frequent Itemsets* [12]. We first define the frequent itemset problem:

4.3.1 The Frequent Itemset Problem: *Let R be an N-row M-column Boolean table, and let r > 0 be an integer known as the* threshold. *Given an itemset I (i.e., a subset of attributes), let freq(I) be defined as the number of rows in R that "support" I (i.e., the set of attributes corresponding to the 1's in the row is a superset of I). Compute all itemsets I such that freq(I) > r.*

Computing frequent itemsets is a well studied problem and there are several scalable algorithms that work well when *R* is sparse and the threshold is suitably large. Examples of such algorithms include [2, 15]. In our case, given a new tuple *t*, recall that our task is to compute *t'*, a compression of *t* by retaining only *m* attributes, such that the number of queries that satisfy *t'* is maximized. This immediately suggests that we may be able to leverage algorithms for frequent itemsets mining over *Q* for this purpose. However, there are several important complications that need to be overcome, which we elaborate next.

4.3.2 Complementing the Query Log: Firstly, in itemset mining, a row of the Boolean table is said to support an itemset if the row is a superset of the itemset. In our case, a query satisfies a tuple if it is a subset of the tuple. To overcome this conflict, our first task is to *complement* our problem instance, i.e., convert 1's to 0's and vice versa. Let ~*t* (~*q*) denote the complement of a tuple *t* (query *q*), i.e., where the 1's and 0's have been interchanged. Likewise let ~*Q* denote the complement of a query log *Q* where each query has been complemented. Now, *freq*(~*t*) can be defined as the number of rows in ~*Q* that support ~*t*.

Rest of the approach is now seemingly clear: compute all frequent itemsets of ~*Q* (using an appropriate threshold to be discussed later), and from among all frequent itemsets of size *M* – *m*, determine the itemset *I* that is a superset of ~*t* with the highest frequency. The optimal compressed tuple *t'* is therefore the complement of *I*, i.e., ~*I*.

However, the problem is that *Q* is itself a sparse table, as the queries in most search applications involve the specification of just a few attributes. Consequently, the complement ~*Q* is an extremely dense table, and this prevents most frequent itemset algorithms from being directly applicable to ~*Q*. For example, most "level-wise algorithms" (such as Apriori [2], which operates level by level of the Boolean lattice over the attributes set by first computing the single itemsets, then itemsets of size 2, and so on) will only progress past just a few initial levels before being overcome by an intractable explosion in the size of candidate sets. To see this, consider a table with *M=50* attributes, and let *m = 10*. To determine a compressed tuple *t'* with 10 attributes, we need to know the itemset of ~*Q* of size 40 with maximum frequency. Due to the dense na-

ture of ~Q, algorithms such as Apriori will not be able to compute frequent itemsets beyond a size of 5-10 at the most. Likewise, the sheer number of frequent itemsets will also prevent other algorithms such as FP-Tree [15] from being effective.

We have developed an adaptation of frequent itemset mining algorithms to overcome this problem of extremely dense datasets. Before we describe details of our approach, let us discuss the issue of how the threshold parameter should be set.

4.3.3 Setting of the Threshold Parameter: Let us assume we can solve the problem of itemset mining of extremely dense datasets. What should be the setting of the threshold? Clearly setting the threshold $r=1$ will solve *CB-QL* optimally. But this is likely to make any itemset mining algorithm impractically slow.

There are two alternate approaches to setting the threshold. One approach is essentially a heuristic, where we set the threshold to a reasonable fixed value dictated by the practicalities of the application. The intuition is that the threshold enforces that attributes should be selected such that the compressed tuple is satisfied by a certain minimum number of queries. This is reasonable in many practical applications, as the eventual goal is to make the compressed tuple visible to as many users as possible. For example, a threshold of 1% means that we are not interested in results that satisfy less than 1% of the queries in the query log, i.e., we are attempting to compress t such that at least 1% of the queries are still able to retrieve the tuple. It is important to note that for a fixed threshold setting such as this, one of two possible outcomes can occur. If the optimal compression t' satisfies more than 1% of the queries, the algorithm will discover it. If the optimal compression satisfies less than 1% of the queries, then the algorithm will return empty.

We also suggest an alternate adaptive procedure of setting the threshold that is guaranteed to find the optimal compression. First initialize the threshold to a high value and compute the frequent itemsets of ~Q. If there are no frequent itemsets of size at least $M - m$ that are supersets of ~t, repeat the process with a smaller threshold which is half of the previous threshold. This process is guaranteed to discover the optimal t'.

We now return to the task of how to compute frequent itemsets of the dense Boolean table ~Q. In fact, we do not compute all frequent itemsets of the dense table ~Q, as we have already argued earlier that there will be prohibitively too many of them. Instead, our approach is to compute the *maximal frequent itemsets* of ~Q.

4.3.4 Random Walk to Compute Maximal Frequent Itemsets: A maximal *frequent itemset* is a frequent itemset such that none of its supersets are frequent. The set of maximal frequent itemsets are much smaller than the set of all frequent itemsets. For example, if we have a dense table with M attributes, then it is quite likely that most of the maximal frequent itemsets will exist very high up in the Boolean lattice over the attributes, very close to the highest possible level M. Fig 2 shows a conceptual diagram of a Boolean lattice over a dense Boolean table ~Q. The shaded region depicts the frequent itemsets and the maximal frequent itemsets are located at the highest positions of the border between the frequent and infrequent itemsets.

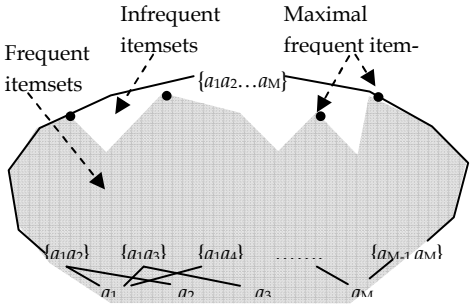


Fig. 2. Maximal frequent itemsets in a Boolean Lattice

There exist several algorithms for computing maximal frequent itemsets, e.g. [3, 5, 12, 14]. We base our approach on the *random walk* based algorithm in [12], which starts from a random singleton itemset I at the bottom of the lattice, and at each iteration, adds a random item to I (from among all items A - I such that I remains frequent), until no further additions are possible. At this point a maximal frequent itemset I has been discovered. If the number of maximal frequent itemsets is relatively small, this is a practical algorithm: repeating this random walk a reasonable number of times will with high probability discover all maximal frequent itemsets. However, since this algorithm is based on traversing the lattice from bottom to top, it implies that the random walk will have to traverse a lot of levels before it reaches a maximal frequent itemset of a dense table.

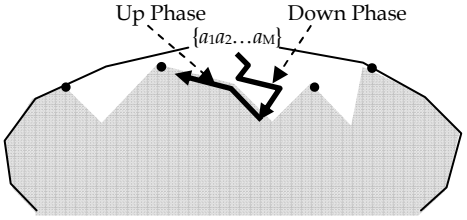


Fig. 3. Two phase random walk

Instead, we propose an alternate approach which starts from the top of the lattice and traverses down. Our random walk can be divided into two phases: (a) *Down Phase*: starting from the top of the lattice (I = {$a_1a_2…a_M$}), walk down the lattice by removing random items from I until I becomes frequent, and (b) *Up Phase*: starting from I, walk up the lattice by adding random items to I (from among all items A - I such that I remains frequent), until the no further additions are possible. At this point a maximal frequent itemset I has been discovered.

Fig 3 shows an example of the two phases of the random walk. What is important to note is that this process is much more efficient than a bottom-up traversal, as our walks are always confined to the top region of the lattice and we never have to traverse too many levels. Complementing the query log eventually results in a dense dataset. In a dense dataset, maximal frequent itemsets are usually at the top of the lattice. That is, they are close to level M - m, where m is comparatively much smaller than M. If a bottom-up approach is used to find maximal frequent itemsets, it will have to traverse a long portion of

the lattice (i.e., too many levels) and will be inefficient. Whereas, in top-down approach, the first phase tries to find the first frequent itemset along the path from the top which is usually close to a maximal frequent itemset, the walks are confined to the top region of the lattice and we never have to traverse too many levels.

4.3.5 Complexity Analysis of a Random Walk Sequence: In the worst case, the cost of a down-up random walk is $2 \cdot M \cdot |Q|$, where M is the total number of attributes and $|Q|$ is the size of the query log. Although in the worst case the random walk will go up and down the whole lattice, in practice we only expect each portion of the walk to traverse only a few levels at the top of the lattice.

4.3.6 Number of Iterations: Repeating this two phase random walk several times will discover, with high probability, all the maximal frequent itemsets. The actual number of such iterations can be monitored adaptively; our approach is to stop the algorithm if each discovered maximal frequent itemset has been discovered at least twice (or a maximum number of iterations have been reached). This stopping heuristic is motivated by the *Good-Turing estimate* for computing the number of different objects via sampling [9]. Good–Turing frequency estimation is a statistical technique for predicting the probability of occurrence of objects belonging to an unknown number of species, given past observations of such objects and their species.

4.3.7 Frequent Itemsets at Level $M - m$: Finally, once all maximal frequent itemsets have been computed, we have to check which ones are supersets of $\sim t$. Then, for all possible subsets (of size $M - m$) of each such maximal frequent itemset (see Fig 4) , we can determine that subset I that is (a) a superset of $\sim t$, and (b) has the highest frequency. The optimal compressed tuple t' is therefore the complement of I, i.e., $\sim I$.

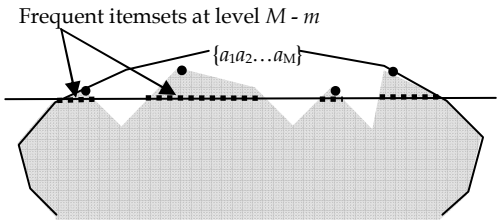


Fig. 4. Checking frequent itemsets at level $M - m$

In summary, the pseudo-code of our algorithm *MaxFreqItemSets-CB-QL* is shown in Fig. 5. Details of how certain parameters such as the threshold are set, are omitted from the pseudo-code.

4.3.8 Preprocessing Opportunities: Note that the algorithm also allows for certain operations to be performed in a preprocessing step. For example, all the maximal itemsets can be precomputed, and the only task that needs to be done at runtime is to determine, for a new tuple t, those itemsets that are supersets of $\sim t$ and have size $M - m$. If we know the range of m that is usually requested for compression in new tuples, we can even precompute all frequent itemsets for those values of m, and lookup the itemset with the highest frequency at runtime.

4.3.9 The Per-Attribute Variant: *CB-QL* has a per-attribute variant, where m is not provided as an input, and we have to determine the best m such that number of satisfied queries divided by m is maximized. This variant can be simply solved by trying out values of m between 1 and M and making M calls to any of the algorithms discussed above, and selecting the solution that maximizes our objective. Since we adopt this general strategy for all per-attribute problem variants, we do not discuss such variants any further in this paper.

Q: Query Log
t: new tuple
m: num attributes of t to retain
r: threshold ← suitable value
MaxFreqItemsets ← {}
MaxNumIter ← suitable value

Algorithm *TwoPhase-Random-Walk*($\sim Q$, r)
  execute Down Phase random walk
  execute Up Phase random walk
  return itemset reached after Up Phase

Algorithm *ComputeMaxFreqItemsets*($\sim Q$, r)
  while
    (i++ ≤ *MaxNumIter*) *and*
    ($\exists$ J in *MaxFreqItemsets s.t.*
      timesDiscovered(J)= 1)
    $I \leftarrow$ *TwoPhase-Random-Walk*($\sim Q$, r)
    timesDiscovered(I)++
    MaxFreqItemsets←*MaxFreqItemsets* ∪ {I}

Algorithm *MaxFreqItemSets-CB-QL*($\sim Q$, r)
  ComputeMaxFreqItemsets($\sim Q$, r)
  let Itemsets(t) ← {I | $I \subseteq$ *MaxFreqItemsets*, $|I| = M - m$, *and* $I \supseteq \sim t$)}
  let I be the itemset in *Itemsets*(t) with highest frequency
  return $\sim I$

Fig. 5. Algorithm *MaxFreqItemSets-CB-QL*

4.4 Greedy Heuristics

While the maximal frequent itemset based algorithm has much better scalability properties than the ILP based algorithm, it also becomes prohibitively slow for really large datasets (query logs). Consequently, we also developed suboptimal greedy heuristics for solving *CB-QL*; in our experiments the results were quite good. We briefly describe the heuristics here.

The algorithm *ConsumeAttr-CB-QL* first computes the number of times each individual attribute appear in the query log. It then selects the top-m attributes of the new tuple that have the highest frequencies.

The algorithm *ConsumeAttrCumul-CB-QL* is a cumulative version of *ConsumeAttr-CB-QL*. It first selects the attribute with the highest individual frequency in the query log. It then selects the second attribute that co-occurs most frequently with the first attribute in the query log, and so on. Instead of consuming attributes greedily, an alternative approach is to consume queries greedily. The algorithm *ConsumeQueries-CB-QL* operates as follows. It first picks the query with minimum number of attributes, and selects all attributes specified in the query. It then picks the

query with minimum number of new attributes (i.e., not already specified in the first query), and adds these new attributes to the selected list. This process is continued until m attributes have been selected.

5 Problem Variant for Text Data

5.1 *Text Data* Problem Definition

A text database consists of a collection of documents, where each document is modeled as a bag of words as is common in Information Retrieval. Queries are sets of keywords, with top-k retrieval via query-specific scoring functions, such as the tf-idf-based BM25 scoring function [26]. *Tk-QR* (described later in Section 7.2) can be directly mapped to a corresponding problem for text data if we view a text database as a Boolean database with each distinct keyword considered as a Boolean attribute. This problem arises in several applications, e.g., when we wish to post a classified ad in an online newspaper and need to specify important keywords that will enable the ad to be visible to the maximum number of potential buyers. A subtle point is that, due to the non-monotonicity of many IR ranking functions, it is possible that a top-m_1 tuple compression is worse (fewer queries retrieve document t') than a top-m_2 compression, where $m_1 > m_2$. The non-monotonicity is usually due to the document length parameter that decreases the score of a document as its length increases.

The attribute selection problem for text data is also NP-complete as it can be converted into Boolean problem considering each keyword as a Boolean attribute. The problem is NP-hard for the case of monotone ranking function, as in *CB-QL*. Hence, it is also NP-hard for the more complex non-monotonic ranking functions.

5.2. Algorithms for *Text Data*

As discussed above, text data can be treated as Boolean data, and all the algorithms developed for Boolean data can be used for text data. There are two issues that we wish to highlight, however. One is that if we view each distinct keyword in the text corpus (or query log) as a distinct Boolean attribute, the dimension of the Boolean database is enormous. Consequently, none of the optimal algorithms, either IP-based or frequent itemset-based, are feasible for text data. Fortunately, the greedy heuristics we have developed scale very well with reasonable results, as described in the experiments section. The second issue is that some of the scoring function that are used in text data – e.g., the BM25 scoring function that takes into account the document length (size of compressed tuple t') – are non-monotonic on the number of keywords added. In particular, adding a query keyword to t' may decrease its BM25 score if this keyword has very low inverse document frequency (idf). Consequently the per-attribute versions of our various problem variants are of interest.

6 Experiments

In this section we describe the experimental setting and the results. Our main performance indicators are (a) the time cost of the proposed optimal and greedy algorithms, and (b) the approximation quality of the greedy algorithms, for the *CB-QL* and *text data* problem variants presented in Sections 3 and 5 respectively. Note that no experimental results are presented for the problem variants of Section 7 since their algorithms are usually adaptations of the ones for *CB-QL*.

System Configuration: We used Microsoft SQL Server 2000 RDBMS on a P4 3.2-GHZ PC with 1 GB of RAM and 100 GB HDD for our experiments. Algorithms are implemented in C#, and connected to RDBMS through ADO.

Datasets: We used two datasets, a cars dataset for the Boolean data experiments (Section 6.1), and a publications titles dataset for the text data experiments (Section 6.2). In particular, we use an online used-cars dataset consisting of 15,191 cars for sale in the Dallas area extracted from autos.yahoo.com. There are 32 Boolean attributes, such as *AC*, *Power Locks*, etc. We used a real workload of 185 queries created by users at UT Arlington, as well as synthetic workloads. In the synthetic workload, each query specifies 1 to 5 attributes chosen randomly distributed as follows: 1 attribute – 20%, 2 attributes – 30%, 3 attributes – 30%, 4 attributes – 10%, 5 attributes – 10%. We assume that most of the users specify two or three attributes.

The publication titles dataset consists of 119,332 titles extracted from the DBLP (http://dblp.uni-trier.de/xml/) database for the major database forums including SIGMOD, VLDB, PODS, ICDE, ICDT and EDBT. These titles have a total of 59,184 distinct (non-stop word) keywords. A query workload of 150 real user queries is used, which was created as follows: We asked 7 MS and PhD students from UTA and FIU to create about 20 queries each that they would pose to the DBLP dataset, containing 1-4 keywords. This query log has a total of 205 distinct (non-stop word) keywords.

6.1 Boolean Data

We focus on *CB-QL*, which can be solved by a superset of the algorithms used in the other variants.

The top-m attributes selected by our algorithms seem promising. For example, even with a small real query log of 185 queries, our optimal algorithms could select top features specific to the car, e.g., sporty features are selected for sports cars, safety features are selected for passenger sedans, and so on.

We first compare the execution times of the optimal and greedy algorithms that solve *CB-QL*. These are (Section 4): *ILP-CB-QL*, *MaxFreqItemSets-CB-QL*, which produce optimal results, and *ConsumeAttr-CB-QL*, *ConsumeAttrCumul-CB-QL*, and *ConsumeQueries-CB-QL*, which are greedy approximations. The *CB-QL* suffix is skipped in the graphs for clarity.

Fig 6 shows how the execution times vary with m for the real query workload, averaged over 100 randomly selected to-be-advertised cars from the dataset. Note that different y-axis scales are used for the two optimal and the three greedy algorithms to better display the differences among the methods. The *MaxFreqItemSets* algorithm consistently performs better than the *ILP* algorithm. An-

other interesting observation is that the cost of *ILP* does not always increase with *m*. The reason seems to be that the *ILP* solver is based on branch and bound, and for some instances the pruning of the search space is more efficient than for others.

The times in Fig 6 for *MaxFreqItemSets* also include the preprocessing stage, which can be performed once in advance regardless of the new tuple (user car), as explained in Section 4.3. If the pre-processing time is ignored, then *MaxFreqItemSets* takes only approximately 0.015 seconds to execute for any *m* value.

Fig 7 shows the quality, that is, the numbers of satisfied queries for the greedy algorithms along with the optimal numbers, for varying *m*. The numbers of queries are averaged over 100 randomly selected to-be-advertised cars from the dataset. Note that no query is satisfied for *m* = 3 because all queries specify more than 3 attributes. We see that *ConsumeAttr* and *ConsumeAttrCumul* produce near-optimal results. In contrast, *ConsumeQueries* has low quality, since it is often the case that the attributes of the queries with few attributes (which are selected first) are not common in the workload.

Fig 8 and Fig 9 repeat the same experiments for the synthetic query workload of 2000 queries. In Fig 8, we do not include the *ILP* algorithm, because it is very slow for more than 1000 queries (as also shown in Fig 10).

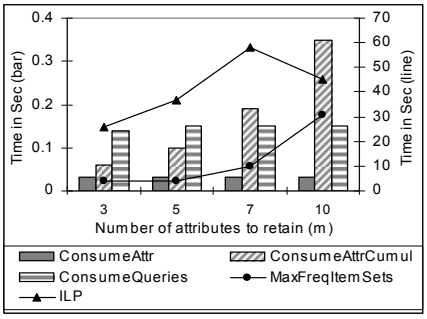


Fig. 6. Execution times for *CB-QL* for varying *m*, for real workload of 185 queries.

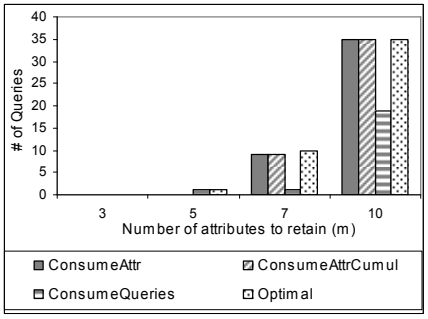


Fig. 7. Satisfied queries for greedy and optimal algorithms for *CB-QL* varying *m*, for real workload of 185 queries.

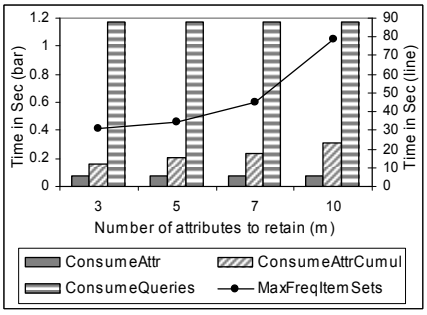


Fig. 8. Execution times for *CB-QL* for varying *m*, for the synthetic workload of 2000 queries.

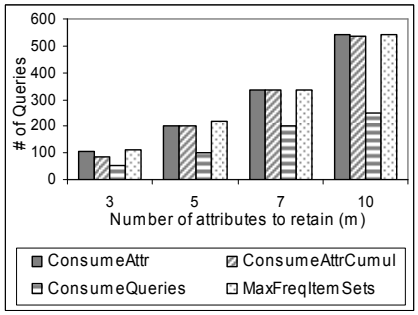


Fig. 9. Satisfied queries for greedy and optimal algorithms for *CB-QL* for varying *m*, for synthetic workload of 2000 queries.

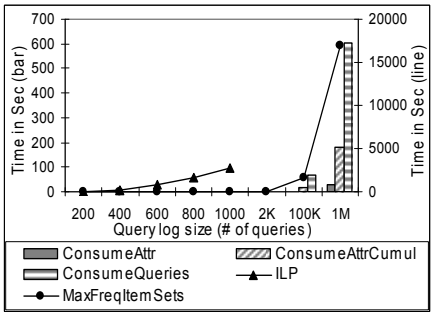


Fig. 10. Execution times for *CB-QL* for varying synthetic workload size for *m* = 5.

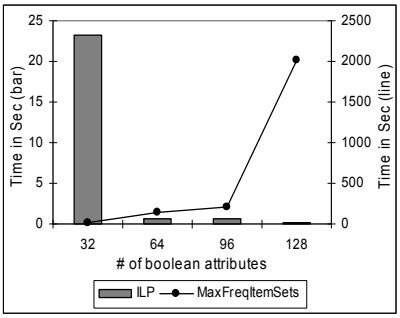


Fig. 11. Execution times for *CB-QL* for varying number of total attributes for the synthetic workload of 200 queries for *m* = 5

Next, we measure the execution times of the algorithms for varying query log size and number of attributes. Fig 10 shows how the average execution time varies with the query log size, where the synthetic workloads were created as described earlier in this section. We observe that *ILP* does not scale for large query logs; this is why there are no measurements for *ILP* for more than 1000 queries. *ConsumeQueries* performs consistently worse than other greedy algorithms since we make a pass on the whole workload at each iteration to find the next query to add. We conclude *ConsumeQueries* is generally a bad choice.

Fig 11 focuses on the two optimal algorithms, and measures the execution times of the algorithms, averaged over 100 randomly selected to-be-advertised cars from the dataset, for varying number *M* of total attributes of the dataset and queries, for a synthetic query log of 200 queries. We observe that *ILP* is faster than *MaxFreqItemSets* for more than 32 total attributes. For 32 total attributes *MaxFreqItemSets* is faster as also shown in Fig 6. However, note that *ILP* is only feasible for very small query logs. For larger query logs, *ILP* is very slow or infeasible, as is also shown by the missing values in Fig 10. To summarize, *ILP* is better for small query logs and many total attributes (i.e. short and wide query log), whereas *MaxFreqItemSets* is better for larger query logs with fewer total attributes (i.e. long and narrow query log). However for query logs those are long and wide, the problem becomes truly intractable, and approximation methods such as our greedy algorithms perhaps the only feasible approaches.

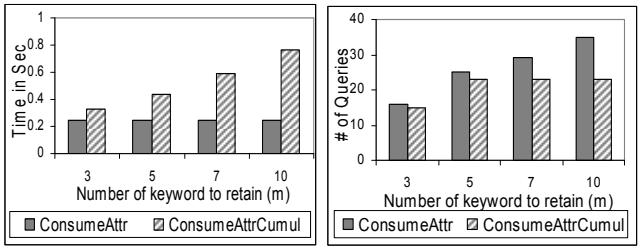
6.2 Text Data

We use a simplified version of the BM25 [26] ranking function, for the case of ad-hoc retrieval where any repetition of terms in the query is ignored. The weight of a term *j* is computed by the following formula:

$$w_j(\bar{d}, C) := \frac{(k_1+1)d_j}{k_1((1-b)+b\frac{dl}{avdl})+d_j} \log\frac{N-df_j+0.5}{df_j+0.5}$$

where d_j is the term frequency and df_j is the document frequency of term j, dl is the document length, $avdl$ is the average document length across the collection, and k_1 and b are free parameters (we set $k_1 = 1$ and $b = 0.5$).

The number of distinct keywords (equivalent to the number of attributes in the Boolean problem) for our titles dataset is 59,184. We randomly selected 50 abstracts of papers to be used as the to-be-advertised tuples. In fact, we retrieved the abstracts of 50 of the papers in our titles dataset. We consider the rest 59, 134 (59,184-50) titles as our query log That is, the tuple t is the abstract, and the compressed t' is the "best" keywords from the abstract to be used in the title. The titles of these papers were removed from the dataset. Fig 12 shows how the greedy algorithms *ConsumeAttr* and *ConsumeAttrCumul* perform in terms of execution time and in terms of quality, for varying m. We set $k = 20$ (recall that we want to maximize the number of queries in the workload for which the to-be-advertised tuple is in the top-k results). As mentioned in Section 5.2, no optimal algorithm is feasible due to the large number of total attributes (distinct keywords). Finally, we note that the *ConsumeQueries* greedy algorithm is inappropriate for the same reason, since it would just select the keywords of the shortest one or two titles, which most likely satisfy no other queries.



(a) Execution times (b) Number of covered queries

Fig. 12. Execution times and Number of covered queries for greedy algorithms for *text data* for varying top-m.

7 Other Problem Variants

In this section we discuss several other problem variants for Boolean as well as categorical and numeric data.

7.1 Conjunctive Boolean - Data (*CB-D*)

This is the situation where we only have access to the database of existing products and no access to the query log. In this case the strategy is to search for a compressed version of the new product that dominates as many products in the database as possible. Thus any buyer that executes a query (with Conjunctive Boolean Retrieval) that selects a dominated product will also get to see the new product.

7.1.1 Problem Definition (*CB-D*): *Given a database D, a new tuple t, and an integer m, compute a compressed tuple t' by retaining m attributes such that the number of tuples in D dominated by t' is maximized.*

EXAMPLE 1 (cont'd): *To illustrate this problem variant, consider the example in Fig 1 again. Suppose we are required to retain m = 4 attributes of the new tuple t. It is not hard to see that if we retain the four attributes AC, Four Door, Power Doors and Power Brakes (i.e., t' = [1, 1, 0, 1, 0, 1]), we dominate four tuples (t_1, t_4, t_5 and t_6). No other selection of four attributes of the new tuple will dominate more tuples.* □

7.1.2 Complexity Results for *CB-D*: The same proof in section 3.2 for the problem *CB-QL* obviously tells also that *CB-D* is NP-hard. *CB-D* also has a per-attribute version which can be naturally defined.

7.1.3 Algorithms for *CB-D*: Any of the above algorithms that solve *CB-QL* can be also used to solve *CB-D*, by simply replacing the query log with the database as input.

7.2 Top-*k* - Global Ranking (*Tk-GR*) and Top-*k* - Query-Specific Ranking (*Tk-QR*)

We consider Top-k Retrieval via Global and Query-Specific Scoring Function for these problem variants.

Top-k Retrieval via Global Scoring Function: Let *Score*(t) be a function that returns a real-valued score for any tuple t. Let k be a small integer associated with a query q. Then $R(q)$ is defined as the set of top-k tuples[1] in the database with the highest scores that satisfy q. Global scoring functions capture the "global importance" of tuples, e.g., the price of a product.

Top-k Retrieval via Query-Specific Scoring Function: Let *Score*(q, t) be a scoring function that returns a real-valued score for any tuple t. Let k be a small integer associated with a query q. Then $R(q)$ is defined as the set of top-k tuples in the database with the highest scores. Note that here we do not insist that the queries are conjunctive, i.e., tuples that do not satisfy all attributes specified in the query may also be returned. An example of a query specific scoring function is the dot product of q and t.

7.2.1 Problem Definition (*Tk-GR*): *Given a database D, a query log Q with Top-k Retrieval via Global Scoring Function, a new tuple t, and an integer m, compute a compressed tuple t' by retaining m attributes such that the number of queries that retrieve t' is maximized.*

7.2.2 Problem Definition (*Tk-QR*): *Given a database D, a query log Q with Top-k Retrieval via Query-Specific Scoring Function, a new tuple t, and an integer m, compute a compressed tuple t' by retaining m attributes such that the number of queries that retrieve t' is maximized.*

EXAMPLE 1 (cont'd): *To illustrate Tk-GR, assume that each query in the query log returns the top-2 tuples (i.e., k = 2) ordered by decreasing Fuel Efficiency, where Fuel Efficiency is a numeric attribute not shown in Fig 1. [2]Let us assume that the fuel efficiencies of the seven cars t_1 … t_7 are 10mpg, 20mpg … 70mpg respectively. The results of executing the five queries in the query log on the database are shown in Fig 13(a).*

Assume the fuel efficiency of the new tuple t is 35mpg and we are required to retain m = 3 of its Boolean attributes. We first argue that the selections suggested in CB-QL are suboptimal. Suppose we decide to retain the three attributes AC, Four Door, and Power Doors as suggested in CB-QL. The new compressed tuple t' will satisfy the Boolean conditions of queries q_1, q_2, and q_3, however, it will not be returned among the top-2 tuples of q_1

[1] *If the number of tuples that satisfy q is less than k, then all such tuples are returned*

[2] *For now, we assume that only Boolean attributes can be specified in queries. Allowing numeric ranges to be specified in queries is discussed in Section 7.6.2*

and q_3, because it has a lower score (fuel efficiency) than t_4 (the second tuple returned by both q_1 and q_3). Consequently, the compressed tuple t' will be returned by only one query, q_2 (by replacing t_3 as the second tuple to be returned, since t_3 has lower fuel efficiency than t'). In contrast, if we decide to retain AC, Power Doors and Power Brakes, the compressed tuple t' will satisfy the Boolean conditions of two queries, q_2 and q_4, and moreover, will replace t_3 as the second tuple to be returned by each query. No other selection of three attributes of the new tuple will ensure that it gets returned by more queries. □

Query ID	Top-2 tuples with scores
q1	t5 (50), t4 (40)
q2	t4 (40), t3 (30)
q3	t6 (60), t4 (40)
q4	t4 (40), t3 (30)
q5	∅

(a) Global scoring function

Query ID	Top-3 tuples with scores
q1	t4 (2), t5 (2), t1 (1)
q2	t3 (2), t4 (2), t1 (1)
q3	t1 (2), t4 (2), t6 (2)
q4	t3 (2), t4 (2), t1 (1)
q5	t2 (1), t3 (1), t7 (1)

(b) Query specific scoring function (dot product)

Fig. 13. Results of Top-*k* Retrieval

We now discuss *Tk-QR*.

EXAMPLE 1 (cont'd): *Assume that each query in the query log returns the top-3 tuples (i.e., k = 3), where the query-specific scoring function is the dot product between a query and a tuple. Recall that in this case a query is no longer a conjunctive query – tuples that do not dominate a query may also be returned. Based on this scoring function, the results of the execution of the five queries are shown in Fig 13(b) (score ties have been broken arbitrarily). Suppose we are required to retain m = 4 attributes. It is not hard to see that if we retain the attributes AC, Four Door, Power Doors and Power Brakes, the compressed tuple will definitely enter into the top-3 result tuples of q1, q2 and q4 respectively (it will also tie scores with the top-3 tuples of q3). No other selection of three attributes of the new tuple will ensure that it gets returned by more queries.* □

Monotonicity of Scoring Functions: One final issue regarding *Tk-QR* needs to be discussed. In this variant, we have assumed that the scoring function is *monotonic*, i.e., that if t' is a compressed representation of t, then the score of t' is no greater than the score of t. This is certainly true of the dot product. Monotonic scoring functions imply that it is to our advantage to retain as many attributes as possible (constrained by the budget, m). However, not all scoring functions are monotonic, as we have seen in Section 5, when we discuss an application in text databases. In such case, the optimal number of attributes to retain may even be less than the budget m.

Finally, *Tk-GR* and *Tk-QR* also have per-attribute versions which can be naturally defined.

7.2.3 Complexity Results for *Tk-GR* and *Tk-QR* : *Tk-GR* can also be shown to be NP-hard by reducing from a *CB-D* problem instance as follows. We create a database with only one competing tuple t_1 with all 1's and with score(t_1) = 1. Let t be the new tuple with all 1's and with score(t) = 2. Let k = 1 for each query in the query log. The task in *Tk-GR* is to determine the top-m attributes such that the compressed tuple t' is returned as the top-1 tuple of as many queries as possible; clearly this is equivalent to solving the *CB-D* problem instance.

The same idea can also be used to show that *Tk-QR* is NP-hard for some scoring functions such as the dot product; e.g., it is easy to see that t' ties with t_1 for the top-1 tuple of a query only if it satisfies each attribute of the query.

7.2.4 Algorithms for *Tk-GR* and *Tk-QR* : Fortunately, due to the scoring function being a global function, we can reduce *Tk-GR* to *CB-QL* as follows, and use any of the algorithms developed in Section 4. We first execute each query q in Q. Let the k^{th} largest score of the returned tuples be s_q. Let the score of the new tuple be s_t. Let Q' be the subset of Q such that for each query q in Q', s_q is no larger than s_t. Then it is easy to see that all we need to solve *CB-QL* for the reduced query log Q', because if the compressed tuple t' satisfies a query q in Q', it will be definitely retuned as part of the top-k tuples of q.

However, in this approach we cannot leverage any pre-processing opportunities if we use the maximal frequent itemset based approach, as the Boolean table Q' has to be constructed at *runtime*. Thus the maximum frequent itemsets have to be computed at runtime.

Tk-QR is identical to *Tk-GR*, except that the scoring function is query specific and the semantics are not conjunctive. Unfortunately, this makes all the difference, and the frequent itemset approach is no longer applicable, as there appears to be no way of reducing this variant to *CB-QL*. Likewise, it appears difficult to formulate the problem naturally as a small integer linear program. The best we can do is to formulate the problem as a general (i.e., non-linear) Integer Program. The details are omitted due to their small practical value.

However, we can develop several effective greedy algorithms for *Tk-QR*, depending on whether we consume attributes or queries cumulatively or non-cumulatively (as Section 4.4). The straightforward details are omitted.

7.3 Skyline Boolean (*SB*)

We consider skyline retrieval semantics for this problem. Given a set of points, the skyline comprises the points that are not dominated by other points. A point dominates another point if it is as good or better in all dimensions and better in at least one dimension [31]. We consider skyline for Boolean data in our problem.

For each query q in the query log we define the *query skyline S(q)* = {$s_1 \ldots s_L$}, which is a collection of *skyline points*. Each skyline point s defines a subset (i.e., projection) of attributes for which any data point (tuple) remains on the skyline. For example, suppose a user poses a query q = "*Select * from Cars where Make = Honda and AC = yes and Power Windows = yes*", and the database has three cars t_1 = <*Toyota*, *AC*, *Power Windows*>, t_2 = <*Honda*, *AC*, *Power Brakes*> and t_3 = <*Nissan*, *AC*, *Power Brakes*>. We can see from the skyline definition that the cars t_1 and t_2 will be on the skyline of q, since they are not dominated by any other cars (t_3 here) present in the database based on the attributes asked by the query q. We do not store the actual skyline data points (all attributes present in the tuple) such as t_1 and t_2 in skyline log, instead the set of attributes for which a data point is visible on the skyline. Here, t_1 = <*Toyota*, *AC*, *Power Windows*> is visible on the skyline of q because of attributes {*AC*, *Power Windows*} asked by q. So,

the skyline points are s_1 = {*AC, Power Windows*} and s_2 = {*Honda*, *AC*} for which t_2 is on the skyline of q. A skylines log contains all the skylines for the query log.

7.3.1 Problem Definition (*SB*): *Given a database of competing products D, a query log Q with Skyline Query semantics, a new tuple t, and an integer m, compute a compressed tuple t' by retaining m attributes such that the number of queries for which t' appears on their skylines is maximized.*

Table 1 displays the skylines log for the query log Q and database D of Fig 1. Note that in Fig 1, none of the tuples in D satisfies q_5 (*Turbo*, *Auto Trans*) completely. In contrast to the conjunctive query semantics, this does not mean that q_5 has no answer. A tuple satisfies q_5 if it has attribute *Turbo* (t_2 and t_7) or Auto Trans (t_3), as shown in Table 1.

A query can have more than one set of attributes for which data points can be visible on the skyline; e.g., for query q_5, tuples t_2 and t_7 are visible on the skyline for attribute *Turbo*, whereas tuple t_3 is visible on the skyline for the attribute *Auto Trans*. We keep separate record for each set of attribute as shown in Table 1.

EXAMPLE 2: *To illustrate the example consider the skylines in Table 1. Assume we are required to retain m = 3 attributes of the new tuple. It is not hard to see that if we retain the attributes AC, Four Door, and Power Doors (i.e., t' = {AC, Four Door, Power Doors}), the compressed tuple t' will be visible on the skylines for the maximum of three queries (q_1, q_2, and q_3). No other selection of three attributes of the new tuple will remain on skylines of more queries.* □

TABLE 1
SKYLINES OF QUERIES

Skyline ID	*Query ID*	*Car ID*	*Attributes for which the car is on the skyline*
s_1	q_1	t_4, t_5	AC, Four Door
s_2	q_2	t_3, t_4	AC, Power Doors
s_3	q_3	t_1, t_4, t_6	Four Door, Power Doors
s_4	q_4	t_3, t_4	Power Brakes, Power Doors
s_5	q_5	t_2, t_7	Turbo
s_5	q_5	t_3	Auto Trans

7.3.2 Complexity Results for *SB*: *SB* is NP-hard since *CB-QL* can be reduced to it if there is a data point that completely satisfies the query (identical to the query), for each query in the query log.

7.3.3 Algorithms for *SB*: There are several methods proposed for efficient processing of skyline queries which are mentioned in related work (Section 8). Any good skyline processing technique such as [22] can be used here to find the skylines for the query log which is efficient for Boolean data. Once these skylines have been found, then our problem is to find the subset of the attributes for the new tuple so that skylines from the maximum number of queries will retrieve the new tuple. So, we can now revert back to conjunctive query semantics where a skyline s will retrieve the new tuple t if all the attributes present in the skyline is also present in t, i.e., $s \in t$, where t retains the selected subset of attributes (top-m attributes).

A new tuple will satisfy a skyline query if the tuple is a superset of a skyline point of the query skyline, that is, the new tuple contains all the attributes of a skyline point. Consider Table 1. If the new tuple has the attributes *AC*, *Four Door*, and *Power Doors* (i.e., t' = {*AC*, *Four Door*, *Power Doors*}), the compressed tuple t' will be visible on the skylines s_1, s_2, and s_3. We need to make sure that we do not just maximize the number of skyline points that t dominates, but maximize the number of queries for which t will be visible on their skylines.

We use *algorithm MaxFreqItemSets* used for the problem *CB-QL* with couple of updates: (1) we use skyline log instead of query log, and (2) we count each query only once rather than each skyline. Considering our running example, when we check if an itemset is frequent or not, we count each query only once regardless of the number of skyline points it has. For example, if we find two skylines points (for q_5) are present when we check an itemset is frequent or not, we only increase the count by one because both come from the same skyline, that of query q_5.

7.4 Conjunctive Boolean - Query Log - Negation (*CB-QL-Negation*)

Sometimes a query can have negation that means a user can specify in the query that he or she does not want specific attribute (i.e., that attribute should not be present in the product). For this problem variant we consider *Conjunctive Boolean Retrieval with Negation* retrieval semantics.

Conjunctive Boolean Retrieval with Negation: This problem variant also considers each query as conjunctive query where a tuple t satisfies a query q if q is a subset of t. However, the query can have negation. For example, a query such as {a_1, a_3, ~a_4} equivalent to "return all tuples such that a_1= 1 and a_3 = 1 and a_4 != 1 (more specifically a_4 must not be present). The set of returned tuples $R(q)$ is the set of all tuples that satisfy q. We assume that if an attribute value a_i is missing in t', but a_i is 0 in t (that is, this feature is missing), then a query q that specifies 1 for a_i is not satisfied by t'. That is, we assume that a user will eventually check all the attributes of the new product t.

7.4.1 Problem Definition (*CB-QL-Negation*): *Given a query log Q where a query can have negation with Conjunctive Boolean Retrieval semantics, a new tuple t, and an integer m, compute a compressed tuple t' by retaining m attributes such that the number of queries that retrieve t' is maximized.*

A query can have value for a Boolean attribute as 1, 0, or -1; where 1 means the attribute must be present, 0 means do not care, and -1 means must not be present. As in *CB-QL*, for buyers interested in browsing products of interest, we wish to ensure that the compressed version of the new product is visible to as many buyers as possible.

7.4.2 Complexity Results for *CB-QL-Negation*: *CB-QL-*Negations is also NP-hard, since *CB-QL* can be reduced to *CB-QL-Negations* if the queries have no negation.

Here we can define another complementary problem where we have access to the database of existing products but do not have access to the query log. This can be done similarly as *CB-QL* is used to define *CB-D* previously.

7.4.3 Algorithms for *CB-QL-Negation*: Direct application

of the algorithms for *CB-QL* does not work for this problem variant. This is because the query log has negations where a query asks that an attribute must not be present in the returned tuple. This breaks the monotonicity property of *CB-QL*, that is, adding more attribute to the new tuple does not always increase the number of satisfied queries. For this problem variant we use the algorithms for *CB-QL* with some preprocessing as follows:

1. *Remove all queries from the query log that do not satisfy a negated new tuple attribute (attribute with value 0).* At this step we remove these queries because the new tuple will never satisfy them.
2. *For each attribute not present in the new tuple, for each query in the query log that has -1 for this attribute, we change the value to 0.* Note that the new tuple has value 0 or 1 for each attribute, where recall that 0 denotes that the attribute is missing. Hence, if a query does not request an attribute which is not present in the new tuple, then value can updated to 0 in the query log.
3. *Then apply algorithms for CB-QL.* We can apply the algorithms for *CB-QL* now as negation is already removed.

7.5 Maximize Query Coverage (*MQC*)

This problem variant is interesting because it is the only one among the variants considered in this paper to have polynomial algorithm. The intuition is that we look for a compressed tuple that has maximum sum of scores over all queries in query log. For instance, find the attributes of a home so that it satisfies as many of the conditions of the past queries as possible. The reason why this problem is polynomial is that it is a best-effort problem. We assume that the scoring function is an aggregation of the scores of the individual attributes, e.g., the sum of the attribute contributions. The attribute contribution could be 1 if it is satisfied or 0 otherwise. For a text database, it could be the tf-idf weight of a keyword.

7.5.1 Problem Definition (*MQC*): *Given a query log Q, and a new tuple t, find compressed tuple t′ that maximizes the sum of scores of t′ over all queries in Q.*

7.5.2 Complexity Results and Algorithms for *MQC*: An optimal polynomial algorithm is the following. At each iteration, select the attribute of t that maximizes the sum of the scores for all queries in Q, assuming that the rest of the attribute values are missing.

7.6 Categorical and Numeric Data

7.6.1 Problems and Algorithms for *Categorical Data*

We also consider *categorical databases*, which are natural extensions of Boolean databases where each attribute a_i can take one of several values from a multi-valued categorical domain Dom_i. A query over a categorical database is a set of conditions of the form $a_i = x_i$, $x_i \in Dom_i$. We can define problem variants for categorical data corresponding to the ones for Boolean data discussed earlier.

Each categorical column a_i can be replaced by $|Dom_i|$ Boolean columns, and consequently a categorical database/query log with M attributes is replaced by a Boolean database/query log with $\prod_{1 \le i \le M} Dom_i$ Boolean attributes.

7.6.2 Problems and Algorithms for *Numeric Data*

Finally we also consider *numeric databases*. We consider queries that specify ranges over a subset of attributes. The above problem variants for Boolean data have corresponding versions for numeric databases. For example, users browsing for used digital cameras may specify queries with ranges on price, age of product, desired resolution, etc, and the returned results may be ranked by price.

Problems involving numeric ranges in queries and global scoring functions can be reduced to Boolean problem instances as follows. We first execute each query in the query log, and reduce Q to Q' by eliminating queries for which the new tuple has no chance of entering into the top-k results, exactly as we did in Section 7.2. Then, for each numeric attribute a_i in Q', we replace it by a Boolean attribute b_i as follows: if the i^{th} range condition of query q contains the i^{th} value of tuple t, then assign 1 to b_i for query q, else assign 0 to b_i for query q. I.e., each query has effectively been reduced to a Boolean row in a Boolean query log Q'. The tuple t can be converted to a Boolean tuple consisting of all 1's. It is not hard to see we have created *CB-QL* variant for Boolean data, whose solution will solve the corresponding problem for numeric data.

However, if we choose to solve this problem using our frequent itemset based approach, it is important to note that we cannot leverage any preprocessing opportunities, as the Boolean query log has to be constructed at runtime.

8 Related Work

A large corpus of work has tackled the problem of ranking the results of a query. In the documents world, the most popular techniques are tf-idf based [28] ranking functions, like BM25 [26], as well as link-structure-based techniques like PageRank [6] if such links are present (e.g., the Web). In the database world, automatic ranking techniques for the results of structured queries have been proposed [1, 7, 30]. Also there has been recent work [8] on ordering the displayed attributes of query results.

Both of these tuple and attribute ranking techniques are inapplicable to our problem. The former inputs a database and a query, and outputs a list of database tuples according to a ranking function, and the latter inputs the list of database results and selects a set of attributes that "explain" these results. In contrast, our problem inputs a database, a query log, and a new tuple, and computes a set of attributes that will rank the tuple high for as many queries in the query log as possible.

Although the problem of choosing attributes is seemingly related to the area of feature selection [10], our work differs from the work on feature selection because our goal is very specific – to enable a tuple to be highly visible to the database users – and not to reduce the cost of building a mining model such as classification or clustering.

Kleinberg et al. [18] present a set of microeconomic problems suitable for data mining techniques; however no specific solutions are presented. Their problem closer to our work is identifying the best parameters for a marketing strategy in order to maximize the attracted customers, given that the competitor independently also prepares a

similar strategy. Our problem is different since we know the competition. Another area where boosting an item's rank has received attention is Web search, where the most popular techniques involve manipulating the link-structure of the Web to achieve higher visibility [13].

Integer and linear programming optimization problems are extremely well studied problems in operations research, management science and many other areas of applicability (see recent book on this subject [27]). Integer programming is well-known to be NP-hard [11]; however carefully designed branch and bound algorithms can efficiently solve problems of moderate size. In our experiments, we use an of-the-shelf ILP solver available from http://lpsolve.sourceforge.net/5.5/download.htm.

Computing frequent itemsets is a popular area of research in data mining and some of the best known algorithms include Apriori [2] and FP-Tree [15]. Several papers have also investigated the problem of computing maximal frequent itemsets [3, 5, 12, 14, 17]. Almost all the popular approaches are designed for sparse datasets and do not work well for our unique problem of dense datasets. Apriori [2] employs a bottom-up, breadth first search that enumerates every single frequent itemset. In many applications (especially in dense data) with long frequent patterns enumerating all possible subsets of an M length pattern (M can easily be 50 or 60 or longer) is computationally unfeasible. Also, we are not interested in mining all frequent itemsets, but only maximal frequent itemsets in our algorithm. A known approach for mining maximal frequent itemsets is the complete random walk [12], which is a bottom-up approach. But in a dense dataset the maximal frequent itemsets usually lie on the top region of the lattice, and if a bottom-up approach is used to find maximal frequent itemsets, it will have to traverse a long portion of the lattice (i.e., numerous levels) and will be inefficient. To see this, consider a table with 50 attributes, and assume we need to determine a compressed tuple t' with 10 attributes. Now, we need to know the itemset of $\sim Q$ (complemented query log which is a dense dataset) of size 40 with maximum frequency. Due to the dense nature of $\sim Q$, the bottom-up approach will not be able to compute frequent itemsets beyond a size of 5-10. Likewise, other approaches for mining maximal frequent itemsets such as the Genetic Algorithm (GA) based approach [17] is also mainly intended for sparse dataset and does not work well for dense dataset. In contrast, our proposed method works well for dense dataset.

The recent works [21] and [20] are related to our work. The former tries to find out the dominant relationship between products and potential buyers where by analyzing such relationships, companies can position their products more effectively while remaining profitable. The latter introduces skyline query types taking into account not only min/max attributes (e.g., price, weight) but also spatial attributes and the relationships between these different attribute types. Their work aims at helping manufacturers choose the right specs for a new product, whereas our work to choose the attributes subset of an existing product for advertising purposes.

In previous work [23], we tackled the main variant of the problem with Boolean conjunctive query semantics where a tuple satisfies a query if all the attributes present in query are also present in the tuple (Section 3). We extend the idea in the current paper. We consider both the database (existing products) and query log with various query semantics (conjunctive, top-k, skyline, negations, etc.).

Several techniques have been proposed for efficient skyline query processing [4, 19, 25, 31]. There has been recent work on categorical skylines [29] and skyline computation over low cardinality domains [22] that also considers skyline for Boolean data as well. One main difference of our work with the existing works is that our goal is not to propose a method for processing or maintaining the skylines, instead we use skylines as a query semantic where a new tuple can be visible for maximum number of queries.

Another related work is mining top-k frequent itemsets without minimum support threshold [16] which finds top-k closed frequent itemsets. This is inapplicable in our case because we are interested in finding out all the maximal frequent itemsets, and not just the top-k frequent itemsets. Also it is not proven that the top-k approach works well for dense dataset. The top-k approach without minimum support threshold [16] finds top-k frequent closed patterns of length no less than min_l, where min_l is the minimal length of each pattern. In our problem, we do not have any min-l restriction.

9 Conclusions

In this work we introduced the problem of selecting the best attributes of a new tuple, such that this tuple will be ranked highly, given a dataset, a query log, or both, i.e., the tuple "stands out in the crowd". We presented variants of the problem for Boolean, categorical, text and numeric data, and showed that even though the problem is NP-complete in most cases; optimal algorithms are feasible for small inputs. Furthermore, we present greedy algorithms, which are experimentally shown to produce good approximation ratios.

While the problems considered in this paper are novel and important to the area of ad-hoc data exploration and retrieval, we observe that our specific problem definition does have limitations. After all, a query log is only an approximate surrogate of real user preferences, and moreover in some applications neither the database, nor the query log may be available for analysis; thus we have to make assumptions about the nature of the competition as well as about the user preferences. Finally, in all these problems our focus is on deciding what subset of attributes to retain of a product. We do not attempt to suggest what values to set for specific attributes, which is a problem tackled in marketing research, e.g., [24].

However, while we acknowledge that the scope of our problem definition is indeed limited in several ways, we do feel that our work takes an important first step towards developing principled approaches for attribute selection in a data exploration environment.

Acknowledgment

We thank the anonymous referees for their useful com-

ments on earlier drafts of this paper. Vagelis Hristidis was supported by NSF grants IIS-0811922 and IIS-0534530. Gautam Das was partially supported by NSF grants IIS-0845644 and NSF IIS-0812601 as well as gifts from Microsoft Research and Nokia Research.

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, Gautam Das, Aristides Gionis: Automated Ranking of Database Query Results. CIDR 2003.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, (eds.), Advances in Knowledge Discovery and Data Mining, pp. 307-328. AAAI/MIT Press, 1996.

[3] Roberto J. Bayardo Jr.: Efficiently Mining Long Patterns from Databases. SIGMOD Conference 1998: 85-93. 1

[4] S. Borzsonyi, D. Kossmann, K. Stocker: The Skyline Operator. ICDE '01.

[5] D. Burdick, M. Calimlim, J. Gehrke: MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases. ICDE 2001

[6] S. Brin and L. Page: The Anatomy of a Large-Scale Hypertextual Web Search Engine. WWW Conference, 1998.

[7] S. Chaudhuri, G. Das, V. Hristidis, G. Weikum: Probabilistic Ranking of Database Query Results. VLDB, 2004.

[8] Gautam Das, Vagelis Hristidis, Nishant Kapoor, S. Sudarshan. Ordering the Attributes of Query Results. SIGMOD, 2006.

[9] Good, I., The population frequencies of species and the estimation of population parameters, Biometrika, v. 40, 1953, pp. 237-264.

[10] Isabelle Guyon and Andre Elisseeff. An introduction to variable and feature selection. Journal of Machine Learning Research, 3(mar): 2003.

[11] Michael R. Garey and David S. Johnson (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman. ISBN 0-7167-1045-5.

[12] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, R. S. Sharm: Discovering all most specific sentences. ACM TODS. 28(2): 2003

[13] M. Gori and I. Witten. The bubble of web visibility. Commun. ACM 48, 3 (Mar. 2005), 115-117.

[14] Karam Gouda, Mohammed J. Zaki: Efficiently Mining Maximal Frequent Itemsets, ICDM 2001.

[15] Jiawei Han, Jian Pei, Yiwen Yin: Mining Frequent Patterns without Candidate Generation. SIGMOD 2000: 1-12.

[16] Jiawei Han, Jianyong Wang, Ying Lu, Petre Tzvetkov: Mining top-k frequent closed patterns without minimum support, ICDM 2002.

[17] Jen-peng Huang, Che-Tsung Yang, Chih-Hsiung Fu: A Genetic Algorithm Based Searching of Maximal Frequent Itemsets. ICAI 2004.

[18] J. Kleinberg, C. Papadimitriou and P. Raghavan.A Microeconomic View of Data Mining. Data Min. Knowl. Discov. 2, 4 (Dec. 1998).

[19] D. Kossmann, F. Ramsak, S. Rost: Shooting Stars in the Sky: an Online Algorithm for Skyline Queries. VLDB 2002.

[20] Cuiping Li, Beng Chin Ooi, Anthony K. H. Tung, Shan Wang: DADA: a Data Cube for Dominant Relationship Analysis. SIGMOD 2006.

[21] Cuiping Li, Anthony K. H. Tung, Wen Jin, Martin Ester: On Dominating Your Neighborhood Profitably. VLDB 2007: 818-829

[22] Michael D. Morse, Jignesh M. Patel, H. V. Jagadish: Efficient Skyline Computation over Low-Cardinality Domains. VLDB 2007.

[23] Muhammed Miah, Gautam Das, Vagelis Hristidis, Heikki Mannila: Standing Out in a Crowd: Selecting Attributes for Maximum Visibility. ICDE 2008: 356-365

[24] Thomas T. Nagle, John Hogan. The Strategy and Tactics of Pricing: A Guide to Growing More Profitably (4th Edition), Prentice Hall, 2005.

[25] Dimitris Papadias, Yufei Tao, Greg Fu, Bernhard Seeger: An Optimal and Progressive Algorithm for Skyline Queries. ACM SIGMOD 2003.

[26] S E Robertson and S Walker. Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval. SIGIR 1994.

[27] Alexander Schrijver: Theory of Linear and Integer Programming. John Wiley and Sons. 1998.

[28] G. Salton. Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer. Addison Wesley, 1989.

[29] Nikos Sarkas, Gautam Das, Nick Koudas, Anthony K. H. Tung: Categorical skylines for streaming data. SIGMOD Conference 2008: 239-250

[30] W. Su, J. Wang, Q. Huang, F. Lochovsky. Query Result Ranking over E-commerce Web Databases. ACM CIKM 2006.

[31] Kian-Lee Tan, Pin-Kwang Eng, Beng Chin Ooi: Efficient Progressive Skyline Computation. VLDB 2001.

Muhammed Miah is a PhD student of Computer Science and Engineering department at the University of Texas at Arlington. He received BS degree in Civil Engineering from Khulna University of Engineering and Technology, Bangladesh, MS degree in Computer and Information Science from University of New Haven, Connecticut, and MBA degree from Qunnipiac University, Connecticut. His main research work includes data mining, information retrieval, and maximizing visibility of products in search queries.

Gautam Das received the PhD degree in computer science from the University of Wisconsin, Madison, in 1990. He has been an associate professor in the Computer Science and Engineering Department, University of Texas, Arlington, since 2004. His research interests include data mining and knowledge discovery, databases, algorithms, and computational geometry. His research has been supported by the US National Science Foundation (NSF), US Office of Naval Research (ONR), and indictries including Microsoft, Nokia and Cadence.

Vagelis Hristidis received the BS degree in Electrical and Computer Engineering from the National Technical University of Athens and the MS and PhD degrees in Computer Science from the University of California, San Diego, in 2004. Since then, he is an Assistant Professor in the School of Computing and Information Sciences at Florida International University, in Miami. His main research work addresses the problem of bridging the gap between databases and information retrieval.

Heikki Mannila is the director of Helsinki Institute for Information Technology HIIT, a joint research institute of University of Helsinki and Helsinki University of Technology TKK, and a professor of computer science at TKK. He has also worked at Technical University of Vienna, Max Planck Institute for Computer Science, Microsoft Research, and Nokia Research Center. He has published two books and over 190 refereed articles in computer science and related areas. His specific area of interest is in algorithms for data analysis, and applications in science and in industry. He received the ACM SIGKDD Innovation award in 2003.