

An Optimal Signature-Tree based Algorithm for Selecting Attributes for Maximum Visibility

Muhammed Miah

University of Texas at Arlington, Arlington, TX, USA

E-mail: md.miah@mavs.uta.edu

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). We focus on a novel and complementary problem that has been recently considered in literature: 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. In this paper we investigate the problem, and provide novel method that guarantee optimal solutions where existing methods fail. Our method also much more efficient than existing methods for several important input data distributions. We conduct a performance study illustrating the benefits of our method.

Keywords

Data mining, knowledge and data engineering, marketing, mining methods and algorithms, e-commerce.

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

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

There are 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.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* 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 advertisement 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 the solution of the problem as an attempt to build a recommendation system for *sellers*, unlike the more traditional recommendation systems for buyers.

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, one might be interested in developing a catchy title, or selecting a few important indexing keywords, for a scientific article.

This problem was first considered in [11] and was shown to be NP-complete, and several solutions were proposed. In this current paper we investigate this problem for further improvement in quality as well as in performance. We propose solution based on novel adaptations of the *Signature Tree* data structure [4]. Our solution is based on smart pruning during creation and searching of the tree, which enables effective and practical solutions for several important input data distributions, thus greatly improving upon the performance over the prior methods considered in [11]. Note that the Signature Tree data structure was originally developed for a different context, i.e., for effective

indexing of Boolean signatures. We mainly focus on an important variant where the data is Boolean and the queries follow conjunctive retrieval semantics but our proposed method is also applicable for other problem variants considered in [11].

Our main contributions are summarized as follows:

1. We investigate the problem of selecting attributes of a product for maximum visibility that benefits a certain class of users interested in designing and marketing their products.
2. We develop scalable optimal solution based on novel adaptations of Signature Tree data structure.
3. We perform detailed performance evaluations to demonstrate the effectiveness of our developed algorithm and performance over existing approaches.

The rest of the paper is organized as follows. In Section 2 we give a formal definition of our problem and mention the complexity of the problem. Section 3 presents the proposed optimal algorithm. In Section 4 we mention related work, and present the result of experiments in Section 5. Section 6 is a short conclusion and Section 7 provides the references.

2. PROBLEM DEFINITION AND COMPLEXITY

2.1 Formal Problem Definition

In this section we present the formal problem definition provided in [11]. Some useful definitions and notations:

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 . 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 $a_3 = 1$ ”.

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

The formal definition of the problem is as follows.

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' by retaining m attributes such that the number of queries that retrieve t' is maximized.

| Query ID | AC | Auto Trans | Four Door | Power Brakes | Power Doors | Turbo |
|----------|----|------------|-----------|--------------|-------------|-------|
| q_1 | 1 | 0 | 1 | 0 | 0 | 0 |
| q_2 | 1 | 0 | 0 | 0 | 1 | 0 |
| q_3 | 0 | 0 | 1 | 0 | 1 | 0 |
| q_4 | 0 | 0 | 0 | 1 | 1 | 0 |
| q_5 | 0 | 0 | 0 | 0 | 0 | 1 |
| q_6 | 0 | 1 | 0 | 0 | 0 | 0 |

Query Log Q

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

New tuple t to be inserted

Figure 1: Illustrating EXAMPLE 1

Intuitively, for buyers interested in browsing products of interest, this is 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 later in the paper.

EXAMPLE 1: Consider Figure 1 which shows a query log for of auto search queries, containing $S=6$ queries and $M=6$ attributes where each tuple (query) represents the preferences of a user. The figure also illustrates a new car t that needs to be advertised in the marketplace. 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. □

2.2 Complexity Results

The problem *CB-QL* described above proved to be NP-complete in [11].

3. ALGORITHMS

In this section we discuss our main algorithmic results. We omit further discussion on non-scalable optimal algorithm based on Integer Linear Programming developed in [11]. We compare our proposed algorithm with another optimal algorithm developed in [11] which is based on Maximal Frequent Itemsets (*MaxFreqItemSets*). We give a brief introduction of *MaxFreqItemSets* algorithm next.

3.1 Optimal Algorithm based on Maximal Frequent Itemsets (*MaxFreqItemSets*)

The *MaxFreqItemSets* algorithm developed in [11] works as follows: First it complements the query log, i.e., convert 1's to 0's and vice versa. Let $\sim Q$ denote the complement of a query log Q where the each query (tuple t) has been complemented ($\sim t$). Then computes all frequent itemsets of $\sim Q$ (using an appropriate threshold), and from among all frequent itemsets of size $M - m$ (where M is the total number of attributes and m is the number of attributes needs to retain), determines 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., $\sim I$. As generating all frequent itemsets is an inefficient approach for a dense dataset like $\sim Q$, the algorithm generate all maximal frequent itemsets employing a top-down random walk approach and then frequent itemsets of size $M - m$.

The algorithm assumes that repeating the two phase random walk approach employed several times will discover, with high probability, all the maximal frequent itemsets. The 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). But this is a heuristic and hence does not guarantee to produce all maximal frequent itemsets in all situations which sometimes might lead to a non-optimal answer. So we are motivated to develop an optimal solution that guarantees optimal answers always and performs better for certain data distribution. We discuss our proposed optimal algorithm next.

3.2 Optimal Algorithm based on Signature Tree Data Structure (SigTree)

We propose a novel optimal algorithm based on adaptations of the Signature Tree data structure [4] employing some smart pruning techniques. Signature trees were originally used for effective indexing and search over a set of signatures, where a signature is a bit string representation of a database tuple. This has applications in transaction databases which are collections of Boolean tuples (transactions) in which a value 0 means a feature (or attribute) is not present and 1 means it is present.

3.2.1 Review of Signature Trees

We illustrate Signature Trees by means of an example. Consider Figure 2(a) which shows a Boolean database with 6 tuples and 6 attributes. The signature of a tuple is simply the bit vector that describes the values assigned to each attribute. For example, in Figure 2(a), the signature of t_3 is 001010. Figure 2(b) shows the corresponding signature tree. In general, given a Boolean database $D = \{t_1 \dots t_N\}$, a signature tree is a binary tree T such that

- T has N leaves labeled with the signatures of the tuples $t_1 \dots t_N$.
- Each internal node v is associated with an attribute $a(v)$. Note that several internal nodes may be associated with the same attribute, e.g. in Figure 2(b) attribute a_3 is associated with two different internal nodes. However the same attribute cannot be repeated along any root to leaf path.
- For each internal node v , the left edge below it is always labeled with 0 and the right edge is always labeled with 1. All tuples reachable from v via the left (resp. right) edge have $a(v) = 0$ (resp. $a(v) = 1$). For example, in Figure 2(b), t_3 , t_4 and t_5 all have $a_2 = 0$.
- Each path from the root to a leaf defines assignments of values to the corresponding attributes of the internal nodes along the path, and these assignments uniquely identify the tuple. For example, in Figure 2(b), t_3 is identified by the assignments $a_1 = 0$, $a_2 = 0$ and $a_3 = 1$.

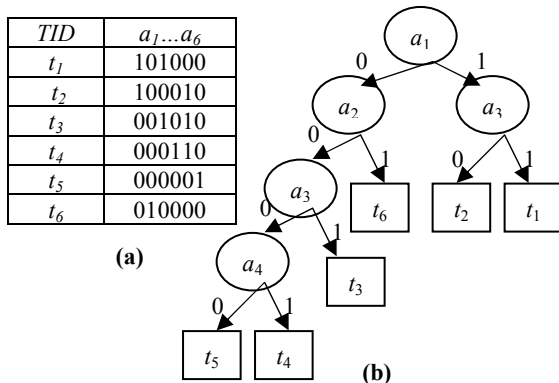


Figure 2. A Boolean Database and a Signature Tree

Given a signature tree, searching for tuples is straightforward. For example, let us search for tuple t_3 in the signature tree in Figure 2(b). We start at the root, and since $a_1 = 0$, we proceed to the left child of the root. Since a_2 is also 0 we proceed to the left child of node a_2 . Now, since a_3

$= 1$, we proceed to the right and arrive at the t_3 leaf node. Clearly not all attributes of t_3 need to be checked by this search process.

There have been several techniques proposed for building signature trees; here we briefly outline a simple procedure described in [14]. We first determine an attribute that can split the tuples in two non-empty groups. For the above example in Figure 2(a), we see that attribute a_1 splits the tuples in two non-empty groups – one with tuples t_3 , t_4 , t_5 and t_6 with $a_1 = 0$, and another with tuples t_1 and t_2 with $a_1 = 1$. Attribute a_1 is associated with the root of the tree, and the left sub-tree is built recursively from the first group, while the right sub-tree is built recursively from the second group. Figure 2(b) shows the signature tree created by this process for the database in Figure 2(a).

Traditional signature trees as described above are not directly applicable for the attributes selection problem considered in this paper, mainly because they were originally motivated for effective search and indexing, whereas we are interested in solving an optimization problem. Thus, they require significant modifications. In our approach, we shall model the query log as a signature tree.

| Query ID | Signature($a_1 a_2 \dots a_6$) |
|----------|----------------------------------|
| q_1 | 101000 |
| q_2 | 100010 |
| q_3 | 001010 |
| q_4 | 000110 |
| q_5 | 000001 |
| q_6 | 010000 |

Figure 3: Signature Query Log Q

3.2.2 Optimal Algorithm (SigTree)

Let us consider the signatures of the query log Q in our running example (see Figure 3).

Building Signature Tree. We build a balanced signature tree for the query log using the weight based method [4]. A balanced signature tree is a signature tree which is completely or almost evenly balanced. The method of building a balanced signature tree is described below. The tree might not be always perfectly balanced, but it would be close to being evenly balanced.

We use $Q[i]$ to represent the i th column of Q . We calculate the weight of each $Q[i]$, i.e., the number of 1's appearing in $Q[i]$, denoted $w(Q[i])$. Then, we choose a j such that $|w(Q[j]) - S/2|$ is minimum, where S is the total number of queries in the query log Q . Here, the tie is resolved arbitrarily. Using this j , we divide Q into two groups $g_1 = \{q_{i1}, q_{i2}, \dots, q_{ik}\}$ with each $q_{ip}[j] = 0$ ($p = 1, \dots, k$) and $g_2 = \{q_{ik+1}, q_{ik+2}, \dots, q_{iS}\}$ with each $q_{iq}[j] = 1$ ($q = k + 1, \dots, S$); and generate a tree as shown in Figure 4(a). In fact, we partition the signatures based on the value on column j ; signatures with value 0 on column j go into one group and signatures with value 1 on column j go into another group. In a next step, we consider each g_i ($i = 1, 2$) as a single query log and perform the same operations as above, leading to two trees generated for g_1 and g_2 , respectively. Replacing g_1 and g_2 with the corresponding trees, we get another tree as shown in Figure 4(b). We repeat this process until the leaf

nodes of a generated tree cannot be divided any more. Considering our running example, we can see that at the first time the sum of 1's in each column $w(Q[i])$ is as follows: $a_1 (AC) = 2, a_2 = 1, a_3 = 2, a_4 = 1, a_5 = 3,$ and $a_6 = 1$. Here, $S = 6$ which is the total number of queries in the query log Q . So, a_5 has the minimum value for $|w(Q[i]) - S/2|$ which is $(3 - 6/2) = 0$. So we choose a_5 which is *Power Doors* as the root of the tree. We follow the same process for each sub-tree from the root. In Figure 4(a), $g_1 = \{q_1, q_5, q_6\}$ and $g_2 = \{q_2, q_3, q_4\}$; and, in Fig. 4(b), $g_{11} = \{q_5, q_6\}, g_{12} = \{q_1\}, g_{21} = \{q_3, q_4\},$ and $g_{22} = \{q_2\}$. Figure 5 shows the complete signature tree built for the query log of our running example.

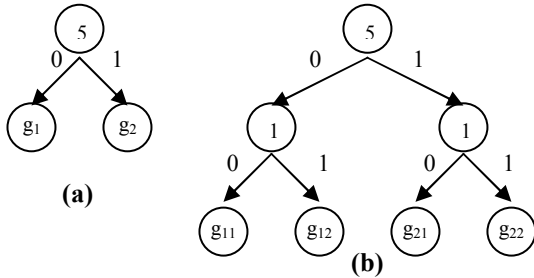


Figure 4: Process of building Signature Tree

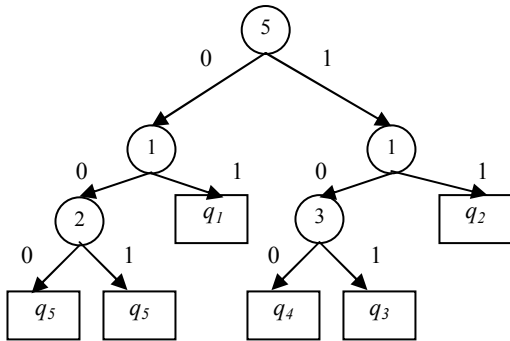


Figure 5: Signature Tree for the Query Log Q

First in the *join* step, it joins L_{k-1} with L_{k-1} :
 Insert into C_k
 select $p.item_1, p.item_2, \dots, p.item_{k-1}, q.item_{k-1}$
 from $L_{k-1} p, L_{k-1} q$
 where $p.item_1 = q.item_1, \dots, p.item_{k-2} = q.item_{k-2},$
 $p.item_{k-1}, = q.item_{k-1}$

Next, in the *prune* step, it deletes all itemsets $c \in C_k$ such that some $(k-1)$ subset of c is not in L_{k-1} :
 for all itemsets $c \in C_k$ do
 for all $(k-1)$ -subsets u of c do
 if $(u \notin L_{k-1})$ then
 delete c from C_k

Figure 6: Function *apriori-gen*

At this step we generate the signature tree only for the queries with less than or equal to m attributes. The reason we ignore the queries with more than m attributes is that none of them can eventually be subset of any m -attributes candidate set which we generate in next step. This will be an efficient technique where there are many queries which have more than m attributes present in the query log.

Generating Possible Candidate Itemsets. After building the signature tree for the query log, we search the tree for possible candidate sets with m attributes. We adapt the candidate set generating function *apriori-gen* used in the *Apriori* algorithm for mining association rules [1] to generate possible candidate sets in this algorithm. The *apriori-gen* function takes L_{k-1} , the set of all large $(k-1)$ itemsets. It returns the set L_k of all large k -itemsets. The function works as shown in Figure 6.

Searching the Signature Tree for an m -itemset. As in the *Apriori* algorithm [1], we start with frequent 1 -itemsets (attribute sets). A *minimum support* is used such that when we select top- m attributes for the new tuple t , then t should be retrieved by the number of queries at least or equal to the *minimum support*. A minimum support is the lower bound such that at least these many queries should retrieve the new tuple. We use a heuristic method to select a good *minimum support*. We first use a fixed value, for example 1% and execute the algorithm. Then we change the minimum support as required, for example if we find no queries for the new tuple then we decrease the minimum support and if too many queries found then we increase the minimum support until a good value for minimum support is set. Using the minimum support we generate frequent 1 -itemsets (attribute sets) from the queries. Here we only consider the attributes which are present in the new tuple to be advertised. One approach now could be to generate all possible m -attribute sets using *apriori-gen* function, and then search the tree. We can search the signature tree as follows:

- a) Create signature for each of the m -attribute candidate sets. Let q_t be the candidate set signature. The i th position of q_t is denoted as $q_t[i]$. During the traversal of a signature tree, the inexact matching is done as follows:
 - i. Let v be the node encountered and $q_t[i]$ be the position to be checked.
 - ii. If $q_t[i] = 0$, we move to the left child of v .
 - iii. If $q_t[i] = 1$, both the right and left child of v will be explored.

In fact, this process just corresponds to the signature matching criterion, i.e., for a bit position i in q_t , if it is set to 0, the corresponding bit position in q must be set to 0; if it is set to 1, the corresponding bit position in q can be 1 or 0. In a traditional signature tree, a query q is passed to the tree and finds the transactions (leaf nodes of the tree) which are possibly the supersets of q (i.e., how many transactions will be retrieved by the query). But our problem is different. We pass a candidate m -attribute set c to the tree and find the queries (leaf nodes) which are possibly subsets of c . Searching the tree is done in a depth-first manner. When we reach a leaf node, we match all the signatures of the leaf node with m -attribute candidate set c . Here queries have to be subsets of c . We keep a count for each candidate set c that how many queries have been found as subsets of c . We remove the candidate set c if the total count for it is less than the *minimum support*.

- b) For all m -attributes candidate sets found in step (a), we simply return the set has the highest count.

There are two major problems with this approach: (i) the number of candidate sets can be huge as there is no pruning at intermediate steps, and (ii) small itemsets would get an unfairly small count because it increases the count of a candidate if it satisfies the whole query in the signature tree. Hence, in order to be able to grow the candidate itemsets and not start directly from m -itemsets, we start generating and searching the tree in order to increase the count of a candidate k -itemset for every query it has a chance to cover if $(m-k)$ items are added. For instance, the 2-itemset 110000 has a chance to cover 110100 if 1 more item is added. So we follow a new method where for each k -itemset we navigate the signature tree from top to bottom and only prune subtrees that need more than $(m-k)$ additional items to be covered.

Using the Signature Tree to Compute Candidate Itemsets. First we create the signature tree for the query log as described earlier. Then, we search the tree at each level from 2-attributes candidate sets to up to m -attributes candidate sets. Candidate sets at each level k ($= 2 \dots m$) are generated using function *a priori-gen* as discussed above. At each level searching is done as follows:

- a) Let v be the node encountered and $q_i[i]$ be the position to be checked.
- b) We move both the right and left child of v whether $q_i[i] = 0$ or $q_i[i] = 1$.
- c) We maintain a variable $m' = (m - k)$, which is the number of *mistakes* allowed, where k is the current number of attributes in the candidate set. A mistake during the search is: when we move to the right of a node (i.e., query has value 1 for the attribute mentioned by the node) and the candidate set has value 0 for that attribute. If this situation happens, we increase the count for mistakes. Consider the signature tree in Figure 5 for our running example. Assume current value of $k = 2$, $m = 3$, and we have a candidate set with signature 110000 . So, the value of $m' = m - k = 1$. Moving left never increases the number of mistakes because if a query q has value 0 for an attribute then a candidate set c can have either value 0 or 1 for the corresponding attribute. When we move right from the root node (Figure 5), the value of a_5 (root node) in candidate set 110000 is 0, so we increase number of mistake made so far, which is $(0+1) = 1$. Next we move both left and right from node a_1 (right child of root). Moving right from node a_1 does not increase number of mistakes made as the value of a_1 in the candidate set is also 1. But when we move right of the node a_3 (left child of node a_1), we increase number of mistakes made because value of a_3 in the candidate set is 0. Here new value of number of mistakes made so far is $(1+1) = 2$ which is greater than m' . So we do not consider any nodes to the right of node a_3 . As we can see from the tree in Figure 5, we do not consider q_3 with signature 001010 as a possible subset of the candidate set 110000 in future.
- d) Once we reach a node and number of mistakes made so far reaching the node from the root is greater than m' , then we do not consider the node and its children (if it is

an internal node) as possible subset of the candidate set.

Otherwise, we keep the query for further match

Once we find the corresponding queries (leaf nodes) by searching the tree for a candidate set c , for each query q_i we do the following:

- a) We find the number attributes r present in the query q_i which is not present in candidate set c . If $r \leq (m - k)$, we count q_i as the possible subset of c .
- b) We keep a count for each candidate set c that how many queries have been found as possible subsets of c . We remove the candidate set c if the total count for it is less than the *minimum support*.
- c) At level m (candidate sets with m -attributes), we simply check how many queries (found after searching the tree) are actually the subsets of the candidate set. We return the candidate set with highest count as the top- m attributes for the new tuple t .

Our proposed *SigTree* algorithm described above always guarantee optimal answer while the existing *MaxFreqItemSets* algorithm [11] might not produce optimal answer in all situations because of the heuristic used to generate all maximal frequent itemsets (described in Section 3.1). Also the algorithm *SigTree* performs better than the algorithm *MaxFreqItemSets* for certain data distribution, e.g., when there are duplicates exist in the dataset, which makes the tree smaller and smaller number of leaf nodes. Searching the tree is also becomes faster for this reason.

4. RELATED WORK

A large corpus of work has tackled the problem of ranking the results of a query such as tf-idf based [12] ranking functions in documents world, link-structure-based techniques like PageRank [3], automatic ranking techniques [2, 5, 13] in database world, as well as ordering the displayed attributes of query results [6]. 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.

Our work also differs from the extensive body of work on feature selection [7], 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. Boosting an item's rank also has received attention is Web search, where the most popular techniques involve manipulating the link-structure of the Web to achieve higher visibility [8].

Recent works on skyline semantics [9, 10] aim 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.

5. EXPERIMENTS

In this section we measure (a) the time cost, and (b) the quality of the existing and proposed optimal algorithms.

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#.

Datasets: We use two synthetic query logs (datasets) for auto search: one is based on *uniform* distribution and the other is based on *zipfian* (*Zipf's law*) distribution.

Synthetic query log based on uniform distribution: We generate synthetic dataset for cars based on uniform distribution where each tuple represents a user search query for a car. There are 32 Boolean attributes, such as *AC*, *Power Locks*, etc. for a car. In the synthetic query log, each query specifies 1 to 5 attributes (users like to have these attributes on the cars they search) chosen randomly

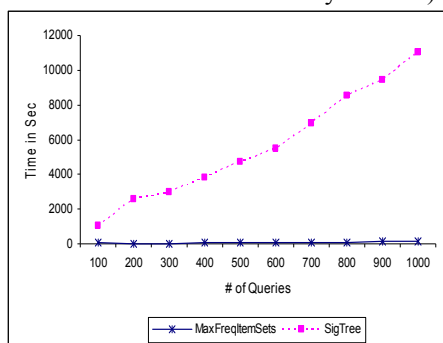


Figure 8: Time performance for varying query log for synthetic data with *uniform* distribution ($m = 5$)

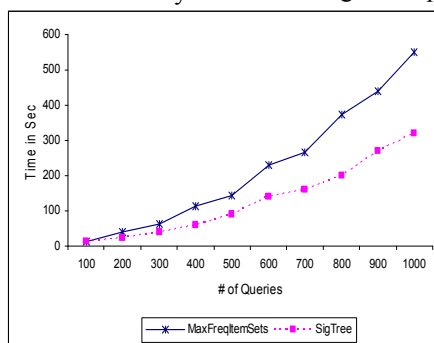


Figure 9: Time performance for varying query log for synthetic data with *zipfian* distribution ($m = 5$)

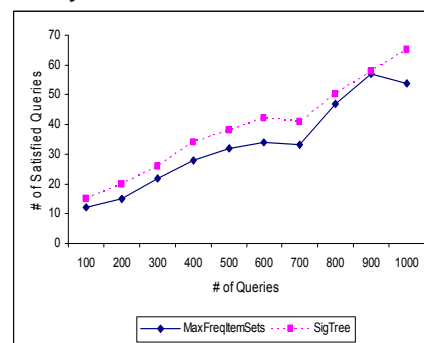


Figure 10: Quality for varying query log for synthetic data with *uniform* distribution (number of queries = 500)

Figures 8 and 9 show the performance of the algorithms for *uniform* and *zipfian* data distribution respectively. As we can see from the graphs that the algorithm *MaxFreqItemSets* performs better than the algorithm *SigTree* for uniform data, whereas, the algorithm *SigTree* performs better than the algorithm *MaxFreqItemSets* for *zipfian* data as the query log increases. This is because for *zipfian* data distribution there are some duplicates exist which makes the signature tree smaller and smaller number of leaf nodes. Searching the tree is also becomes faster for this reason. The more duplicates exist in the query log the faster the *SigTree* algorithm is.

Both the algorithms return optimal (same) answers (number of satisfied queries) for *zipfian* distribution (results not shown). But as we can see in Figure 10 that the algorithm *MaxFreqItemSets* does not produce optimal answers for *uniform* data distribution, whereas our proposed algorithm *SigTree* does. The reason could be as discussed in Section 3.1 that the *MaxFreqItemSets* algorithm uses the heuristic to generate all maximal frequent itemsets that might not perform optimally in all situations.

6. CONCLUSIONS

In this work we investigated the problem of selecting the best attributes of a new tuple, such that this tuple will be viewed by as many potential customers as possible. We presented optimal solution for the NP-complete problem and compared with existing methods. We showed that our proposed solution performs better than the existing methods in certain situation as well as guarantee optimal output in all situations where existing methods might fail.

distributed evenly as follows: 1 attribute – 20%, 2 attributes – 20%, and so on. We assume that most of the users specify small number of attributes.

Synthetic query log based on zipfian distribution: 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 use this dataset to rank the attributes based on the number of cars have the attribute/feature present. Then we calculate the probability for each attribute using *Zipf's law* [14] to generate the synthetic query log. We generate synthetic query log based on the probability found using *Zipf's law*. Each query specifies 1 to 5 attributes (with value equals 1) according to the probability.

7. REFERENCES

- [1] R. Agrawal, R. Srikant: Fast Algorithms for Mining Association Rules. VLDB 1994: 487-499.
- [2] S. Agrawal, S. Chaudhuri, G. Das, A. Gionis: Automated Ranking of Database Query Results. CIDR 2003.
- [3] S. Brin, L. Page: The Anatomy of a Large-Scale Hypertextual Web Search Engine. WWW Conf., 1998
- [4] Yangjun Chen, Yibin Chen: On the Signature Tree Construction and Analysis. IEEE TKDE. 18(9), 2006.
- [5] S. Chaudhuri, G. Das, V. Hristidis, G. Weikum: Probabilistic Ranking of Database Query Results. VLDB 2004
- [6] G. Das, V. Hristidis, N. Kapoor, S. Sudarshan: Ordering the Attributes of Query Results. SIGMOD'06
- [7] I. Guyon, A. Elisseeff: An introduction to variable and feature selection. Journal of Machine Learning Research, 3(mar):1157-1182, 2003.
- [8] M. Gori and I. Witten. The bubble of web visibility. Commun. ACM 48, 3 (Mar. 2005), 115-117
- [9] C. Li, B. C. Ooi, A. K. H. Tung, S. Wang: DADA: A Data Cube for Dominant Relationship Analysis. SIGMOD 2006.
- [10] C. Li, A. K. H. Tung, W. Jin, M. Ester: On Dominating Your Neighborhood Profitably. VLDB 2007: 818-829
- [11] Muhammed Miah, Gautam Das, Vagelis Hristidis, Heikki Mannila: Standing Out in a Crowd: Selecting Attributes for Maximum Visibility. ICDE 2008.
- [12] G. Salton. Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer. Addison Wesley, 1989
- [13] W. Su, J. Wang, Q. Huang, F. Lochovsky: Query Result Ranking over E-commerce Web Databases. CIKM 2006.
- [14] George K. Zipf: Human Behavior and the Principle of Least-Effort. Addison-Wesley, 1949.