International Journal of Science and Engineering Applications
Volume 6 Issue 03, 2017, ISSN-2319-7560 (Online)

Smallest Set for Reverse Selection Queries to Satisfy All

Muhammed Miah
Department of Computer Information Systems
Southern University at New Orleans
New Orleans, LA 70126, USA

Abstract: In recent years, research has moved from traditional query processing (e.g., Selection, Nearest Neighbor (NN), Top-*k*, Skyline), to reverse query processing (e.g., Reverse NN, Reverse Top-*k*, Reverse Skyline), to maximal reverse query processing (e.g., find spatial points that maximize the number of Reverse NNs), and so on. This paper considers the *Smallest Set Reverse Selection Queries Problem* also known as the *Multiple Tuple Design Problem*: Given a set of selection queries with conjunctive conditions, where the task is to create the smallest set of tuples such that each query returns at least one of these tuples. The problem is an interesting variant of the Maximal Reverse Selection Queries Problem (also referred to as the *Tuple Design Problem*) introduced by Miah et al. (2016). The paper shows that the problem is NP-Complete and develops approximation algorithms with provable approximation guarantees, as well as carefully designed heuristics that work well in practice. The paper also designs efficient exact algorithm that are feasible for moderate instances. It provides extensive experiments that demonstrate the effectiveness of the proposed algorithms.

Keywords: smallest set; reverse selection queries; multiple tuple/product design; maximize visibility, satisfy all.

1. INTRODUCTION

Significant research has been done in the area of traditional query processing, where a query and a database (set of tuples) are given, and the task is to return all tuples in the database that satisfy the query. Different query models have been developed for query processing including "selection queries" to find the set of tuples that satisfy a selection condition, "*k*NN queries" to find the set of *k* tuples that are closes to a query tuple, "skyline queries" to find all tuples that are not dominated by any other tuple), "top-*k* queries" to find the top-*k* tuples , and so on.

Research has moved in recent years to the complementary area of reverse query processing, where in the database (set of tuples), a query log, and a potential tuple are given, and the task is to find all queries in the query log that return the given tuple. The reverse query processing also has been studied for a variety of query models, such as "reverse *k*NN", "reverse top-*k*", "reverse skyline", and so on. While traditional query processing applications focus on the customers or end users, reverse query processing applications focus on the manufacturers or sellers, e.g., helping manufacturer identify products that are most preferred by customers.

Maximal reverse query processing has been studied as well for skyline (Li et al. 2007; Li et al. 2006) and *k*NN (Cabello et al. 2005; Wong et al. 2009) queries with numeric attributes.

Most recently, an interesting related area of maximal reverse selection queries problem has been introduced. Given a database (set of tuples) and a query log, the task is to construct a new tuple such that the set of queries in the query log that return the new tuple is larger than for any other tuple in the database. This focused on the maximal reverse queries problem on the important class of "selection queries" over Boolean databases. Assume that a Boolean tuple (e.g., a new product) needs to be designed by selecting a subset of Boolean features (or attributes) from a large set of possible features. Assume that we are given a set of user preferences in the form of a query log (or workload) of user queries, where each query is a conjunction of positive or negative preferences for some of the features (e.g., "Select * from *Database* where a_1=0 and a_4=1 and a_6=1"). The problem also known as *Tuple Design (TD) or Single Tuple Design (STD) Problem* (Miah et al. 2016).

This paper focus on an interesting variant of the Single Tuple Design problem. Instead of just designing a single tuple, we may be interested in creating a minimum number of tuples that collectively satisfy all queries in the query log. This is referred to as the *Multiple Tuple Design (MTD) problem*.

MTD (Multiple Tuple Design) Problem: Given a query log *Q* consisting of conjunctive selection queries over Boolean attributes, construct the smallest set *T* of Boolean tuples, such that for each query *q* in *Q*, there exists a tuple *t* in *T* that is returned.

The *MTD* problem has several potential applications. Consider a travel agency that wishes to design vacation packages, given the travel preferences of its clients. For example, a vacation package to Costa Rica can include some of the following attractions: beaches such as Puerto Vijeo, Jaco, Flamingo, etc.; mountains and national parks such as Arenal area, Monteverde, Tortuguero, etc. The clients of the agency provide their preferences by specifying "yes", "no", or "don't care" for each attraction. The travel agency might want to create a minimum set of vacation packages to satisfy all its customers, because each package induces fixed overheads to the agency, e.g., requires a dedicated vacation guide, or a transportation vehicle, etc. Likewise, a product manufacturer may wish to design and manufacture a small range of products that will cover the preferences of all customers.

One of the practical challenges is to ensure the availability of a large and rich query log to make the design of the new product/package truly effective for the customer base. While customer preferences can of course be explicitly collected through tools such as surveys and questionnaires, a very effective alternative is to implicitly collect such preferences by observing and recording user behavior on the internet – e.g., their browsing and navigation patterns on a product manufacturer or e-tailer's website, such as the pages and

products they click on. The vast use of the Internet nowadays allows enormous amounts of such preferences to be very easily collected. However, in this paper, we do not focus on how exactly such query logs can be collected. Rather we assume the presence of such logs, and focus more on the technical challenges, i.e., designing effective and scalable solutions of the maximal reverse selection query problems (Miah et al. 2016).

The *MTD* problem is technically challenging because the problems turn out to be NP-complete (proof of NP-completeness shown later in the paper). It is therefore necessary to design good approximation algorithm that has provable approximation bound or heuristics that work well in practice, or even exact algorithms that work well for moderate problem instances. However, it is not easy to simply reuse well-known approximation algorithms for related NP-complete problems, as none of these algorithms are an exact fit for *MTD*.

Major Contributions:

- The paper considers the problem of Maximal Reverse Selection Queries for Boolean databases, and focus on specific problem, the Multiple Tuple Design (*MTD*) problem. The first result is to show that *MTD* is NP-complete. The proof for *MTD* uses a reduction from Graph Coloring (*GC*).
- Approximation algorithms for the *MTD* problem are developed. The first algorithm has a provable approximation bound, and is based on a combination of known approximation algorithms (and their approximation factors) of two separate NP-complete problems. However, this algorithm is mainly of theoretical interest as its approximation factor is quite large.
- Two other more practical approximation algorithms are also developed. While these algorithms do not guarantee any approximation bounds, they are scalable and are shown to have small approximation factors in practice.
- The extension of the problem to the Categorical and Numeric Databases also discussed.
- Detailed performance evaluations are performed on real and synthetic data to demonstrate the effectiveness of the developed algorithms.

The rest of the paper is organized as follows. Section 2 provides formal problem definitions for *MTD*. Section 3 analyzes the computational complexities for the problems. Section 4 presents approximation algorithms with provable bounds. Section 5 provides approximation heuristics that are scalable and have small approximation factors in practice. Section 6 presents the result of extensive experiments. Section 7 provides the extension of the problems to other databases. Related work is discussed in Section 8, conclusions in Section 9, and Section 10 provide references.

2. PROBLEM FRAMEWORK

To define the problem more formally, we need to develop a few abstractions.

Attributes: Let $A = \{a_1 \ldots a_M\}$ be a set of Boolean attributes (or elements, or features).

Query (with negation): We view each user query as a subset of attributes and/or negation of attributes. The semantics is *conjunctive*, e.g., query $\{a_1, a_3\}$ is equivalent to "$a_1 = 1$ and $a_3 = 1$". We also consider queries *with negations*, e.g., $\{a_1, \sim a_2\}$ is equivalent to "$a_1 = 1$ and $a_2 = 0$". The remaining attributes for which values are not mentioned in the query are assumed to be "don't care", i.e., the value in the new designed tuple can be either 0 or 1.

Query Log or Workload: Let $Q = \{q_1 \ldots q_S\}$ be a collection of queries.

For the *MTD* problem, we need to design a number of tuples. In fact, we need to design the minimum number of tuples such that together they satisfy all the queries in the query log. Below is more formal definitions of the problems introduced in Section 1:

Multiple Tuple Design (MTD) Problem: Given a query log Q with conjunctive semantics where a query can have negations, design the minimum number of tuples (assign value [0, 1] for each attribute in each tuple) such that for each query of Q there exists a tuple that satisfies it.

Example 1. Consider Table 1 which shows a query log for a vacation package application, containing S=6 queries and M=6 attributes where each tuple (query) represents the preferences of a user. A query has values 1, 0, or ?, where 1 means the attribute must be present, 0 means the attribute must not be present, and "?" means "don't care". As we can see for this example that if we design three new packages as t_1 (with *Beach* = 1, *Boating* = 0, *Casino* = 0, *Fishing* = 1, *Historical Site* = 1, *Museum* = 0 which satisfies three queries q_2, q_4 and q_6), t_2 (with *Beach* = 0, *Boating* = 1, *Casino* = 1, *Fishing* = 0, *Historical Site* = 1, *Museum* = 1 which satisfies two queries q_3 and q_5), and t_3 (with *Beach* = 1, *Boating* = 0, *Casino* = 1, *Fishing* = 1, *Historical Site* = 0, *Museum* = 0 which satisfies one query q_1), we can satisfy all 6 queries in Table 1. No other combination of three or less packages will satisfy all queries. □

Table 1. Query Log Q for Running Example

Query ID	*Beach*	*Boating*	*Casino*	*Fishing*	*Historical Site*	*Museum*
q_1	1	0	1	?	?	?
q_2	1	?	0	?	1	?
q_3	0	?	1	?	1	?
q_4	?	0	0	1	1	?
q_5	?	1	?	0	?	1
q_6	1	0	0	?	?	0

3. COMPLEXITY RESULTS

The *MTD* problem is NP-complete.

Theorem 1: *MTD* is NP-complete.

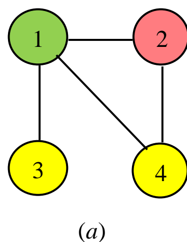
Proof: The problem is clearly in NP as a proposed solution can be easily verified in polynomial time by a single pass over the query log. To prove that it is NP-complete, we reduce the Graph Coloring (*GC*) problem to *MTD*. In the *GC* problem, we wish to assign colors to the vertices of a graph such that no two adjacent vertices share the same color, and are required to

find the minimum number of colors needed to color the graph (this number is known as the chromatic number of the graph).

The reduction is as follows. Let $G = (V, E)$ be the graph in the *GC* instance. Edges will correspond to Boolean attributes and vertices will correspond to queries. Each edge $e_i = (v_j, v_k) \in E$ represents a Boolean attribute a_i, with the condition $a_i = 1$ a part of the query q_j corresponding to vertex v_j and the condition $a_i = 0$ a part of the query q_k corresponding to vertex v_k. (i.e., with one end point representing the positive literal a_i and the other end point representing the negative literal $\sim a_i$). To make it little more clear, when considering vertex v_j (for query q_j), then only considering the end point of edge e_j connected to v_j, not the end connected to v_k; and similarly when considering vertex v_k (for query q_k), then the same edge e_j is being represented as e_k only considering the end point of edge e_k connected to v_k, not the end connected to v_j. Thus, each vertex v_j represents a query q_j, where the query is represented as a conjunction of the literals corresponding to the end points of all edges that are incident to v_j, and the remaining attributes are "don't care"s.

Thus, Figure 1(a) shows a 3-colorable graph G with a valid coloring and Figure 1(b) shows the corresponding query log for the *MTD* instance. It is easy to see that finding the chromatic number of G is equivalent to finding the smallest set of tuples (with an assignment of {0, 1} values to each Boolean attribute) in the *MTD* instance that satisfies all the queries. □

The *GC* problem is NP-hard only for graphs with chromatic number > 2 and hence the *MTD* problem is NP-hard only if we need to design more than 2 tuples to cover all queries.



(*a*)

Query ID	$a_{1,2}$ ($a_{2,1}$)	$a_{1,3}$ ($a_{3,1}$)	$a_{1,4}$ ($a_{4,1}$)	$a_{2,4}$ ($a_{4,2}$)
q_1	1	1	1	0
q_2	1	0	0	1
q_3	0	1	0	0
q_4	0	0	1	1

(*b*)

Figure 1. An instance graph and corresponding query log.

4. APPROXIMATION ALGORITHM WITH PROVABLE BOUND

Although the *Graph Coloring* (*GC*) problem is used to prove NP-completeness of *MTD*, *GC* does not have a good bounded-factor approximation algorithm. In fact, the best known approximation algorithm has an approximation factor that can be very close to the number of vertices in the graph. Hence such algorithms are not useful for developing bounded-factor approximation algorithms for *MTD*. Moreover, an approximation algorithm for *GC* is not directly applicable to *MTD* problem.

Instead, proposed approach for developing an approximation algorithm for *MTD* is based on two steps. We first define a problem that is more convenient than *MTD* for developing approximation algorithms, called the *Multiple Tuple Selection* problem (*MTS*). We then consider an approximation algorithm for *MTS*, and make modifications so that it can eventually be used to solve *MTD*. The modifications entail making repeated invocations to the approximation algorithm for *STD* (Miah et al. 2016) for different instances of *STD*. The details are given below.

The *Multiple Tuple Selection* (*MTS*) problem is almost the same as the *MTD*, except that in addition to the query log, a set of candidates tuples *P* is also given, and we are restricted to select the minimum subset of tuples *T* from this set *P*.

Lemma 1: *MTS is NP-Complete*

Proof: The problem is clearly in NP as a solution can be verified in polynomial time. To prove NP-completeness, we reduce from the *Hitting Set* (*HS*) problem (Garey and Johnson 1979). Given a ground set *Z* of elements, and a collection *Y* of subsets of *Z*, the goal of the *HS* problem is to find the smallest subset $H \subseteq Z$ of elements that hits every set of *Y*.

The reduction of *HS* to *MTS* is as follows. Assume *MTS* has *V* candidate tuples, *S* queries, and *M* attributes; and *HS* has *n* sets and *m* elements. For each element e_i in *HS*, create a query q_i and an attribute a_i in *MTS*. Set $a_i(q_i) = 1$ and the rest of the attributes of q_i to "don't care". For each set s_i in *HS*, create a tuple t_i in *MTS*. For each element e_j in s_i, set $a_j(t_i) = 1$. Set the rest of the attributes of t_i to 0. Then, a solution to *HS* is a solution to *MTS* and vice-versa.

Note that in the above reduction, we set $S = M = m$ and $V = n$. That is, we use as many attributes as queries. If we would assume that $M = log(S)$, then the above reduction is not valid and it is an open problem whether the problem is NP-complete or not. □

Even though the *MTS* problem is NP-complete (as is *MTD*), *MTS* has an advantage over *MTD* - it can be solved using the well-known greedy approximation algorithm for the *SETCOVER* problem (Cormen et al. 2001), which we describe below.

An instance (*X*, *F*) of the *SETCOVER* problem consists of a finite set *X* and a family *F* of subsets of *X*, such that every element of *X* belongs to at least one subset of *F*. The goal is to find a minimum subset $E \in F$ whose members cover all of *X*. A greedy approximation algorithm (Cormen at al. 2001) provides an approximation bound of $H(max\ \{|B| : B \in F\})$ where $H(d)$ represents the dth harmonic number which is equal to $log(max\ \{|B| : B \in F\}) + O(1)$. The greedy algorithm works as follows: (*i*) at each iteration, the algorithm picks the set with highest number of elements from the sets not picked yet, and (*ii*) the process repeats until all elements are covered.

We can directly relate *MTS* to the *SETCOVER* problem as follows: let us assume *X* is the query log (*Q*) and *F* is the set of given tuples (*P*), where each package represents the set of queries that are satisfied by it. Now the goal is to find the minimum set of tuples *T* from *P* such that they cover all the

queries in Q. Thus *MTS* also has the same approximation bound as the *SETCOVER* problem which would be $log(max\{|V| : V\in P\}) + O(1)$, where V is a subset $V\in P$ which covers all queries in Q. Thus, in the worst case the approximation factor for *MTS* is $log(S)$ where S is the number of queries in the query log Q.

Extending this approach to the *MTD* problem is challenging as we do not have a set of candidate packages. Moreover, it is not possible to simply first enumerate all possible packages, as this is exponential in the number of attributes! However, one of the interesting contributions of this paper is the observation that *we can avoid this enumeration by combining the approximation algorithm for STD (discussed earlier) with the greedy approximation algorithm for SETCOVER to eventually design an approximation algorithm for MTD.* Essentially, we take the greedy algorithm for *SETCOVER*, and make modifications that entail making repeated invocations to the approximation algorithm for *STD* for different *STD* instances.

This combined algorithm proceeds as follows: (*i*) at each iteration the algorithm makes a call to the approximation algorithm for *STD* (Miah et al. 2016) over the not-yet-satisfied queries, which returns a tuple such that the number of new queries that are satisfied is at least $ck/2^k$ times the number of new queries that would have been satisfied by an optimal tuple, and (*ii*) the process repeats until all queries are satisfied.

The following theorem shows that the above algorithm has a provable approximation bound.

Theorem 2: *The number of tuples returned by the approximation algorithm described above is at most* $(2^k/ck)log(S)$ *times the number of tuples returned by an exact algorithm for MTD.*

Proof: Let T_{min} be the optimal (minimum) set of tuples that together satisfy all queries in the query log Q. Let $T' = t'_1, t'_2, \ldots, t'_x$ be the sequence of tuples returned by the above approximation algorithm. We shall show that $x=|T'| \leq |T_{min}|(2^k/ck)log(S)$.

Let us imagine that each tuples is allotted a weight of 1, and the weight is evenly distributed to all queries that are satisfied for the first time by that package. For example, if t'_1 satisfies queries q_1 and q_2, then $wt(q_1) = wt(q_2) = 1/2$; if tuple t'_2 satisfies query q_3, q_4, and q_5, then $wt(q_3) = wt(q_4) = wt(q_5) = 1/3$. It is easy to see that the accumulated weight of all the queries aggregate to x.

Consider any tuple t_i of T_{min}. Let $t'_{i,1}, t'_{i,2}, \ldots t'_{i,r}$ be the order in which the approximation algorithm returns tuples that have non-empty intersection with t_i, i.e., that there exists at least one query in the query log that is satisfied by both $t'_{i,j}$ and t_i.

To make notation convenient, we will refer to any tuple t as the set of queries that it satisfies, and $|t|$ as size of this set.

Let $u_{i,j} = |t' - (t'_{i,1} \cup t'_{i,2}, \ldots \cup t'_{i,j})|$. Thus, of all the queries that are satisfied by t', $u_{i,j}$ refers to the number of queries that are still not satisfied by the approximation algorithm at the time the last generated tuple is $t'_{i,j}$.

Consider $t'_{i,j+1}$. We argue that $|t'_{i,j+1}| \geq (ck/2^k)\ u_{i,j}$. Because if this were not so, then the greedy approximation algorithm for *STD* would not have returned $t'_{i,j+1}$, since the number of remaining queries it would have satisfied would have been too small and its known approximation bound would have been violated – in fact, t_i itself would have been a better package to return next instead of $t'_{i,j+1}$.

Thus, the aggregate weight of all queries satisfied by t_i is

$$\leq \sum_{1\leq j\leq r}\left(u_{i,j+1}-u_{i,j}\right)\frac{1}{\left(\frac{ck}{2^k}\right)u_{i,j}} \leq \sum_{1\leq j\leq S}\frac{1}{\left(\frac{ck}{2^k}\right)j} \leq \left(\frac{2^k}{ck}\right)\log(S)$$

Thus, the aggregate weight of all queries is

$$x = |T'| \leq |T_{\min}|\left(\frac{2^k}{ck}\right)\log(S)$$

□

In summary, the above discussions demonstrate the existence of bounded approximation factor algorithms for the *MTD* problem. However, while interesting from a conceptual point of view, in practice these approximation factors are too suboptimal to be useful, and the algorithms themselves require complex in-memory implementations based on semi-definite programming (*SDP*) relaxation methods. More practical approximation algorithms are needed that work well for moderate as well as large problem instances. Such algorithms are discussed in the next section.

5. SCALABLE APPROXIMATE ALGORITHMS

The approximation algorithm of Section 4 is useful for theoretical purposes, but it is not very feasible in practice. This is because that is main memory algorithms based on semi-definite programming (*SDP*) relaxation, which do not scale for large instances. Moreover, the bounded approximation factors are too suboptimal to be useful in practice. This section proposes scalable approximation algorithm *MTD*, which are shown to perform well in terms of scalability and approximation error in Section 6.

The solution for *MTD* involves a combination of the solution of multiple instances of *STD*. Miah et al. (2016) proposed an exact algorithm for *STD* based on Signature Tree data structures. A depth first creation of the signature tree is adapted, where at each downward step it picks an attribute that will be used as the splitting node. Further, for every traversed node *v*, it checks if this node has a chance to be the best package. That is, it compares the number of satisfied queries in *v* to the *current maximum number C of satisfied queries*. If *v* fails this test, it prunes the whole subtree rooted at *v*, since the children of *v* satisfy at most as many queries as *v*. During the depth-first traversal, it first traverses the child that contains the maximum number of queries, since the objective is to create an assignment of values to all attributes that maximizes the number of satisfied queries. When it finds the node *v* with the maximum *C*, it creates the corresponding "best" tuple by traversing the path from *v* to the root and assigning attribute values accordingly.

Figure 2 shows the pseudocode of *SigTreeSTD* exact algorithm for *STD* (Miah et al. 2016)

```
Algorithm: SigTreeSTD
Signature Tree T //initially single node u that contains all
   queries in Q, that is u=Q.
Let C be the current max # queries in a leaf node of the
signature tree. Initially C ← 0
Stack V
V.push(T) //T is root node of T
While V not empty do
   u←V.top()
   If both children of u have been processed before or there
   is no splitting attribute then
      update C //if |u|>C then C=|u|
      V.pop();
      Continue;
   Find next splitting attribute A for u
   If |u_{A=0}| > |u_{A=1}| and u_{A=0} not processed before // u_{A=1} is
   the
     set of queries from u that satisfy A=1 (similar for u_{A=0})
      V.push (u_{A=0}) //dfs to create tree for left child.
   Else if u_{A=1} not processed before
           V.push (u_{A=1}) //dfs to create tree for right child
Return assignment for C
```

Fig 2. Pseudocode of *SigTreeSTD* exact algorithm for *STD* (Miah et al. 2016)

Miah et al. (2016) also proposed an approximation algorithm *HeuristicSTD* for *STD*, the pseudocode is shown in Figure 3.

```
Approx Algorithm: HeuristicSTD
Let Q be the query log, A (a_1…a_M) be the attributes in Q
Complement the query log (~Q)   // convert 1 to 0 and 0 to
1, also convert conjunctive form to disjunctive form
For (int i = 1 to M)
    If ~Q not empty
       Count # of queries satisfied both for a_i = 1 and a_i =
     0.
       Assign the value of a_i that gives the minimum count
       Remove queries from ~Q satisfied by the value of a_i
Return the attributes assignment
```

Fig 3. Pseudocode of *HeuristcSTD:* approximation algorithm for *STD* (Miah et al. 2016)

We consider two variants for *STD*: one using the exact or optimal *STD* algorithm, and the other using the heuristic of *STD* (Miah et al. 2016). In particular, the algorithms are as follows:

a) Apply algorithm for *STD* and add solution assignment (tuple) to result.

b) Remove the queries from the query log that are satisfied by assignment in step (a).

c) Repeat steps (a) and (b) until no further queries or attributes are left.

Figure 4 displays the pseudo-code of two approximation algorithms for *MTD*. Note that the way that the idea of greedily removing satisfied queries is inspired by a SETCOVER heuristic (Cormen et al. 2001).

The *SigTreeMTD* algorithm has approximation bound log(S), where S is the number of queries in *Q*, since the *STD* component of the algorithm is exact. There is no approximation bound for *HeuristicMTD*, but it is shown to perform well in Section 6.

```
Algorithm: SigTreeMTD
While Q not empty do   // Q is the query log
   Apply algorithm SigtreeSTD on Q
   Remove queries from Q those satisfy the assignment

Algorithm: HeuristicMTD
While Q not empty do
   Apply algorithm HeuristicSTD on Q
   Remove queries from Q those satisfy the assignment
```

Fig 24. Pseudocode of *SigTreeMTD* and *HeuristicMTD* approximation algorithms for *MTD*

6. EXPERIMENTS

The main performance indicators are (a) the time cost of exact and approximation algorithms, and (b) the approximation quality of approximation algorithms.

6.1 System Configuration and Datasets

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: Datasets of products and product queries are used. Note that products are just one of the possible instantiations of the more general packages of this paper. We used real and synthetic datasets (query logs). In specific, we use two datasets: (i) REAL: real query log, and (ii) REAL+: synthetic query log generated from the real query log.

Real query log (REAL): 237 queries collected for cell phones from university users and friends through an online survey. The survey was designed with 30 Boolean features such as Bluetooth, Wi-Fi, Camera, Speakerphone and so on. Users were asked to select the features they prefer to have (positive) and most likely not to have (negative) in their cell phones. Users selected 3-6 positive and 1-2 negative features on average. Hard disk was a popular negative feature.

Synthetic query log generated from real query log (REAL+): As the real query log is very small, it is inappropriate for scalability experiments. So larger datasets were generated from the real query log. A total of 251,575 queries were generated as follows: at each step we randomly select a query from the REAL query log, randomly select two of its attributes and swap their values. We also generate datasets for a fixed size of query log for varying number of attributes (10, 15, 20, 25, 30).

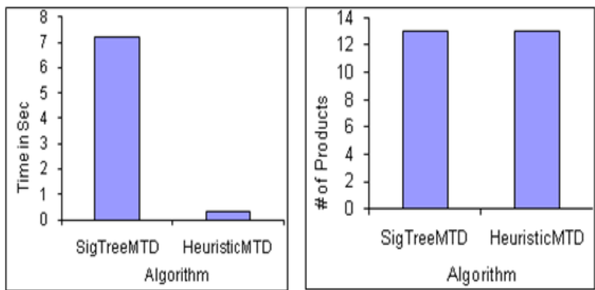
Table 2 summarizes the query logs or datasets.

Table 2. Summary of Query Logs (Datasets)

Query log	*# of attributes*	*Query log size*
REAL	30	237
REAL+_30	30	25K, 50K, …, 200K
REAL+_1000	10, 15, …, 30	1000

6.2 Experimental Results

Figure 5 shows the performance and quality of the algorithms for the REAL dataset. Figures 6 and 7 show the performance of the algorithms for varying query log size and number of attributes respectively, for REAL+ dataset. As we can see from the graphs, the *HeuristicMTD* algorithm is much more efficient than the *SigTreeMTD* which is developed based on exact algorithm of *STD*. The missing data in Figure 6 is due to the very slow speed of this algorithm for large datasets. The running time of *SigTreeMTD* algorithm increases exponentially as the number of total attributes increases.



(a) Time cost (b) Quality

Fig 5. Time cost and Quality for REAL dataset

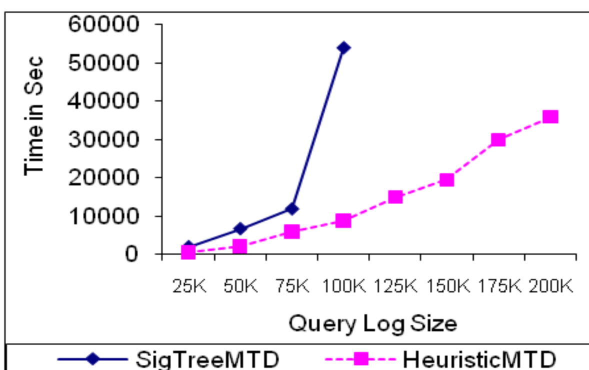


Fig 6. Time cost for varying query log size for REAL+_30

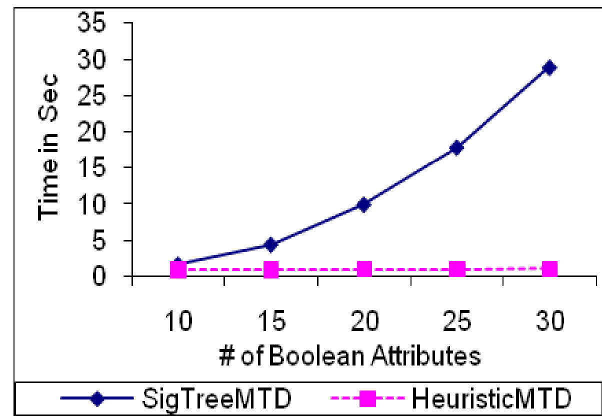


Fig 7. Time cost for varying # of attributes for REAL+_1000

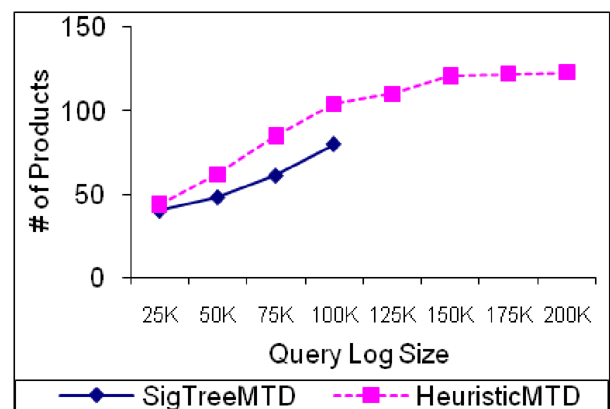


Fig 8. Quality for varying query log size for REAL+_30

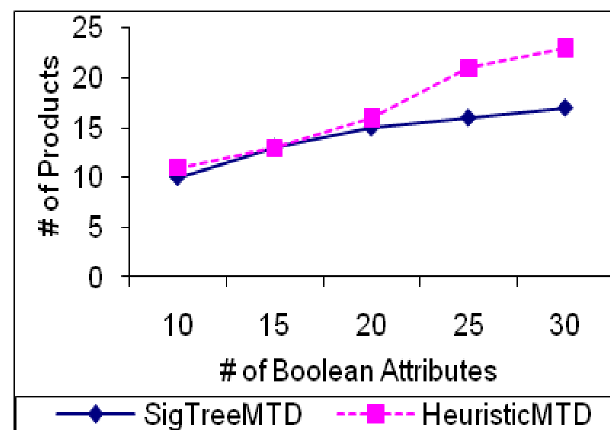


Fig 9. Quality for varying # of attributes for REAL+_1000

Figures 8 and 9 show the quality (number of products need to design to satisfy all queries in the query log) of the approximation algorithm for varying query log size and number of attributes respectively for REAL+. For the same reason as in Figure 6, Figure 9 also has missing data for *SigTreeMTD* algorithm.

7. EXTENSION TO OTHER TYPES OF DATABASES

The Boolean problems discussed above can be extended to categorical and numerical databases as well.

7.1 Problem Framework

Categorical: Can be converted into a set of Boolean attributes according to its distinct categorical values.

Numeric: Comprises of a subset of attributes taking numeric values, while the remaining attributes for which values are not specified are assumed to be "don't care" (represented as "?"), i.e., in the newly designed tuple the numeric attribute can have any value between the range of values it can take. For e.g, query {a1=10, a3=5.2} is equivalent to "a1=10 and a3=5.2", while the remaining attributes are ?.

7.2 Complexity Results

Corollary 1: *MTD* is NP-complete for categorical databases. It is easy to see that the reduction from *GC* to the Boolean *MTD* instance can be readily extended to reduce *GC* to the categorical *MTD* instance by having as many edges as the number of distinct categorical values an attribute can take.

Corollary 2: *MTD* is NP-complete for numeric databases. The proof of NP-completeness for numeric databases follows from Corollary 1 where the distinct categorical attribute values in the graph edges are replaced by numeric values from the range of values that numeric attribute can have.

7.3 Numeric *SigTreeMTD* Algorithm

The Boolean version of the algorithm a_i extended as follows. For each attribute, let $R(a_i)=r_1,..,r_s$ be the set of range conditions specified for a_i in the query log Q. Let $E(a_i)$ be the list of endpoints of the ranges in $R(a_i)$, in ascending order. Then, we define $I(a_i)$ as the list of intervals created from $E(a_i)$, one interval for each two consecutive endpoints. E.g. if $r_1=(2,3)$, $r_2=(3,5)$, $r_3=(4,7)$, then $E(a_i)=\{2,3,4,5,7\}$ and $I(a_i)=\{[2,3], [3,4], [4,5], [5,7]\}$. Note that $|E(a_i)| \leq 2\times|Q|$ and $|I(a_i)| \leq 2\times|Q|-1$, where $|Q|$ is the number of queries in Q. Then, each attribute has $|I(a_i)|$ children, denoting all possible intervals for a_i. Each path from root to a node u can be viewed as a list of intervals, one for each attribute along the path. To compute the number of queries for u, we make a pass on Q and count the number of queries that satisfy all the conditions along the path from root to u. The rest of the algorithm and pruning conditions are the same as the Boolean version, where instead of having only two values 0,1 to choose from at each step, we choose among the intervals in $|I(ai)|$.

8. RELATED WORK

The most related work to work of this paper is the the maximal reverse problem or *Single Tuple Design (STD) Problem:* Given a database (set of tuples) and a query log, the task is to construct a new tuple such that the set of queries in the query log that return the new tuple is larger than for any other tuple in the database (Miah et al. 2016). However, this paper focus on an interesting variant of the Single Tuple Design problem. Instead of just designing a single tuple, we are interested in creating a minimum number of tuples that collectively satisfy all queries in the query log. This is referred to as the *Multiple Tuple Design (MTD) problem.*

There is been recent interest in the area of Reverse Query Processing for various query models such as *k*NN, Top-*k*, skyline, and so on (Binning et al. 1994, 2007; Dellis and Seeger 2007; Korn and Muthukrishnan 2000; Vlachou et al. 2010). Unlike traditional query processing, the applications are not from a customer's point of view, but rather are from the manufacturer's point of view, i.e., of how to determine the set of customers that find a particular product appealing. Reverse Nearest Neighbor (*RNN*) queries were first investigated by Korn and Muthukrishnan (2000). Given any query point *q*, Reverse NN is to determine the set *RNN*(*q*) of reverse nearest neighbors. Reverse Skyline Queries (Dellis and Seeger 2007) considers for a multidimensional data set *D* the problem of dynamic skyline queries according to a query point *q*. This kind of dynamic skyline corresponds to the skyline of a transformed data space where point *q* becomes the origin and all points of *D* are represented by their distance vector to *q*. The reverse skyline query returns the objects whose dynamic skyline contains the query object *q*. Recent work on Reverse top-*k* queries (Vlachou et al. 2010) is from the perspective of the product manufacturer. The problem is, given a potential product, which are the user preferences for which this product is in the top-*k* query result?

The work in this paper is different than all the works on reverse query processing discussed above. We are not given the set of data tuples to pick from, instead we have to design a set of new tuples (*MTD*) that satisfy maximum number of queries in the given query log. In this regard, our STD problem is somewhat similar to maximal reverse query processing problems that has recently received some attention for skyline (Li et al. 2007; Li et al. 2006) and *k*NN (Cabello et al. 2005; Wong et al. 2009) queries with numeric attributes. Our work is different from these because we consider selection queries over Boolean attributes.

Works on dominant relationship (Li et al. 2006) and dominating neighborhood (Li et al. 2007) uses skyline query semantics assuming that attributes are min/max, that is, all users have the same preference for an attribute (e.g., 2 doors is always better than 4 doors). Further, they assume there is a profitability plane which simplifies the algorithm given that the optimal solution is a point on the profitability plane. In contrast, in work of this paper along with the work of Miah et al. (2016) users may have opposite preferences for the same attribute, and the algorithms can be used with or without a profitability plane.

Miah et al. (2009) tackled another related problem of maximizing the visibility of an existing object by selecting a subset of its attributes to be advertised. The main problem was: given a query log with conjunctive query semantics and a new tuple, select a subset of attributes to retain for the new tuple so that it will be retrieved by the maximum number of queries. The work did not consider negated conditions as in the work of this paper.

Optimal product design or positioning is a well-studied problem in Operations Research and Marketing. Shocker and Srinivasan (1974) first represented products and consumer preferences as points in a joint attribute space. After that, several approaches and algorithms (Albers and Brockhoff 1977; Albers and Brockhoff 1980; Albritton and McMullen 2007; Gavish et al. 1983; Gruca and Klemz 2003; Kohli and Krishnamurti 1989) have been developed to design/position a new product. Works in this domain require direct involvement (one or two step) of consumers and users are usually shown a set of existing alternative products (predesigned) to choose or set preferences. Users in this domain in fact do not get to select the attributes or features they like and do not like. Instead of involving users directly in the process of designing new products, this paper uses previous user search queries to model user preferences, since it is easy to collect the preferences (search queries) for large number of Internet users

nowadays. This paper also consider large query logs to design the new set of products and allow users to express their interests in attribute or feature level in terms of positive, negative and "don't care".

The *MTD* problem can be viewed as the segmentation problem (Kleinberg et al. 1998) for the *STD* problem (Miah et al. 2016). However, in *MTD* the size of each segment is not given.

9. CONCLUSIONS

This paper investigated the problem of designing smallest set of tuples for maximal reverse selection queries - given a set of selection queries with conjunctive conditions (where a query can have negations), create the smallest set of tuples that that collectively satisfy all queries in the query log. The problem has several natural applications, such as designing best vacation packages, designing new products, and so on. The paper shows the difference of the proposed problem from the existing techniques in various fields such as marketing, product design, operation research, query processing, etc. The paper considers several interesting variants of the problem as well as various types of databases such as Boolean, categorical, and numerical. It proves intractability results, and provide approximation algorithms, some of which are shown to work well in practice. A future direction is to extend the problem to develop more scalable algorithms for categorical, numeric, and possibly text data and different query semantics such as top-*k* and skyline queries.

10. REFERENCES

[1] Albers, S., and Brockhoff, K. "A procedure for new product positioning in an attribute space", European Journal of Operational Research, 1, 4 (Jul 1977), 230-238.

[2] Albers, S., and Brockhoff, K. "Optimal Product Attributes in Single Choice Models", Journal of the Operational Research Society (1980) 31, 647–655.

[3] Albritton, D. M., and McMullen P. R. "Optimal product design using a colony of virtual ants", European Journal of Operational Research, 176, 1 (Jan 2007), 498-520.

[4] Cabello, S., Diaz-Banez, J. M., Langerman, S., Seara, C., and Ventura, I. 2005. "Reverse facility location problems", Canadian Conference on Computational Geometry.

[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2001. Introduction to Algorithms, The MIT Press.

[6] Dellis, E., and Seeger, B. 2007. "Efficient computation of reverse skyline queries", VLDB.

[7] Garey, M. R., and Johnson, D. S. 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness, New York, W.H. Freeman, ISBN 0-7167-1045-5.

[8] Gavish, B., Horsky, D., and Srikanth, K. 1983. "An Approach to the Optimal Positioning of a New Product", Management Science, 29, 11, 1277-1297.

[9] Gruca, T. S., and Klemz, B. R. 2003. "Optimal new product positioning: A genetic algorithm approach", European J. of Operational Research, 146, 3, 621-633.

[10] Kleinberg, J., Papadimitriou, C., and Raghavan, P.1998. "Segmentation Problems", ACM Symposium on the Theory of Computing, 473-482.

[11] Kohli, R., Krishnamurti, R. 1989. "Optimal product design using conjoint analysis: Computational complexity and algorithms", European Journal of Operational Research, 40,186–195.

[12] Korn, F., Muthukrishnan, S. 2000. "Influence sets based on reverse nearest neighbor queries", SIGMOD.

[13] Li, C., Tung, A. K. H., Jin, W., and Ester, M. 2007. "On Dominating Your Neighborhood Profitably", VLDB, 818-829.

[14] Li, C., Ooi, B. C., Tung, A. K. H., Wang, S. 2006. "DADA: a Data Cube for Dominant Relationship Analysis", SIGMOD.

[15] Miah, M., Das, G., Hristidis, V., and Mannila, H. 2009. "Determining Attributes to Maximize Visibility of Objects", IEEE Transactions on Knowledge and Data Engineering (TKDE) vol. 21 no. 7, pp. 959-973.

[16] Miah, M., Omar, A. 2016. The International Academy of Business and Public Administration Disciplines (IABPAD) Conference Proceedings.

[17] Shocker, A. D., and Shrinivasan, V. 1974. "A consumer-based methodology for the identification of new product ideas", Management Science, 20, 6, 921-937.

[18] Vlachou, A., Doulkeridis, C., Kotidis, Y., and Norvag, K., 2010. "Reverse Top-k Queries", ICDE.

[19] Wong, R. C-W., Özsu, M. T., Yu, P. S., Fu, A. W-C., and Liu, L. 2009. "Efficient method for maximizing bichromatic reverse nearest neighbor", VLDB.