Standing Out in a Crowd: Selecting Attributes for Maximum Visibility

Muhammed Miah[#1], Gautam Das[#2], Vagelis Hristidis[*], Heikki Mannila[+]

[#]*Department of Computer Science and Engineering, University of Texas at Arlington*
416 Yates Street, Arlington, TX 76019, USA
[1]mzmiah@uta.edu
[2]gdas@uta.edu

[*]*School of Computing and Information Sciences, Florida International University*
11200 S.W. 8th Street, Miami, FL 33199, USA
vagelis@cis.fiu.edu

[+]*HIIT, Helsinki University of Technology and University of Helsinki*
P.O Box 68, FI 00014, University of Helsinki, Helsinki, Finland
Heikki.Mannila@tkk.fi

***Abstract*— In recent years, there has been significant interest in 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). In this paper we focus on a novel and complementary problem: how to guide a seller in selecting the best attributes of a new tuple (e.g., new product) to highlight such that it stands out in the crowd of existing competitive products and is widely visible to the pool of potential buyers. We develop several interesting formulations of this problem. Although these problems are NP-complete, we can give several exact algorithms as well as approximation heuristics that work well in practice. Our exact algorithms are based on Integer Programming (IP) formulations of the problems, as well as on adaptations of maximal frequent itemset mining algorithms, while our approximation algorithms are based on greedy heuristics. We conduct a performance study illustrating the benefits of our methods on real as well as synthetic data.**

I. 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 (*Boolean retrieval*), or may rank and return the top-*k* data objects using suitable scoring functions (*Top-k retrieval*).

However, the focus of this paper in *not* on new search and retrieval techniques that will aid users in effective exploration of such databases. Rather, the focus is on a complementary yet novel problem of data exploration, as described below.

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, *as well as* users who insert new objects into these databases in the hope that they will be easily discovered by the first types of users. For example, in a database representing an e-marketplace (such as Craigslist.com, 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* or *manufacturers* of products – where products could range from automobiles to phones to rental apartments to job advertisements. We note that almost all of the prior research efforts on effective search and retrieval techniques – such as new top-*k* algorithms, new ranking functions, and so on – have been designed with the first kind of user in mind (i.e., the buyer). In contrast, relatively less research has been expended towards developing 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 real-world 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 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"). 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. Consider a new product t that needs to be inserted into this database. While the product has numerous attributes, due to budget constraints there is a limit, say m, on the number of attributes 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 a thorough investigation of this novel optimization problem. We mainly focus on an important variant where the data is Boolean and the queries follow conjunctive retrieval semantics. We also briefly consider several important variants, including Boolean data with other types of retrieval semantics (disjunctive as well as top-k retrieval), as well as categorical, numeric and text data. Our main contributions are summarized below:

1. We introduce the problem of *selecting attributes of a tuple for maximum visibility* as a new data exploration problem that benefits a certain class of users interested in designing and marketing their products. We consider several interesting variants of the problem as well as diverse application scenarios.
2. We show that the problem is NP-complete.
3. We give exact Integer Programming (IP) and Integer Linear Programming (ILP)-based algorithms to solve the problem. These algorithms are effective for moderate-sized problem instances.
4. 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 present fast greedy approximation algorithms that work well in practice.
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 II we give a formal definition of our problem, including several interesting problem variants. Section III analyzes the computational complexity of the problem, and Section IV presents various optimal algorithms as well as heuristics. In Section V we extend these techniques to also work for other problem variants. In Section VI we discuss related work, and present the result of extensive experiments in Section VII. Section VIII is a short conclusion.

II. Problem Framework

In this section we first develop, with a detailed example, a formal definition of the problem for the case of Boolean data and a query log of conjunctive Boolean queries. We then briefly define other Boolean variants, as well as extensions for other kinds of data such as categorical, text and numeric data. Due to space constraints, throughout the paper we primarily focus on the first Boolean variant, and restrict our discussion of other variants to briefly outlining the extensions necessary.

A. A Boolean Problem Variant.

Some useful definitions and notations are given here.

***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: We view each query as a subset of attributes. The retrieval semantics is *Conjunctive Boolean Retrieval*, e.g., 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 is denoted as $R(q)$.

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

We are now ready to formally define the main problem variant considered in the paper.

> **PROBLEM *SOC-CB-QL*** *(or "Stand Out in a Crowd-Conjunctive Boolean-Query Log"): Given a query log Q 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.*

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. The following running example will be used to illustrate Problem ***SOC-CB-QL*** (as well as other variants later in the paper).

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

EXAMPLE 1: *Consider an inventory database of an auto dealer, which contains a single database table D with N=7 rows and M=6 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, etc; categorical attributes such as Make, Color, etc; numeric attributes such as Price, Age, 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.* □

Suppose 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., $t' = [1, 1, 0, 1, 0, 0]$), 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. Notice that in *SOC-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. That is, there is no need of access to the database.

B. Other Problem Variants.

Here we define several other variants of the problem that are useful in different applications. Problem *SOC-CB-QL* 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/manufacturing the new product, this problem maximizes the number of potential buyers per unit cost.

A complementary variant to *SOC-CB-QL* is *SOC-CB-D* (or "Stand Out in a Crowd-Conjunctive Boolean–Data"): given a database D, a new tuple t, and an integer m, we wish to compute a compressed tuple t' by retaining m attributes such that the number of tuples in D *dominated* by t' is maximized. This is useful in scenarios where we have access to the database, but do not have access to the query log. To illustrate this 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. It is also easy to see that any algorithm that solves *SOC-CB-QL* can be also used to solve *SOC-CB-D*, by replacing the query log with the database as input. *SOC-CB-D* also has a natural per-attribute version.

A more general problem variant arises when the retrieval semantics is not simply conjunctive Boolean; e.g., the retrieval semantics could be *disjunctive Boolean* or even *Top-k retrieval*. To understand the latter, 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. The problem variant *SOC-Topk* is defined as: given a database D, a query log Q with top-k retrieval semantics via a 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. Solving this problem requires access to both the query log as well as the database.

The above problems are not restricted only to Boolean databases. C*ategorical databases* 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. Problem variants similar to Boolean can be defined for categorical databases. Furthermore, *text databases* consist 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 scoring functions such as the tf-idf-based BM25 scoring function [19]. *SOC-Topk* 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. Finally, the above problems can be extended to *numeric* databases, i.e., databases with numeric attributes, where queries specify ranges over a subset of attributes. For example, users browsing a database for digital cameras may specify desired ranges on price, weight, resolution, etc, and the returned results may be ranked by price.

In the rest of the paper we primarily focus on analyzing the complexity and developing solutions for our main variant *SOC-CB-QL* defined in Section II.A; due to lack of space our discussion of the other variants is restricted to brief outlining the extensions necessary of our proposed approaches.

III. Complexity Results

In this section we show that our main problem variant is NP-complete.

Theorem 1: *SOC-CB-QL is NP-complete*.

Proof (sketch): The membership of the decision version of the problem in NP is obvious. To see NP-hardness, we reduce the Clique problem to *SOC-CB-QL*. 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 *SOC-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 $m = r$. It is straightforward to verify that t has a compressed representation with m attributes that satisfies m(m-1)/2 queries if and only if the graph G has a clique of size r. □

It is not hard to show that the same proof can be extended to show that all the other variants described in Section II.B are also NP-hard. We omit further details due to lack of space.

IV. Algorithms for SOC-CB-QL

In this section we discuss our main algorithmic results. We restrict our discussion to Problem SOC-CB-QL, and defer discussion of other problem variants to Section V.

A. Optimal Brute Force Algorithm.

Clearly, since *SOC-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 *BruteForce-SOC-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.

B. Optimal Algorithm Based on ILP.

We next show how *SOC-CB-QL* can be described in an integer programming (ILP) framework. Let the new tuple be the Boolean vector $t = \{a_1(t),..., a_M(t)\}$, 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:

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

It is easy to see that the maximum gives exactly the solution to *SOC-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.

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

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

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-SOC-CB-QL*. The integer *linear* formulation is particularly attractive as unlike more general IP solvers, ILP solvers are usually more efficient in practice.

C. 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-SOC-CB-QL*, is based on an interesting adaptation of an algorithm for mining

Maximal Frequent Itemsets [11]. We first define the frequent itemset problem:

The Frequent Itemset Problem. *Let R be a 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, 14]. 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 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.

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 $\sim t$ ($\sim q$) denote the complement of a tuple t (query q), i.e., where the 1's and 0's have been interchanged. Likewise let $\sim Q$ denote the complement of a query log Q where the each query has been complemented. Now, $freq(\sim t)$ can be defined as the number of rows in $\sim Q$ that support $\sim t$.

The rest of the approach is now seemingly clear: compute all frequent itemsets of $\sim Q$ (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 $\sim t$ with the highest frequency. The optimal compressed tuple t' is therefore the complement of I, i.e., $\sim I$.

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 $\sim Q$ is an extremely dense table, and this prevents most frequent itemset algorithms from being directly applicable to $\sim Q$. 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 $\sim Q$ of size 40 with maximum frequency. Due to the dense nature of $\sim Q$, 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 [14] from being effective.

Setting of the Threshold Parameter. Let us assume we can solve the itemset mining problem of extremely dense datasets. What should be the setting of the threshold? Clearly setting the threshold $r=1$ will solve *SOC-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. For example, setting the threshold as 1% of the query log size implies that we are attempting to compress t such that at least 1% of the queries are still able to retrieve the tuple. 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. Otherwise, the algorithm will return empty.

The alternate adaptive procedure of setting the threshold is, first initialize the threshold to a high value and compute the frequent itemsets of $\sim Q$. If there are no frequent itemsets of size at least $M - m$ that are supersets of $\sim t$, repeat the process with a smaller threshold (say half of the previous threshold). This process is guaranteed to discover the optimal t'.

Random Walk to Compute Maximal Frequent Itemsets. We now return back to the task of how to compute frequent itemsets of the dense Boolean table $\sim Q$. In fact, we do not compute all frequent itemsets of the dense table $\sim Q$, 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 $\sim Q$. 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 $\sim Q$. 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.

There exist several algorithms for computing maximal frequent itemsets, e.g. [3, 4, 13, 11]. Let us consider the *random walk* based algorithm in [11], 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 have 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 discover all maximal frequent itemsets with high probability. However, since this algorithm is based on traversing the lattice from bottom to top, the random walk will have to traverse a lot of

levels before it reaches a maximal frequent itemset of a dense table.

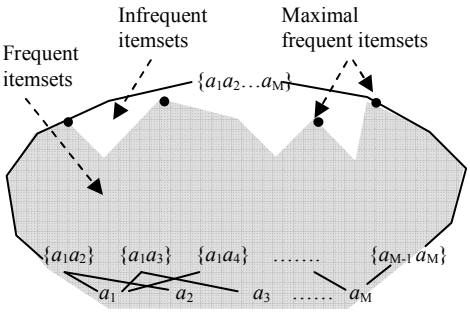

Fig 2 Maximal frequent itemsets in a Boolean Lattice

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 ($I = \{a_1a_2\ldots a_M\}$), 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, resulting in a maximal frequent itemset I.

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.

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 [8].

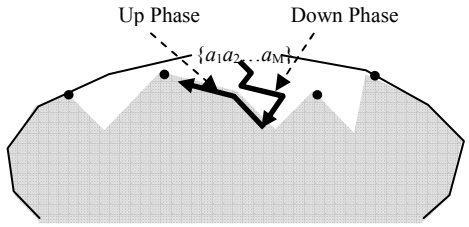

Fig 3 Two phase random walk

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., $\sim I$.

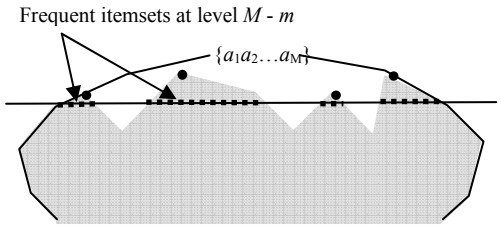

Fig 4 Checking frequent itemsets at level $M - m$

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

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 pre-computed, and the only task that needs to be done at runtime would be 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 simply lookup the itemset with the highest frequency at runtime.

D. Greedy Algorithms.

While the maximal frequent itemset based algorithm has much better scalability properties than the ILP based algorithm, it too becomes prohibitive for really large datasets (query logs). Consequently, we also developed three (suboptimal) greedy heuristics for solving *SOC-CB-QL*, some of whom worked very well in our experiments.

The greedy algorithm *ConsumeAttr-SOC-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 greedy algorithm *ConsumeAttrCumul-SOC-CB-QL* is a cumulative version of the previous algorithm. 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. Finally, instead of consuming attributes greedily, an alternative approach is to consume queries greedily. The greedy algorithm *ConsumeQueries-SOC-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.

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*(~Q, r)
  execute Down Phase random walk
  execute Up Phase random walk
  return itemset reached after Up Phase

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

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

Fig 5 Algorithm MaxFreqItemSets-SOC-CB-QL

V. Algorithms for other problem variants

In this section we briefly outline how the algorithms developed in Section IV can be extended to solve the problem variants defined in Section II.B.

The per-attribute variant of *SOC-CB-QL* can be simply solved by trying out values of m between 1 and M (recall that M is the total number of attributes of the database) and making M calls to any of the algorithms discussed in Section IV, and selecting the solution that maximizes our objective.

SOC-CB-D can be solved using any algorithm for *SOC-CB-QL* by replacing the query log with the database.

For the *SOC-Topk* problem, since the retrieval semantics is not always conjunctive Boolean, the frequent itemset approach is not always applicable. In general, the problem can be formulated as a non-linear integer program. However, ILP formulations and frequent itemset mining approaches are possible in the case of *global* scoring functions – i.e., functions of the form *score*(t) which are dependent on the tuple, but not on the query. For example, a user may be interested in getting the top-10 cars with AC and Turbo, ordered by decreasing number of available features – here the scoring function is the number of attributes in each tuple that have value 1. Another example of a global scoring function is ordering by a numeric attribute such as Price. We omit further details of these formulations from this version of the paper. Alternatively, the greedy algorithms of Section IV.D can be extended to the *SOC-Topk* problem.

The case of categorical data is a straightforward generalization of Boolean data. Likewise, as discussed in Section II.B, text data can be treated as Boolean data, and in principle all the algorithms developed for Boolean data can be used for text data. However, 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, the greedy approaches are the only ones feasible in this scenario.

Finally, problems involving numeric databases and query logs of range queries can be reduced to Boolean problem instances as follows: We first execute each query in the query log, and for each numeric attribute a_i in Q, we replace it by a Boolean attribute b_i as follows: if the ith range condition of query q contains the ith 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 that we have created an instance of *SOC-CB-QL*, whose solution will solve the corresponding problem for numeric data.

VI. 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 [21] ranking functions, like BM25 [19], as well as link-structure-based techniques like PageRank [5] if such links are present (e.g., the Web). In the database world, automatic ranking techniques for the results of structured queries have been recently proposed [1, 6, 22]. In addition to ranking the results of a query, there has been recent work [7] on ordering the displayed attributes of query results.

Both of these tuple and the 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* [9], our work differs from the extensive body of work on feature selection because our goal is very specific – to enable a tuple to be highly visible to the users of the database – and not to reduce the cost of building a mining model such as classification or clustering.

Kleinberg at al. [15] 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 (other data items). 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 [12].

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 [20]). Integer programming is well-known to be NP-hard [10]; however carefully designed branch and bound algorithms can efficiently solve problems of moderate size. In our own 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 [14]. Several papers have also investigated the problem of computing maximal frequent itemsets [3, 4, 11, 13].

The recent works on dominant relationship analysis [16] and dominating neighborhood profitably [17] 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, and the latter introduces skyline query types taking into account not only min/max attributes (e.g., price, weight) but also spatial attributes (e.g., location 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 aims at choosing the attributes subset of an existing product for advertising purposes.

VII. EXPERIMENTS

In this section we measure (a) the time cost of the proposed optimal and greedy algorithms, and (b) the approximation quality of the greedy algorithms. Due to lack of space we have chosen to present the results for the most representative problem variant, *SOC-CB-QL*. Due to space constraints we omit experiments on text, categorical and numeric data problem variants as they are adaptations of the algorithms used for *SOC-CB-QL*, as described in Section V.

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. We implemented all algorithms in C#, and connected to the RDBMS through ADO.

Dataset: We use an *online used-cars dataset* consisting of 15,211 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 a synthetic workload of 2000 queries. 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%. That is, we assume that most of the users specify two or three attributes.

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 *SOC-CB-QL*. These are (Section IV.A): *ILP-SOC-CB-QL, MaxFreqItemSets-SOC-CB-QL,* which produce optimal results, and *ConsumeAttr-SOC-CB-QL, ConsumeAttrCumul-SOC-CB-QL,* and *ConsumeQueries-SOC-CB-QL*, which are greedy approximations. The *SOC-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 themselves. We note that the *MaxFreqItemSets* algorithm consistently performs better than the *ILP* algorithm. Another 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 IV.C. 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.

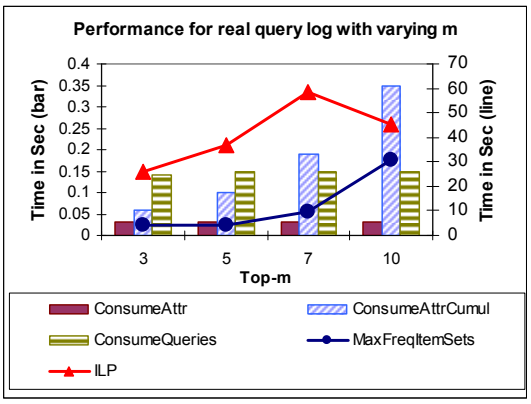

Fig 6 Execution times for SOC-CB-QL for varying *m*, for real workload.

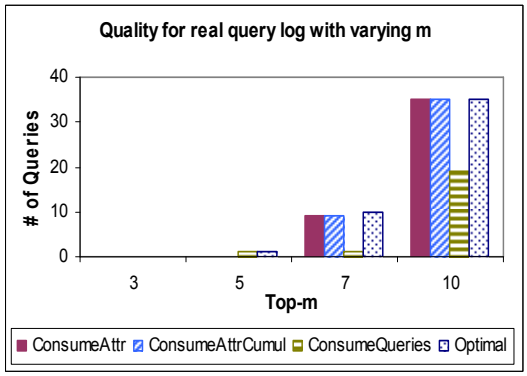

Fig 7 Satisfied queries for SOC-CB-QL for varying *m*, for real workload.

Fig 8 and Fig 9 repeat the same experiments for the synthetic query workload. 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).

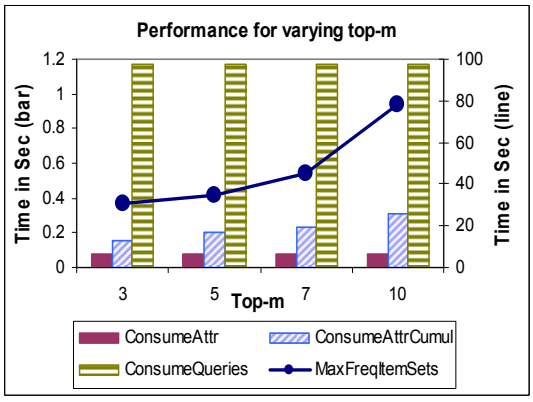

Fig 8 Execution times for SOC-CB-QL for varying *m*, for the synthetic workload of 2000 queries

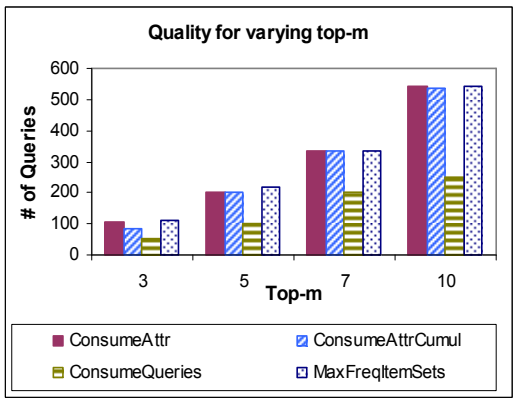

Fig 9 Satisfied queries for SOC-CB-QL for varying *m*, for synthetic workload of 2000 queries.

Next, we measure the execution times of the algorithms for varying query log size and number of attributes. The quality results for these experiments are not reported due to lack of space. 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. Combined with the fact that *ConsumeQueries* has inferior quality, we conclude that it is generally a bad choice.

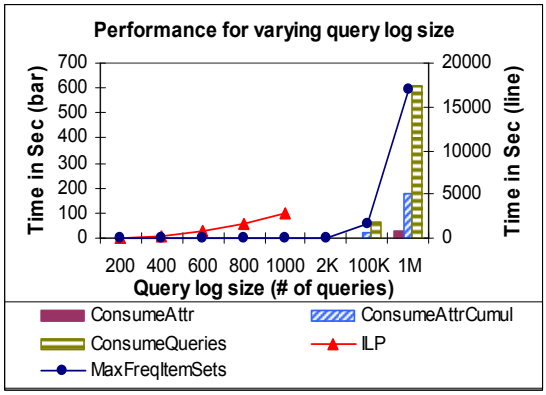

Fig 10 Execution times for SOC-CB-QL for varying query log size, varying synthetic workload size, m = 5

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 as well as wide, the problem becomes truly intractable, and approximation methods such as our greedy algorithms are perhaps the only feasible approaches.

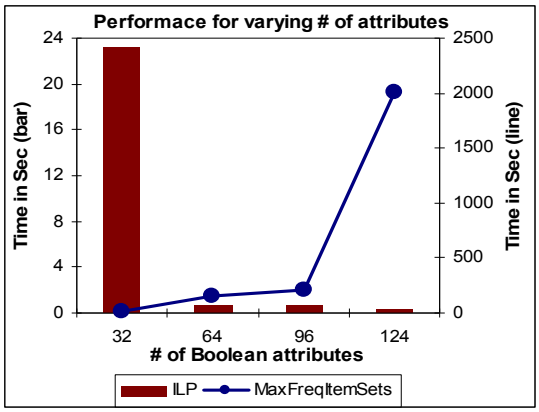

Fig 11 Execution times for SOC-CB-QL for varying M, synthetic workload of 200 queries, m = 5

VIII. 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, and showed that even though the problem is NP-complete, 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., [18].

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. Overcoming the limitations mentioned above is subject of future work.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers, Stefan Schoenauer, and Valentin Polishchuk for useful comments. The work of Vagelis Hristidis was partially supported by National Science Foundation Grant IIS-0534530. The work of Gautam Das and Muhammed Miah was partially supported by unrestricted gifts from Microsoft Research and start-up funds from the University of Texas, Arlington. The work of Heikki Mannila was partially supported by the Academy of Finland.

REFERENCES

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

[2] R. Agrawal, and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. VLDB, 1994.

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

[4] Douglas Burdick, Manuel Calimlim, Johannes Gehrke: MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases. ICDE 2001

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

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

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

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

[9] Isabelle Guyon and Andre Elisseeff. An introduction to variable and feature selection. Journal of Machine Learning Research, 3(mar):1157–1182, 2003.

[10] 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.

[11] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, R. S. Sharm: Discovering all most specific sentences. ACM TODS. 28(2): 140-174 (2003)

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

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

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

[15] J. Kleinberg, C. Papadimitriou and P. Raghavan.A Microeconomic View of Data Mining. Data Min. Knowl. Discov. 2, 4 (Dec. 1998), 311-324

[16] Cuiping Li, Beng Chin Ooi, Anthony K. H. Tung, Shan Wang: DADA: a Data Cube for Dominant Relationship Analysis. SIGMOD Conference 2006: 659-670

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

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

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

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

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

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