ATTRIBUTES SELECTION AND PACKAGE DESIGN TO MAXIMIZE VISIBILITY OF OBJECTS

by

MD ZAHIDUZZAMAN MIAH

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2009

ACKNOWLEDGEMENTS

First of all I would like to thank and praise Allah for all the accomplishments I have achieved in my life.

I would like to express profound gratitude to my supervisor, Dr. Gautam Das, for his invaluable support, encouragement, supervision and constant guidance throughout this research, and for giving me a wonderful opportunity to work on such challenging projects as well as giving me extraordinary experiences through out the work. Above all and the most needed, he provided me unflinching encouragement and support in various ways. His truly scientist and academic intuition exceptionally inspired and enriched my growth as a student, a researcher and a scientist want to be.

I gratefully thank my supervising committee members Dr. Sharma Chakravarthy, Dr. Ramez Elmasri and Dr. Leonidas Fegaras for serving on my committee and providing me valuable suggestions throughout this research work.

I would also like to thank my research collaborators Dr. Vagelis Hristidis of Florida International University and Dr. Heikki Mannila of University of Helsinki.

I thank my family and friends for their support. My heartiest gratitude goes to the soul of my late father, Abdur Rouf Miah, who always gave me the inspiration and taught me to never give up. May Allah bless his departed soul. And finally I would like to thank my wife, Simanaz Happy, without her there are many things in my life would not be possible.

October 23, 2009

ABSTRACT

ATTRIBUTES SELECTION AND PACKAGE DESIGN TO MAXIMIZE VISIBILITY OF OBJECTS

Md Zahiduzzaman Miah, PhD

The University of Texas at Arlington, 2009

Supervising Professor: Gautam Das

In recent years, there has been significant interest in the development of ranking functions and efficient top-*k* retrieval algorithms to help users in ad-hoc search and retrieval in databases (e.g., buyers searching for products in a catalog). We introduce a complementary problem: how to guide a seller in selecting the best attributes of a new tuple (e.g., a new product) to highlight so that it stands out in the crowd of existing competitive products and is widely visible to the pool of potential buyers. For example, assume one wants to sell an iPod in e-commerce site, but the title allows only 12 characters of space, should he write "Purple iPod", "Apple iPod" or "iPod 30g"? Which title is good? Do people care more about color, brand or size? We refer this problem as *"attributes selection"* problem. Package design based on user input is a problem that has also attracted recent interest. Given a set of elements, and a set of user preferences (where each preference is a conjunction of positive or negative preferences for individual elements), we investigate the problem of designing the most "popular package", i.e., a subset of the elements that maximizes the number of satisfied users. Numerous instances of this problem occur in practice. For example, a vacation package consisting of a subset of all possible activities may need to be assembled, that satisfies as many potential customers as possible, where each potential customer may have expressed his preferences

(positive or negative) for certain activities. Likewise, selecting topics for a social network group based on users' preferences as well as the problem of designing new products, i.e., deciding which features to add to a new product (e.g., to an iPod) that satisfies as many potential customers as possible, also falls under this framework. We refer this later problem as *"package design"* problem. We develop several formulations of both the problems. Even for the NP-complete problems, we give several exact (optimal) and approximation algorithms that work well in practice. For *"attributes selection"* problem, one type of exact algorithms is based on Integer Programming (IP) formulations. Another class of exact methods is based on Maximal Frequent Itemset mining algorithms. For *"package design"* problem, the exact algorithm is based on Signature Tree data structure that is feasible for relatively moderate problem instances. Further, we present approximate algorithms, and study the performance, both theoretically and experimentally. Our experimental evaluation on real and synthetic datasets shows that the optimal and approximate algorithms are efficient for moderate and large datasets respectively, and also that the approximate algorithms have small approximation error.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

CHAPTER 1

INTRODUCTION

In recent years, there has been significant interest in developing effective techniques for ad-hoc search and retrieval in unstructured as well as structured data repositories, such as text collections and relational databases. In particular, a large number of emerging applications require exploratory querying on such databases; examples include users wishing to search databases and catalogs of products such as homes, cars, cameras, restaurants, or articles such as news and job ads. Users browsing these databases typically execute search queries via public front-end interfaces to these databases. Typical queries may specify sets of keywords in case of text databases, or the desired values of certain attributes in case of structured relational databases. The query-answering system answers such queries by either returning all data objects that satisfy the query conditions, or may rank and return the top-*k* data objects, or return the results that are on the query's skyline. If ranking is employed, the ranking may either be simplistic – e.g., objects are ranked by an attribute such as Price; or more sophisticated – e.g., objects may be ranked by the degree of "relevance" to the query. While unranked retrieval (also known as *Boolean Retrieval*) is more common in traditional SQL-based database systems, ranked retrieval (also known as *Top-k Retrieval*) is more common in text databases, e.g. tf-idf ranking [40]. Recently there has been widespread interest in developing suitable top-*k* retrieval techniques even for structured databases [1, 11, 44]. Skyline retrieval semantics is also investigated where a data point is retrieved by a query if it is not dominated by any other data point in all dimensions [7, 32, 36, 38, 41, 45].

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

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

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

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

> *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 work we initiate an investigation of this novel optimization problem. We consider several variants, including Boolean, categorical, text and numeric data, and conjunctive and disjunctive query semantics. We also consider variants in which the "budget", i.e., *m*, is not specified; in this case our objective is to determine the value of *m* such that the number of

satisfied queries *divided by m* is maximized. Thus we seek to maximize the "per dollar" benefit. A special case of this no-budget variant is when ranking is performed using functions that are non-monotone on the number of specified attributes (keywords), such as the BM25 [39] scoring function used in Information Retrieval.

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

In addition to *"attributes selection"* problem, in this research we also investigate the problem of designing popular packages to maximize visibility of objects which we discuss next.

Consider a travel agency that wishes to design one (or more) vacation packages, given the travel preferences of its clients. For example, a vacation package to Costa Rica can include some of the following elements: 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 element. The purpose of trip/vacation package design is to select a subset of these elements to satisfy as many customers as possible.

As another example, consider the problem of creating a social network and selecting the main topics of the network based on users' interests, with the goal of representing the collective group interests as optimally as possible. For example, assume one wants to create a new group focused on sports interests, and a user's profile specifies a positive or negative

preference for each of a set of topics (*Basketball*, *Soccer*, *Baseball*, etc.). One can leverage the users' profiles to select the main topic preferences of the network– *e.g., Basketball*, *Baseball*–of the users.

The above examples can be generalized to an abstract problem, which we call the *Package Design Problem*. Assume that a package needs to be designed by selecting a subset of Boolean features (or elements, or attributes) from a large set of possible features. In particular, we focus on a specific and novel problem formulation, where 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, and we are asked to design the most *popular* package, i.e., the package that satisfies the maximum number of queries in the query log. We refer to this problem as the *Single Package Design* (*SPD*) problem. Because of the vast use of the Internet nowadays, it is very easy to collect online such query logs of user preferences for many such package design applications, and the new package can be designed based on real users' perception on desirable features.

As an interesting variant to *SPD*, instead of just designing a single package, we may also be interested in creating a minimum number of packages that collectively satisfy *all* queries in the query log. We refer to this as the *Multiple Package Design* (*MPD*) problem. For example, a tourist 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.

We note that in many real applications the package design problem may be more complex than as described above. There may be dependencies between attributes (e.g., if *fishing* is selected, then a *lake* also has to be selected). Moreover, for certain applications a profitability function must also be taken into consideration, e.g., in a product design application such as designing a new cell phone, the manufacturer may have to decide on features such as *ring tone*, *speed dialing*, *weight*, *size*, *camera*, etc., based on customers' preferences. In this

work we mainly focus on the *SPD* and *MPD* problems as described earlier, but include a brief discussion of these more complex problem variants in Section 3.6. Moreover, although we consider Boolean attributes where the value of the attribute is either 0 (feature is not present) or 1 (the feature is present), our problems can be generalized to features of any type such as categorical, text, and numeric.

Our package design problems are NP-complete, and it is thus challenging to design good approximate algorithms for them. Although at first sight the problems appear similar to other well-known NP-complete problems, such as Maximum Satisfiability (*MAXSAT*) [27], Minimum Satisfiability (*MINSAT*) [30], and Set Cover (*SETCOVER*) [15], none of these problems are an exact fit for the package design problems, and thus we cannot simply reuse known approximate algorithms for these well-known problems.

For example, *MAXSAT* is not very useful because its goal is to maximize the number of satisfied constraints (queries), where each constraint is a disjunction of literals, whereas each constraint (query) in *SPD* is a conjunction of literals (elements), given that we follow conjunctive query semantics. In contrast, it is easy to reduce *MINSAT* to *SPD*; however as will be shown later, the well-known approximate algorithms for *MINSAT* cannot be used for *SPD* to achieve similar approximation bounds. Likewise, we shall show that although *MPD* is seemingly similar to *SETCOVER*, there are important difference which makes designing algorithms for *MPD* even more challenging – in *SETCOVER*, the candidate sets are given, whereas in *MPD* the number of candidate sets can be exponential in the number of features since any combination of features is a candidate set.

Even though the *SPD* problem is NP-complete, we propose an optimal branch-and-bound algorithm, based on a novel adaptation of the *Signature Tree* data structure, which interestingly was originally developed for a different context, i.e., for effective indexing of Boolean signatures [12. In our adaptation, the signature tree is designed to leverage the specific nature of the *SPD* problem, and employs smart branching and pruning techniques to

minimize the execution cost. We also provide two approximate algorithms for the *SPD* problem. The first has a provable approximation bound, but does not scale to very large instances in practice as it is memory-based. The second does not have provable bounds, but is scalable and is shown to work very well in practice.

For the *MPD* problem, we develop an approximate algorithm with a provable approximate bound, by combining known approximate algorithms (and their approximation factors) of two separate NP-complete problems. We also develop scalable approximate algorithms for *MPD*.

In summary, we define the problem of designing optimal packages given user preferences, expressed as positive and negative preferences on the elements. We also show that the problem variants are NP-complete. We adapt existing theoretical results and approximate algorithms from NP-complete problems where possible, and provide new algorithms and theoretical analysis. We present a feasible optimal (exact) algorithm based on the Signature Tree data structure, by employing smart pruning. We present fast approximate algorithms that work well in practice for large problem instances. We perform detailed performance evaluations on real and synthetic data to demonstrate the effectiveness of our developed algorithms.

CHAPTER 2

ATTRIBUTES SELECTION

2.1 Preliminaries

First we provide some useful definitions.

Boolean Database: Let $D = \{t_1 \ldots t_N\}$ be a collection of Boolean tuples over the attribute set $A = \{a_1 \ldots a_M\}$, where each tuple t is a bit-vector where a 0 implies the absence of a feature and a 1 implies the presence of a feature. A tuple t may also be considered as a subset of A, where an attribute belongs to t if its value in the bit-vector is 1.

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

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

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

The following running example will be used throughout the *"attributes selection"* problem to illustrate various concepts.

EXAMPLE 1: *Consider an inventory database of an auto dealer, which contains a single database table D with N rows and M attributes where each tuple represents a car for sale. The table has numerous attributes that describe details of the car: Boolean attributes such as AC, Four Door, Turbo, Power Doors, Auto Trans, Power Brakes, etc; categorical attributes such as Make, Color, Engine Type, Zip Code, etc; numeric attributes such as Price, Age, Fuel Mileage, etc; and text attributes such as Reviews, Accident History, and so on. Figure 2.1 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. □

2.2 Main Problem Variant: Conjunctive Boolean with Query Log

In this section we formally define the main problem for Boolean data. As we discuss in Section 2.4, many other variants can be reduced to this problem. In Section 2.2.1 we formally define the problem and in Section 2.2.2 we present our complexity results. In Section 2.3 we provide algorithms for this problem variant. We first defined the query semantics to be used in the beginning.

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

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

2.2.1. Problem Definition

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

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

EXAMPLE 1 (cont'd): *To illustrate this problem variant, consider the example in Figure 2.1, which shows a new tuple t that needs to be inserted into the database. Suppose we are required to retain m = 3 attributes. It is not hard to see that if we retain the attributes AC, Four*

Door, and Power Doors (i.e., t' = [1, 1, 0, 1, 0, 0]), we can satisfy a maximum of three queries (q1, q2, and q3). No other selection of three attributes of the new tuple will satisfy more queries.

□

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

Notice that in *CB-QL*, it is the query log *Q* that needs to be analyzed in solving the problem; the actual database *D* (i.e., the "competing products") is irrelevant. We consider variants that involve the products database in Section 2.4.

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

Figure 2.1 Illustrating EXAMPLE 1

2.2.2. NP-Completeness Results

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

Proof: An instance of decision version of CB-QL is similar to an instance of CB-QL, except that it has in addition a target parameter X, and is satisfied if there is a compressed tuple t' having m attributes such that the number of queries that retrieve t' is at least X. Clearly, the decision version of CB-QL is in NP. To prove that it is NP-complete, we reduce from the Clique problem. Given a graph G = (V, E) and an integer r, the task in the Clique problem is to check if there is a clique of size r in G. We transform this to an instance of decision version of CB-QL as follows. The attribute set A will be V, and the query log will contain one row for each edge. If e = (u, v) is an edge, then the query log Q contains the conjunctive query {u, v}, i.e., the query retrieving all tuples with u = 1 and v = 1. The new tuple t has all the attributes in V set to 1. Let m = r and X = m(m-1)/2. Now t has a compressed representation with m attributes that satisfies X number of queries if and only if the graph has a clique of size r. □

2.3 Algorithms for Conjunctive Boolean with Query Log

In this section we discuss our main algorithmic results for the main problem variant discussed in Section 2.2.

2.3.1. Optimal Brute Force Algorithm

Clearly, since *CB-QL* is NP-hard, it is unlikely that any optimal algorithm will run in polynomial time in the worst case. The problem can be obviously solved by a simple brute force algorithm (henceforth called *BruteForce-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.

2.3.2. Optimal Algorithm based on Integer Linear Programming

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

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

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

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

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

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

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

2.3.3. Optimal Algorithm based on Maximal Frequent Itemsets

The algorithm based on Integer Linear Programming described in the previous subsection has certain limitations; it is impractical for problem instances beyond a few hundred queries in the query log. The reason is that it is a very generic method for solving arbitrary integer linear programming formulations, and consequently fails to leverage the specific nature

of our problem. In this subsection we develop an alternate approach that scales very well to large query logs. This algorithm, called *MaxFreqItemSets-CB-QL*, is based on an interesting adaptation of an algorithm for mining *Maximal Frequent Itemsets* [20].

We first define the frequent itemset problem:

2.3.1.1. The Frequent Itemset Problem

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

Computing frequent itemsets is a well studied problem and there are several scalable algorithms that work well when *R* is sparse and the threshold is suitably large. Examples of such algorithms include [2, 22]. In our case, given a new tuple *t*, recall that our task is to compute *t'*, a compression of *t* by retaining only *m* attributes, such that the number of queries that satisfy *t'* is maximized. This immediately suggests that we may be able to leverage algorithms for frequent itemsets mining over *Q* for this purpose. However, there are several important complications that need to be overcome, which we elaborate next.

2.3.1.2. Complementing the Query Log

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

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

However, the problem is that *Q* is itself a sparse table, as the queries in most search applications involve the specification of just a few attributes. Consequently, the complement ~*Q* is an extremely dense table, and this prevents most frequent itemset algorithms from being directly applicable to ~*Q*. For example, most "level-wise algorithms" (such as Apriori [2], which operates level by level of the Boolean lattice over the attributes set by first computing the single itemsets, then itemsets of size 2, and so on) will only progress past just a few initial levels before being overcome by an intractable explosion in the size of candidate sets. To see this, consider a table with *M* = 50 attributes, and let *m* = 10. To determine a compressed tuple *t'* with 10 attributes, we need to know the itemset of ~*Q* of size 40 with maximum frequency. Due to the dense nature of ~*Q*, algorithms such as Apriori will not be able to compute frequent itemsets beyond a size of 5-10 at the most. Likewise, the sheer number of frequent itemsets will also prevent other algorithms such as FP-Tree [22] from being effective.

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

2.3.1.3. Setting of the Threshold Parameter

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

There are two alternate approaches to setting the threshold. One approach is essentially a heuristic, where we set the threshold to a reasonable fixed value dictated by the

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

We also suggest an alternate adaptive procedure of setting the threshold that is guaranteed to find the optimal compression. First initialize the threshold to a high value and compute the frequent itemsets of $\sim Q$. 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 which is half of the previous threshold. This process is guaranteed to discover the optimal t'.

We now return to the task of how to compute frequent itemsets of the dense Boolean table $\sim Q$. 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 $\sim Q$.

2.3.1.4. Random Walk to Compute Maximal Frequent Itemsets

A maximal frequent itemset is a frequent itemset such that none of its supersets are frequent. The set of maximal frequent itemsets are much smaller than the set of all frequent itemsets. For example, if we have a dense table with M attributes, then it is quite likely that most of the maximal frequent itemsets will exist very high up in the Boolean lattice over the attributes, very close to the highest possible level M. Figure 2.2 shows a conceptual diagram of a Boolean

lattice over a dense Boolean table ~Q. The shaded region depicts the frequent itemsets and the maximal frequent itemsets are located at the highest positions of the border between the frequent and infrequent itemsets.

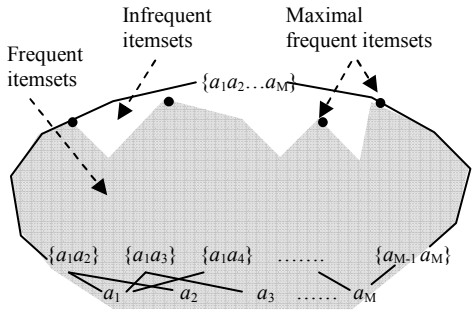


Figure 2.2 Maximal frequent itemsets in a Boolean Lattice

There exist several algorithms for computing maximal frequent itemsets, e.g. [6, 8, 18, 20]. We base our approach on the *random walk* based algorithm in [20], which starts from a random singleton itemset *I* at the bottom of the lattice, and at each iteration, adds a random item to *I* (from among all items *A* - *I* such that *I* remains frequent), until no further additions are possible. At this point a maximal frequent itemset *I* has been discovered. If the number of maximal frequent itemsets is relatively small, this is a practical algorithm: repeating this random walk a reasonable number of times will with high probability discover all maximal frequent itemsets. However, since this algorithm is based on traversing the lattice from bottom to top, it implies that the random walk will have to traverse a lot of levels before it reaches a maximal frequent itemset of a dense table.

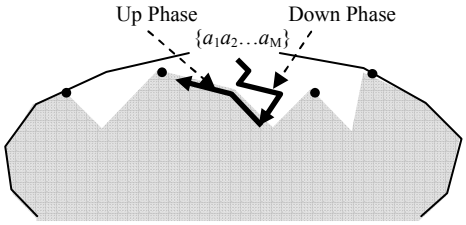


Figure 2.3 Two phase random walk

Instead, we propose an alternate approach which starts from the top of the lattice and traverses down. Our random walk can be divided into two phases: (a) *Down Phase*: starting from the top of the lattice ($I = \{a_1a_2\ldots a_M\}$), 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. At this point a maximal frequent itemset *I* has been discovered.

Figure 2.3 shows an example of the two phases of the random walk. What is important to note is that this process is much more efficient than a bottom-up traversal, as our walks are always confined to the top region of the lattice and we never have to traverse too many levels. Complementing the query log eventually results in a dense dataset. In a dense dataset, maximal frequent itemsets are usually at the top of the lattice. That is, they are close to level *M* - *m,* where *m* is comparatively much smaller than *M*. If a bottom-up approach is used to find maximal frequent itemsets, it will have to traverse a long portion of the lattice (i.e., too many levels) and will be inefficient. Whereas, in top-down approach, the first phase tries to find the first frequent itemset along the path from the top which is usually close to a maximal frequent itemset, the walks are confined to the top region of the lattice and we never have to traverse too many levels.

2.3.1.5. Complexity Analysis of a Random Walk Sequence

In the worst case, the cost of a down-up random walk is $2 \cdot M \cdot |Q|$, where *M* is the total number of attributes and $|Q|$ is the size of the query log. Although in the worst case the random

walk will go up and down the whole lattice, in practice we only expect each portion of the walk to traverse only a few levels at the top of the lattice.

2.3.1.6. Number of Iterations

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

2.3.1.7. Frequent Itemsets at Level *M* – *m*

Finally, once all maximal frequent itemsets have been computed, we have to check which ones are supersets of ~*t*. Then, for all possible subsets (of size *M* – *m*) of each such maximal frequent itemset (see Figure 2.4), we can determine that subset *I* that is (a) a superset of ~*t*, and (b) has the highest frequency. The optimal compressed tuple *t'* is therefore the complement of *I*, i.e., ~*I*.

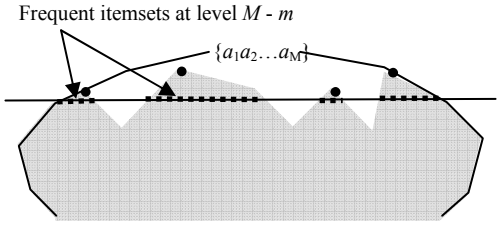


Figure 2.4 Checking frequent itemsets at level *M* - *m*

In summary, the pseudo-code of our algorithm *MaxFreqItemSets-CB-QL* is shown in Figure 2.5. Details of how certain parameters such as the threshold are set, are omitted from the pseudo-code.

```
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-CB-QL(~Q, r)
   ComputeMaxFreqItemsets(~Q, r)
   let Itemsets(t) ← {I |  I ⊆ MaxFreqItemsets,
                          |I| = M – m, and   I ⊇ ~t)}
   let I be the itemset in Itemsets(t) with highest
       frequency
   return ~I
```

Figure 2.5 Pseudo code for algorithm *MaxFreqItemSets-CB-QL*

2.3.1.8. Preprocessing Opportunities

Note that the algorithm also allows for certain operations to be performed in a preprocessing step. For example, all the maximal itemsets can be precomputed, and the only task that needs to be done at runtime is to determine, for a new tuple *t*, those itemsets that are supersets of *~t* and have size *M* – *m*. If we know the range of *m* that is usually requested for

compression in new tuples, we can even precompute all frequent itemsets for those values of *m*, and lookup the itemset with the highest frequency at runtime.

2.3.1.9. The Per-Attribute Variant

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

2.3.4. Greedy Heuristics

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

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

The algorithm *ConsumeAttrCumul-CB-QL* is a cumulative version of *ConsumeAttr-CB-QL*. It first selects the attribute with the highest individual frequency in the query log. It then selects the second attribute that co-occurs most frequently with the first attribute in the query log, and so on.

Instead of consuming attributes greedily, an alternative approach is to consume queries greedily. The algorithm *ConsumeQueries-CB-QL* operates as follows. It first picks the query

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

2.4 Other Problem Variants for Boolean Data

In this section we discuss several other problem variants for Boolean data.

2.4.1. Conjunctive Boolean – Data (CB-D)

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

2.4.1.1. Problem Definition (*CB-D*)

Given a database D, a new tuple t, and an integer m, compute a compressed tuple t' by retaining m attributes such that the number of tuples in D dominated by t' is maximized.

EXAMPLE 1 (cont'd): *To illustrate this problem variant, consider the example in Figure 2.1 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.* □

2.4.1.2. Complexity Results for *CB-D*

The same proof in section 2.2.2 for the problem *CB-QL* obviously tells also that *CB-D* is NP-hard. *CB-D* also has a per-attribute version which can be naturally defined.

2.4.1.3. Algorithms for *CB-D*

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

2.4.2. Top-k - Global Ranking (Tk-GR) and Top-k - Query-Specific Ranking (Tk-QR)

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

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

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

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

2.4.2.1. Problem Definition (*Tk-GR*)

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

2.4.2.2. Problem Definition (*Tk-QR*)

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

EXAMPLE 1 (cont'd): *To illustrate Tk-GR, assume that each query in the query log returns the top-2 tuples (i.e., k = 2) ordered by decreasing Fuel Efficiency, where Fuel Efficiency is a numeric attribute not shown in Figure 2.1. [2]Let us assume that the fuel efficiencies of the seven cars t_1 … t_7 are 10mpg, 20mpg … 70mpg respectively. The results of executing the five queries in the query log on the database are shown in Figure 2.6(a).*

Assume the fuel efficiency of the new tuple t is 35mpg and we are required to retain m = 3 of its Boolean attributes. We first argue that the selections suggested in CB-QL are suboptimal. Suppose we decide to retain the three attributes AC, Four Door, and Power Doors as suggested in CB-QL. The new compressed tuple t' will satisfy the Boolean conditions of queries q_1, q_2, and q_3, however, it will not be returned among the top-2 tuples of q_1 and q_3, because it has a lower score (fuel efficiency) than t_4 (the second tuple returned by both q_1 and q_3). Consequently, the compressed tuple t' will be returned by only one query, q_2 (by replacing t_3 as the second tuple to be returned, since t_3 has lower fuel efficiency than t'). In contrast, if we decide to retain AC, Power Doors and Power Brakes, the compressed tuple t' will satisfy the Boolean conditions of two queries, q_2 and q_4, and moreover, will replace t_3 as the second tuple

[2] *For now, we assume that only Boolean attributes can be specified in queries. Allowing numeric ranges to be specified in queries is discussed in Section 2.6*

to be returned by each query. No other selection of three attributes of the new tuple will ensure that it gets returned by more queries. □

Query ID	Top-2 tuples with scores
q_1	t_5 (50), t_4 (40)
q_2	t_4 (40), t_3 (30)
q_3	t_6 (60), t_4 (40)
q_4	t_4 (40), t_3 (30)
q_5	∅

(a)

Query ID	Top-3 tuples with scores
q_1	t_4 (2), t_5 (2), t_1 (1)
q_2	t_3 (2), t_4 (2), t_1 (1)
q_3	t_1 (2), t_4 (2), t_6 (2)
q_4	t_3 (2), t_4 (2), t_1 (1)
q_5	t_2 (1), t_3 (1), t_7 (1)

(b)

Figure 2.6 Results of Top-*k* Retrieval - (a) Global scoring function, (b) Query specific scoring function (dot product)

We now discuss *Tk-QR*.

EXAMPLE 1 (cont'd): *Assume that each query in the query log returns the top-3 tuples (i.e., k = 3), where the query-specific scoring function is the dot product between a query and a tuple. Recall that in this case a query is no longer a conjunctive query – tuples that do not dominate a query may also be returned. Based on this scoring function, the results of the execution of the five queries are shown in Figure 2.6(b) (score ties have been broken arbitrarily). Suppose we are required to retain m = 4 attributes. It is not hard to see that if we retain the attributes AC, Four Door, Power Doors and Power Brakes, the compressed tuple will definitely enter into the top-3 result tuples of q_1, q_2 and q_4 respectively (it will also tie scores with the top-3 tuples of q_3). No other selection of three attributes of the new tuple will ensure that it gets returned by more queries.* □

Monotonicity of Scoring Functions: One final issue regarding *Tk-QR* needs to be discussed. In this variant, we have assumed that the scoring function is monotonic, i.e., that if *t'* is a compressed representation of *t*, then the score of *t'* is no greater than the score of *t*. This is certainly true of the dot product. Monotonic scoring functions imply that it is to our advantage to

retain as many attributes as possible (constrained by the budget, m). However, not all scoring functions are monotonic, as we see in Section 2.7, when we discuss an application in text databases. In such case, the optimal number of attributes to retain may even be less than the budget m.

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

2.4.2.3. Complexity Results for *Tk-GR* and *Tk-QR*

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

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

2.4.2.4. Algorithms for *Tk-GR* and *Tk-QR*

Fortunately, due to the scoring function being a global function, we can reduce *Tk-GR* to *CB-QL* as follows, and use any of the algorithms developed in Section 2.3. We first execute each query q in Q. Let the k^{th} largest score of the returned tuples be s_q. Let the score of the new tuple be s_t. Let Q' be the subset of Q such that for each query q in Q', s_q is no larger than s_t. Then it is easy to see that all we need to solve *CB-QL* for the reduced query log Q', because if the compressed tuple t' satisfies a query q in Q', it will be definitely retuned as part of the top-k tuples of q.

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

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

However, we can develop several effective greedy algorithms for *Tk-QR*, depending on whether we consume attributes or queries cumulatively or non-cumulatively (as Section 2.3.4). The straightforward details are omitted.

2.4.3. Skyline Boolean (SB)

We consider skyline retrieval semantics for this problem. Given a set of points, the skyline comprises the points that are not dominated by other points. A point dominates another point if it is as good or better in all dimensions and better in at least one dimension [45]. We consider skyline for Boolean data in our problem, but to get a clear picture let consider a common example in the literature, *"choosing a set of hotels that is closer to the beach and cheaper than any other hotel in distance and price attributes respectively from the database system of the travel agents"* [7]. Figure 2.7 illustrates this case in 2-*D* space, where each point corresponds to a hotel record. The *x*-axis and *y*-axis specify the room price of a hotel and its distance to the beach respectively. Clearly, the most interesting hotels are {*a, g, i, n*}, called *skyline*, for which there is no any other hotel in {*a, b, . . . , m, n*} that is better on both dimensions. As mentioned earlier, we mainly consider Boolean skylines (skylines with Boolean data), where all the attributes asked by a query need not to be present in the tuple to be

returned by the query unlike conjunctive Boolean retrieval. Consider a car database with Boolean attributes (*AC, Power Doors, Power Brakes,* etc.). Thus if a user poses a query such as *"Select * from Cars where Make = Honda and AC = yes and Power Windows = yes"*, then a car such as <*Toyota, AC, Power Windows*> would appear in the skyline if there is no car that exactly satisfies the query conditions.

For each query q in the query log we define the *query skyline* $S(q) = \{s_1 \ldots s_L\}$, which is a collection of *skyline points*. Each skyline point s defines a subset (i.e., projection) of attributes for which any data point (tuple) remains on the skyline. For example, suppose a user poses a query q = *"Select * from Cars where Make = Honda and AC = yes and Power Windows = yes"*, and the database has three cars t_1 = <*Toyota, AC, Power Windows*>, t_2 = <*Honda, AC, Power Brakes*> and t_3 = <*Nissan, AC, Power Brakes*>. We can see from the skyline definition that the cars t_1 and t_2 will be on the skyline of q, since they are not dominated by any other cars (t_3 here) present in the database based on the attributes asked by the query q. We do not store the actual skyline data points (all attributes present in the tuple) such as t_1 and t_2 in skyline log, instead the set of attributes for which a data point is visible on the skyline. Here, t_1 = <*Toyota, AC, Power Windows*> is visible on the skyline of q because of attributes {*AC, Power Windows*} asked by q. So, the skyline points are s_1 = {*AC, Power Windows*} and s_2 = {*Honda, AC*} for which t_2 is on the skyline of q. A skylines log contains all the skylines for the query log.

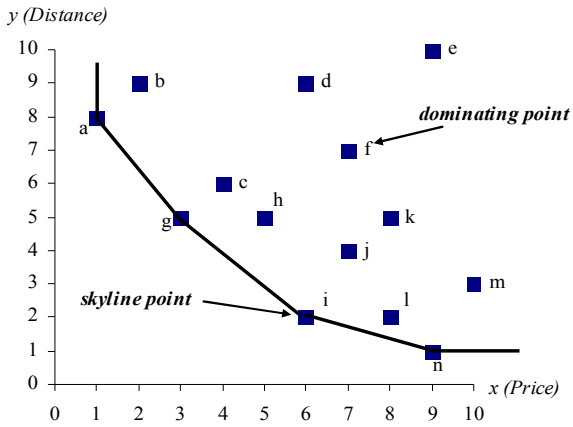


Figure 2.7 Skyline Example

2.4.3.1. Problem Definition (*SB*)

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

Table 2.1 displays the skylines log for the query log *Q* and database *D* of Figure 2.1. Note that in Figure 2.1, none of the tuples in *D* satisfies q_5 (*Turbo, Auto Trans*) completely. In contrast to the conjunctive query semantics, this does not mean that q_5 has no answer. A tuple satisfies q_5 if it has attribute *Turbo* (t_2 and t_7) or *Auto Trans* (t_3), as shown in Table 2.1.

A query can have more than one set of attributes for which data points can be visible on the skyline; e.g., for query q_5, tuples t_2 and t_7 are visible on the skyline for attribute *Turbo*, whereas tuple t_3 is visible on the skyline for the attribute *Auto Trans*. We keep separate record for each set of attribute as shown in Table 2.1.

EXAMPLE 2: *To illustrate the example consider the skylines in Table 2.1. Assume we are required to retain m = 3 attributes of the new tuple. It is not hard to see that if we retain the attributes AC, Four Door, and Power Doors (i.e., t' = {AC, Four Door, Power Doors}), the compressed tuple t' will be visible on the skylines for the maximum of three queries (q_1, q_2, and*

q_3). No other selection of three attributes of the new tuple will remain on skylines of more queries. □

Table 2.1 Skylines of Queries

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

2.4.3.2. Complexity Results for *SB*

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

2.4.3.3. Algorithms for *SB*

There are several methods proposed for efficient processing of skyline queries which are mentioned in related work (Section 4.1). Any good skyline processing technique such as [36] can be used here to find the skylines for the query log which is efficient for Boolean data. Once these skylines have been found, then our problem is to find the subset of the attributes for the new tuple so that skylines from the maximum number of queries will retrieve the new tuple. So, we can now revert back to conjunctive query semantics where a skyline *s* will retrieve the new tuple *t* if all the attributes present in the skyline is also present in *t*, i.e., $s \in t$, where *t* retains the selected subset of attributes (top-*m* attributes).

A new tuple will satisfy a skyline query if the tuple is a superset of a skyline point of the query skyline, that is, the new tuple contains all the attributes of a skyline point. Consider Table 2.1. If the new tuple has the attributes *AC*, *Four Door*, and *Power Doors* (i.e., *t'* = {*AC, Four Door, Power Doors*}), the compressed tuple *t'* will be visible on the skylines s_1, s_2, and s_3. We

need to make sure that we do not just maximize the number of skyline points that *t* dominates, but maximize the number of queries for which *t* will be visible on their skylines.

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

2.4.4. Conjunctive Boolean - Query Log - Negation (CB-QL-Negation)

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

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

2.4.4.1 Problem Definition (*CB-QL-Negation*)

Given a query log Q where a query can have negation with Conjunctive Boolean Retrieval semantics, a new tuple t, and an integer m, compute a compressed tuple t' by retaining m attributes such that the number of queries that retrieve t' is maximized.

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

Query Log *Q*

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

New tuple *t* to be inserted

Figure 2.8 Illustrating EXAMPLE 3

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

EXAMPLE 3: *To illustrate this problem variant, consider the example in Figure 2.8, which shows the query log Q with negations and a new tuple t that needs to be inserted into the database. Suppose we are required to retain m = 3 attributes. It is not hard to see that if we retain the attributes Four Door, Turbo and Auto Trans (i.e., t' = [0, 1, 1, 0, 1, 0]), we can satisfy a maximum of two queries (q_2 and q_3). No other selection of three attributes of the new tuple will satisfy more queries.* □

2.4.4.2. Complexity Results for *CB-QL-Negation*

CB-QL-Negation is also NP-hard, since *CB-QL* can be reduced to *CB-QL-Negation* if the queries have no negation.

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

2.4.4.3. Algorithms for *CB-QL-Negation*

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

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

2.4.5. Maximize Query Coverage (MQC)

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

2.4.5.1. Problem Definition (*MQC*)

Given a query log Q, and a new tuple t, find compressed tuple t' that maximizes the sum of scores of t' over all queries in Q.

2.4.5.2. Complexity Results and Algorithms for *MQC*

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

2.5 Problem Variants for Categorical Data

2.5.1. Problem Definition for Categorical Data

We also consider *categorical databases*, which are natural extensions of Boolean databases where each attribute a_i can take one of several values from a multi-valued categorical domain Dom_i. A query over a categorical database is a set of conditions of the form $a_i = x_i$, $x_i \in Dom_i$. In Conjunctive Boolean Retrieval semantics, a tuple is returned if it satisfies all the conditions in the query. As was discussed for Boolean data, alternate top-*k* retrieval

semantics can also be defined for categorical data via appropriate scoring functions (e.g., the PIR ranking function [11]). We can define problem variants for categorical data corresponding to the ones for Boolean data discussed earlier.

2.5.2. Algorithms for Categorical Data

The case of categorical data can be reduced to Boolean data in a straightforward manner. Essentially, each categorical column a_i can be replaced by $|Dom_i|$ Boolean columns, and consequently a categorical database/query log with M attributes is replaced by a Boolean database/query log with $\prod_{1 \le i \le M} Dom_i$ Boolean attributes.

2.6 Problem Variants for Numeric Data

2.6.1. Problem Definition for Numeric Data

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

2.6.2. Algorithms for Numeric Data

The Boolean variants have corresponding versions for numeric databases. For example, users browsing a database for used digital cameras may specify queries with ranges on price, age of product, desired resolution, etc, and the returned results may be ranked by price.

Such problems involving numeric ranges in queries and global scoring functions can be reduced to Boolean problem instances as follows. We first execute each query in the query log,

and reduce Q to Q' by eliminating queries for which the new tuple has no chance of entering into the top-k results, exactly as we did in Section 2.4.2. Then, for each numeric attribute a_i in Q', we replace it by a Boolean attribute b_i as follows: if the i^{th} range condition of query q contains the i^{th} value of tuple t, then assign 1 to b_i for query q, else assign 0 to b_i for query q. I.e., each query has effectively been reduced to a Boolean row in a Boolean query log Q'. The tuple t can be converted to a Boolean tuple consisting of all 1's. It is not hard to see we have created *CB-QL* variant for Boolean data, whose solution will solve the corresponding problem for numeric data.

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

2.7 Problem Variants for Text Data

2.7.1. Text Data Problem Definition

A text database consists of a collection of documents, where each document is modeled as a bag of words as is common in Information Retrieval. Queries are sets of keywords, with top-k retrieval via query-specific scoring functions, such as the tf-idf-based BM25 scoring function [39]. *Tk-QR* (described earlier in Section 2.4.2) can be directly mapped to a corresponding problem for text data if we view a text database as a Boolean database with each distinct keyword considered as a Boolean attribute. This problem arises in several applications, e.g., when we wish to post a classified ad in an online newspaper and need to specify important keywords that will enable the ad to be visible to the maximum number of potential buyers. A subtle point is that, due to the non-monotonicity of many IR ranking functions, it is possible that a top-m_1 tuple compression is worse (fewer queries retrieve document t') than a top-m_2 compression, where $m_1 > m_2$. The non-monotonicity is usually due to the document length parameter that decreases the score of a document as its length increases.

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

2.7.2. Algorithms for Text Data

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

2.8 Experiments

In this section we describe the experimental setting and the results. Our main performance indicators are (a) the time cost of the proposed optimal and greedy algorithms, and (b) the approximation quality of the greedy algorithms, for the *CB-QL* and text data problem variants presented in Sections 2.2 and 2.7 respectively. Note that no experimental results are presented for the problem variants of Sections 2.4, 2.5, and 2.6 since their algorithms are usually adaptations of the ones for *CB-QL*.

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

Datasets: We used two datasets, a cars dataset for the Boolean data experiments (Section 2.8.1), and a publications titles dataset for the text data experiments (Section 2.8.2). In particular, we use an online used-cars dataset consisting of 15,191 cars for sale in the Dallas area extracted from autos.yahoo.com. There are 32 Boolean attributes, such as *AC*, *Power Locks*, etc. We used a real workload of 185 queries created by users at UT Arlington, as well as synthetic workloads. In the synthetic workload, each query specifies 1 to 5 attributes chosen randomly distributed as follows: 1 attribute – 20%, 2 attributes – 30%, 3 attributes – 30%, 4 attributes – 10%, 5 attributes – 10%. We assume that most of the users specify two or three attributes.

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

2.8.1. Boolean Data

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

The top-*m* attributes selected by our algorithms seem promising. For example, even with a small real query log of 185 queries, our optimal algorithms could select top features

specific to the car, e.g., sporty features are selected for sports cars, safety features are selected for passenger sedans, and so on.

We first compare the execution times of the optimal and greedy algorithms that solve *CB-QL*. These are (Section 2.3): *ILP-CB-QL*, *MaxFreqItemSets-CB-QL*, which produce optimal results, and *ConsumeAttr-CB-QL*, *ConsumeAttrCumul-CB-QL*, and *ConsumeQueries-CB-QL*, which are greedy approximations. The *CB-QL* suffix is skipped in the graphs for clarity.

Figure 2.9 shows how the execution times vary with *m* for the real query workload, averaged over 100 randomly selected to-be-advertised cars from the dataset. Note that different *y*-axis scales are used for the two optimal and the three greedy algorithms to better display the differences among the methods. The *MaxFreqItemSets* algorithm consistently performs better than the *ILP* algorithm. 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 Figure 2.9 for *MaxFreqItemSets* also include the preprocessing stage, which can be performed once in advance regardless of the new tuple (user car), as explained in Section 4.3. If the pre-processing time is ignored, then *MaxFreqItemSets* takes only approximately 0.015 seconds to execute for any *m* value.

Figure 2.10 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.

Figure 2.11 and Figure 2.12 repeat the same experiments for the synthetic query workload of 2000 queries. In Figure 2.11, we do not include the *ILP* algorithm, because it is very slow for more than 1000 queries (as also shown in Figure 2.13).

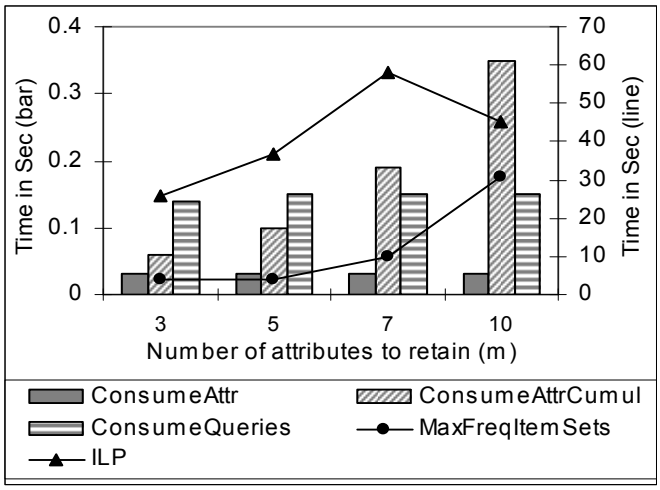


Figure 2.9 Execution times for *CB-QL* for varying *m*, for real workload of 185 queries.

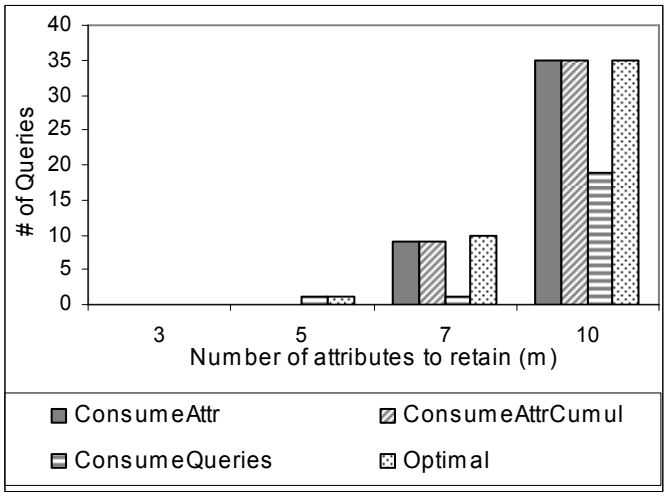


Figure 2.10 Satisfied queries for greedy and optimal algorithms for *CB-QL* for varying *m*, for real workload of 185 queries.

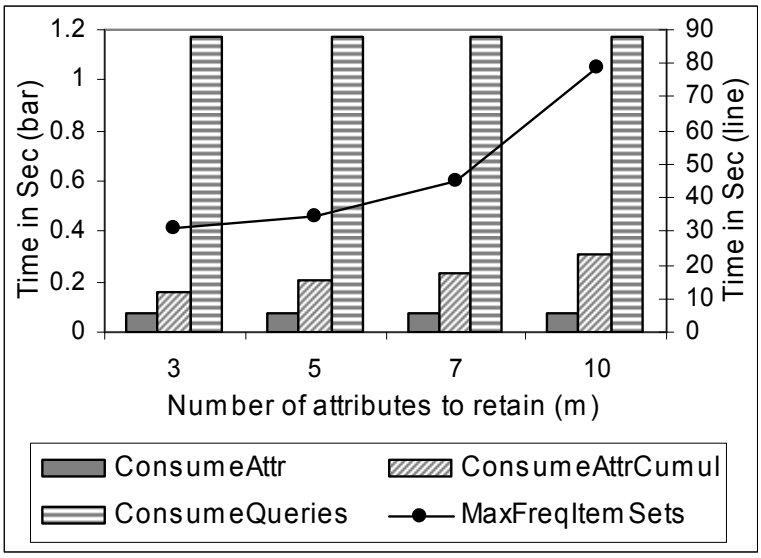


Figure 2.11 Execution times for *CB-QL* for varying *m*, for the synthetic workload of 2000 queries.

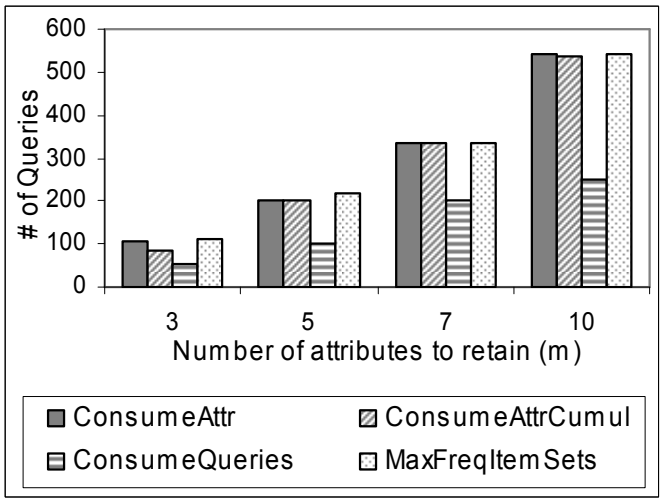


Figure 2.12 Satisfied queries for greedy and optimal algorithms for *CB-QL* for varying *m*, for synthetic workload of 2000 queries.

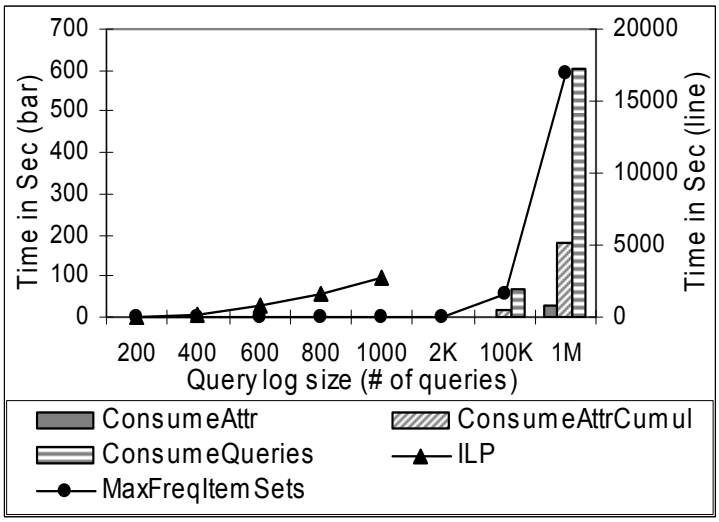


Figure 2.13 Execution times for *CB-QL* for varying synthetic workload size for m = 5.

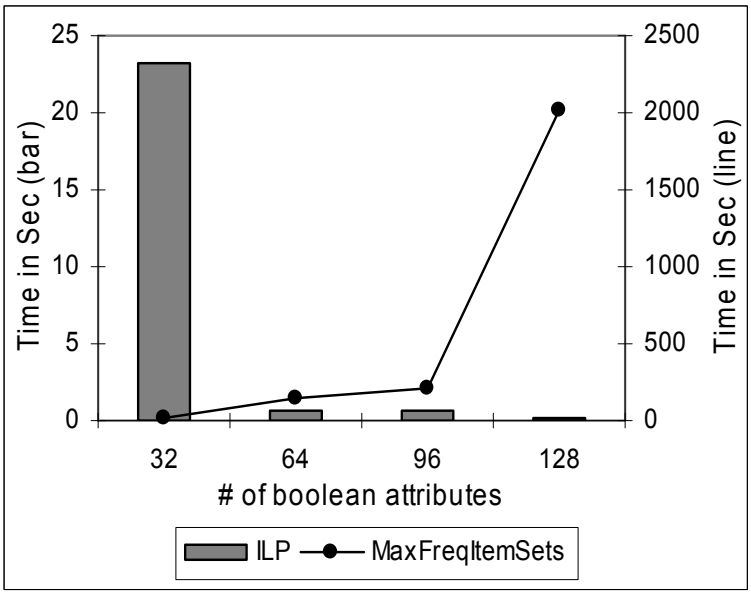


Figure 2.14 Execution times for *CB-QL* for varying number of total attributes for the synthetic workload of 200 queries for m = 5.

Next, we measure the execution times of the algorithms for varying query log size and number of attributes. Figure 2.13 shows how the average execution time varies with the query log size, where the synthetic workloads were created as described earlier in this section. We observe that *ILP* does not scale for large query logs; this is why there are no measurements for

ILP for more than 1000 queries. *ConsumeQueries* performs consistently worse than other greedy algorithms since we make a pass on the whole workload at each iteration to find the next query to add. We conclude *ConsumeQueries* is generally a bad choice.

Figure 2.14 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 Figure 2.9. 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 Figure 2.13. To summarize, *ILP* is better for small query logs and many total attributes (i.e. short and wide query log), whereas *MaxFreqItemSets* is better for larger query logs with fewer total attributes (i.e. long and narrow query log). However for query logs those are long and wide, the problem becomes truly intractable, and approximation methods such as our greedy algorithms perhaps the only feasible approaches.

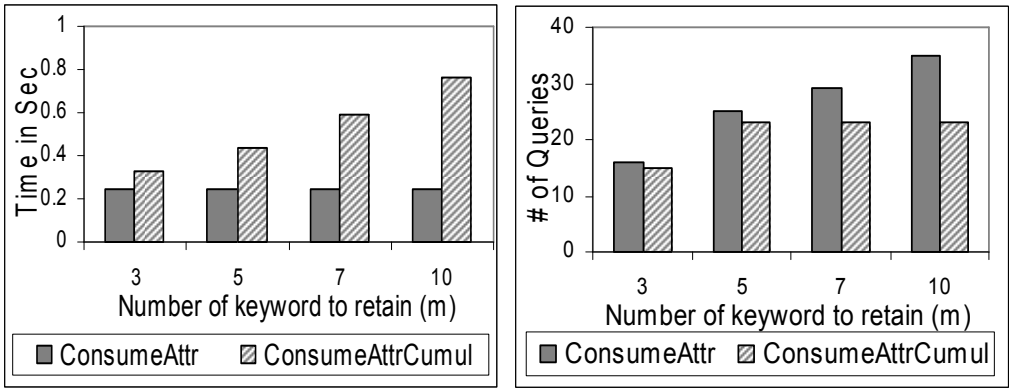
2.8.2. Text Data

We use a simplified version of the BM25 [39] ranking function, for the case of ad-hoc retrieval where any repetition of terms in the query is ignored. The weight of a term *j* is computed by the following formula:

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

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

The number of distinct keywords (equivalent to the number of attributes in the Boolean problem) for our titles dataset is 59,184. We randomly selected 50 abstracts of papers to be used as the to-be-advertised tuples. In fact, we retrieved the abstracts of 50 of the papers in our titles dataset. We consider the rest 59, 134 (59,184-50) titles as our query log That is, the tuple *t* is the abstract, and the compressed *t'* is the "best" keywords from the abstract to be used in the title. The titles of these papers were removed from the dataset. Figure 2.15 shows how the greedy algorithms *ConsumeAttr* and *ConsumeAttrCumul* perform in terms of execution time and in terms of quality, for varying *m*. We set *k* = 20 (recall that we want to maximize the number of queries in the workload for which the to-be-advertised tuple is in the top-*k* results). As mentioned in Section 2.7.2, no optimal algorithm is feasible due to the large number of total attributes (distinct keywords). Finally, we note that the *ConsumeQueries* greedy algorithm is inappropriate for the same reason, since it would just select the keywords of the shortest one or two titles, which most likely satisfy no other queries.



(a) (b)

Figure 2.15 Experiment for greedy algorithms for text data for varying top-*m*
a) Execution times, (b) Number of covered queries.

CHAPTER 3

PACKAGE DESIGN

In this section we discuss the problem "designing popular packages" which we refer to as *"package design"* problem.

3.1 Problem Framework

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

Attributes: Let $A = \{a_1 \ldots a_M\}$ be the 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 negation*s, 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 can be either 0 or 1.

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

The problem definitions are as follows:

Single Package Design (SPD) Problem: Given a query log Q with conjunctive semantics where a query can have negations, design a new tuple t (assign value [0, 1] for each attribute for the new tuple) such that the number of queries that retrieve t is maximized.

Thus, we wish to ensure that the new package (or tuple) satisfies as many customers as possible.

EXAMPLE 4. *Consider Figure 3.1 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". For this specific*

example, it is not hard to see that if we design a new package with Beach = 1, Boating = 0, Casino = 0, Fishing = 1, Historical Site = 1, Museum = 0 (i.e., new tuple t = [1, 0, 0, 1, 1, 0]), we can satisfy a maximum of 3 queries (q_2, q_4 and q_6). No other selection of attribute values for the new tuple will satisfy more queries. □

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

Figure 3.1 Query log *Q* for Running EXAMPLE 4

For the *MPD* problem, instead of designing a single package as in *SPD*, we need to design a number of packages. In fact, we need to design the minimum number of packages such that together they satisfy all the queries in the query log.

Multiple Package Design (MPD) 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 4 (Cont'd). *To illustrate this problem variant, consider Figure 3.1 again. As we can see for this example that if we design three new packages as p_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), p_2 (with Beach = 0, Boating = 1, Casino = 1, Fishing = 1, Historical Site = 1, Museum = 0 which satisfies two queries q_3 and q_5), and p_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 Figure 1. No other combination of three packages will satisfy all queries.* □

3.2 Complexity Results and Bounded Approximate Algorithms

Theorem 2: *SPD is NP-complete.*

Proof: The problem is clearly in NP since, given an assignment we just make a pass over the query log to count how many queries it satisfies. To prove that it is NP-complete, we reduce the *Maximum Independent Set* (*MIS*) problem to our *Single Package Design* (*SPD*) problem. Given a graph $G = (V, E)$, an independent set is a subset $V' \subset V$ such that no two vertices in V' are connected by an edge in E. The *MIS* problem is to find an independent set of maximum cardinality.

The reduction is as follows. Let $G = (V, E)$ be the graph in the *MIS* 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. Thus, each vertex corresponds to a query represented as a conjunction of such conditions corresponding to all edges that are incident to the vertex, and the remaining attributes are "don't care"s.

It is easy to see that finding a maximum independent set of G is equivalent to finding an assignment of {0, 1} values to each Boolean attribute in the *SPD* instance that satisfies the maximum number of queries. □

Note that for a bounded number of attributes, *SPD* can be solved in time polynomial in the size of the query log. The NP-hardness arises when the number of attributes is a variable.

Theorem 3: *MPD is NP-complete.*

Proof: To prove this problem variant is NP-complete, we reduce the *Hitting Set* (*HS*) problem [15] to a simpler variant of *MPD* problem, where there is a given set of candidate packages to pick from. We call this variant as *Multiple Package Selection* (*MPS*). If the simpler *MPS* problem is NP-Hard, it follows that the more complex *MPD* problem is also NP-Hard.

Given a ground set Z of elements, and a collection Y of subsets of Z, the goal of *HS* problem is to find the smallest subset $H \subseteq Z$ of elements that hits every set of Y.

The reduction of *HS* to *MPS* is as follows. Assume *MPS* has V candidate packages, 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 *MPS*. 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 package p_i in *MPS*. For each element e_j in s_i, set $a_j(p_i) = 1$. Set the rest of the attributes of p_i to 0. Then, a solution to *HS* is a solution to *MPS* 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.

MPD is clearly harder than *MPS* and hence *MPD* is also NP-Complete. □

3.2.1. Approximate Algorithm for SPD with Provable Bounds

Although we used the *MIS* problem to prove NP-completeness of *SPD*, *MIS* does not have a bounded-factor approximation algorithm, and hence is not useful for developing bounded-factor approximate algorithms for *SPD*. We also considered the *MINSAT* problem [30] which is also very similar to *SPD*. *MINSAT* has an approximate algorithm, with a bounded approximation factor. However, interestingly, as we explain in Section 5.1, it does not translate into a bounded-factor approximate algorithm for *SPD*.

In order to develop an approximate algorithm for *SPD* with bounded approximation factor, we consider another problem - the *Maximum Constraint Satisfaction Problem* (*Max k-CSP*), which is a well studied constraint satisfaction problem in computer theory [10, 24]. Given a set of Boolean variables and constraints, where each constraint is a Boolean formula containing at most k variables, the goal of *Max k-CSP* is to find an assignment to the variables to maximize the number of satisfied constraints. A special variant of *Max k-CSP* is the *Max k-*

CONJSAT problem, where each constraint is a conjunction of at most *k* literals. If we restrict our queries in *SPD* to contain at most *k* attributes, then *SPD* is directly mapped to *Max k-CONJSAT*. This is a reasonable restriction in many practical applications of *SPD*, as most users do not specify too many preferences in their queries.

Gustav [24] showed that the *Max k-CSP* problem has the same approximation ratio as the *Max k-CONJSAT* problem. Charikar et el in [10] proved that the approximation factor for *Max k-CSP* is > $ck/2^k$ (where *c* is absolute constant >0.44). I.e., the number of constraints satisfied is at least $ck/2^k$ times the number of constraints satisfied by the optimal assignment. So our *SPD* problem also has the same approximation factor of $ck/2^k$ as in *Max k-CSP*. To solve *Max k-CSP*, the algorithm in [10] first reduces *Max k-CSP* to the *Max k-AllEqual* problem, where given a set of clauses/constraints where each constraint has *k* literals with same value (i.e., either all true or all false), the goal of *Max k-AllEqual* problem is to find an assignment so as to maximize the satisfied clauses/constraints. Then it shows that if there is an *L* approximation factor guarantee for *Max k-AllEqual* problem, then there is an *L*/2 approximation factor guarantee for *Max k-CSP* problem. In the worst case, the approximation factor of *SPD* is $cM/2^M$, where *M* is the total number of attributes.

3.2.2. Approximate Algorithm for MPD with Provable Bounds

Recall in the NP-completeness proof of *MPD* that we first proved NP-completeness of a variant called *MPS*, where a set of candidates packages are given and we have to select a minimum subset from this set. This specific variant can be solved using the well-known greedy approximate algorithm for the *SETCOVER* problem [13], 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 approximate algorithm [13] provides an approximation bound of $H(max\ \{|B| : B \in F\})$ where $H(d)$ represents the d^{th}

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 *MPS* to the *SETCOVER* problem as follows: let us assume X is the query log (Q) and F is the set of given packages (P), where each package represents the set of queries that are satisfied by it. Now the goal is to find the minimum set of packages from P such that they cover all the queries in Q. Thus *MPS* 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 *MPS* is *log(S)* where S is the number of queries in the query log Q.

Extending this approach to the *MPD* 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, we can avoid this enumeration by combining the approximate algorithm for *SPD* with the greedy approximate algorithm for *SETCOVER*. This combined algorithm proceeds as follows: (i) at each iteration the algorithm makes a call to the approximate algorithm for *SPD* over the not-yet-satisfied queries, which returns a package 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 package, and (ii) the process repeats until all queries are satisfied.

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

Theorem 4: The number of packages returned by the approximation algorithm described above is at most $(2^k/ck)\log(S)$ times the number of packages returned by an optimal algorithm for *MPD*.

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

Let us imagine that each package 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 p'_1 satisfies queries q_1 and q_2, then $wt(q_1) = wt(q_2) = 1/2$; if package p'_2 satisfies query q_2, 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 t.

Consider any package p_i of P_{min}. Let $p'_{i,1}, p'_{i,2}, \ldots p'_{i,r}$ be the order in which the approximation algorithm returns packages that have non-empty intersection with p_i, i.e., that there exists at least one query in the query log that is satisfied by both $p'_{i,j}$ and p_i.

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

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

Consider $p'_{i,j+1}$. We argue that $|p'_{i,j+1}| \geq (ck/2^k)$ uij. Because if this were not so, then the greedy approximate algorithm for *SPD* would not have returned $p'_{i,j+1}$, 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, p_i itself would have been a better package to return next instead of $p'_{i,j+1}$.

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

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

$$\leq \sum_{1 \leq j \leq S} \frac{1}{\left(\frac{ck}{2^k}\right) j}$$

$$\leq \left(\frac{ck}{2^k}\right) \log(S)$$

Thus, the aggregate weight of all queries is

$$t = \quad |P'| \quad \leq \quad |P_{\min}| \left(\frac{ck}{2^k}\right) \log(S)$$

□

In summary, the above discussions are interesting because they demonstrate the existence of bounded approximation factor algorithms for the *SPD* and *MPD* problems. However, in practice these approximation factors are too suboptimal to be useful, and more practical algorithms (both optimal as well as approximate) are needed that work well for moderate as well as large problem instances. Such algorithms are discussed next in this paper.

3.3 A Feasible Optimal Algorithm for *SPD*

In this section we optimal algorithms for the *SPD* problem. It is easy to see that a naïve brute-force optimal approach to design a new tuple *t* can be designed as:

a) For each of the possible 2^M combination of attribute values, find the number of queries in *Q* that will be satisfied by this assignment.

b) Return the assignment with the highest count.

Clearly, while the naïve algorithm is polynomial in the size *S* of *Q*, it is unfortunately exponential in *M*. Thus it is not feasible when the number of attributes is large since the algorithm has to generate an exponential number of possible combinations of attribute values.

We propose a novel optimal algorithm based on adaptations of the Signature Tree data structure [12] which is much more efficient than the Naïve algorithm, as well as generic branch-and-bound and backtracking methods, due to smart pruning techniques we employ. Our algorithm works well for moderate problem instances. 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.

3.3.1. Review of Signature Trees

We illustrate Signature Trees by means of an example. Consider Figure 3.2(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 3.2(b) shows the corresponding signature tree. In general, given a Boolean database $D = \{t_1 \ldots t_N\}$, a signature tree is a binary tree *T* such that

a) *T* has *N* leaves labeled with the signatures of the tuples $t_1 \ldots t_N$.

b) 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 3.2(b) attribute a_3 is associated with two different internal nodes. However the same attribute cannot be repeated along any root to leaf path.

c) 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 3.2(b), t_3, t_4 and t_5 all have $a_2 = 0$.

d) 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 uniquely identified by the assignments $a_1 = 0$, $a_2 = 0$ and $a_3 = 1$.

Given a signature tree, searching for tuples is straightforward. For example, let us search for tuple t_3 in the signature tree in Figure 3.2(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 [12]. 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 3.2(b) shows the signature tree created by this process for the database in Figure 3.2(a).

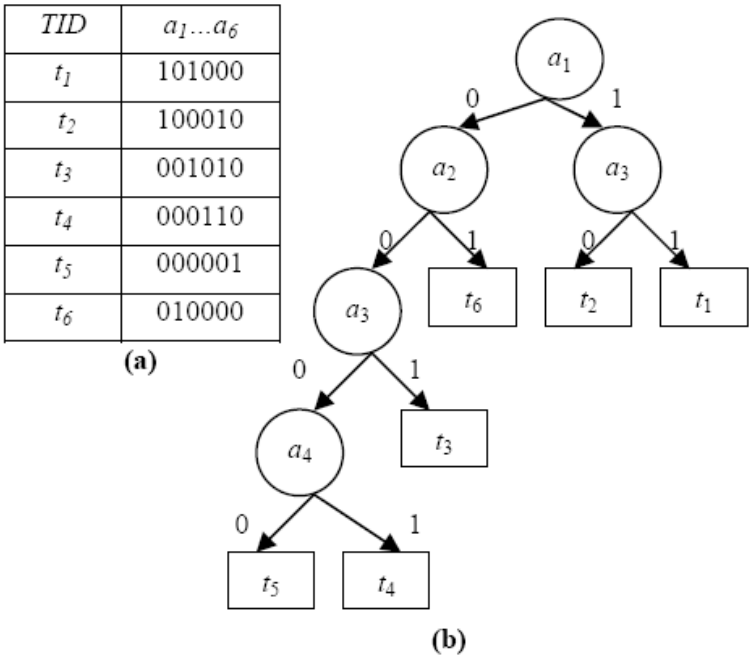


Figure 3.2 A Boolean Database and a Signature Tree – (a) Boolean database, (b) signature tree

3.3.2. Limitations of Signature Trees

Traditional signature trees as described above are not directly applicable for the *package design* 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. However, queries also have "don't care" values in addition to Boolean values. In fact, for any package with many attributes/features, users usually show positive/negative interest only for a few attributes, and for the remaining attributes they specify "don't care". Signature trees need to be significantly modified to take care of such "don't care" values.

Secondly, traditional signature trees are created without any pruning. So, for large databases with a large number of attributes, the tree can be very large. In our adaptation, we

avoid constructing the entire tree, but store only one path at any time. Moreover, as shall be described later, another difference is that we do not perform traditional search operations. We also perform several types of pruning to improve the efficiency.

3.3.3. Optimal Algorithm

Let us consider the signatures of the query log *Q* in our running example (see Figure 3.3).

Overview of algorithm: We do a depth first creation of the signature tree, where at each downward step we pick an attribute that will be used as the splitting node, as discussed below. Further, for every traversed node *v*, we check if this node has a chance to be the best package. That is, we compare the number of satisfied queries in *v* to the *current maximum number C of satisfied queries*. If *v* fails this test, we prune the whole subtree rooted at *v*, since the children of *v* satisfy at most as many queries as *v*.

Query ID	*Signature($a_1a_2…a_6$)*
q_1	101???
q_2	1?0?1?
q_3	0?1?1?
q_4	?0011?
q_5	?1?0?1
q_6	100??0

Figure 3.3 Signatures for Query log *Q*

Note that unlike traditional signature trees, we do not recursively compute both the left and right sub-trees. Instead, we only consider the child that contains the maximum number of queries, since our objective is to create an assignment of values to all attributes that maximizes the number of satisfied queries. That means, after splitting the dataset based on the splitting attribute, if the left child contains more number of queries than that of the right child then we only consider the left child, and vice versa. This is a pruning beyond generic branch-and-bound

technique. Like branch-and-bound, our algorithm is different than classic backtracking technique in terms of traversing the tree and also used for optimization purpose.

When we find the node *v* with the maximum *C*, we create the corresponding "best" package by traversing the path from *v* to the root and assigning attribute values accordingly.

Selection of best attribute to use as splitting node: Identifying a good splitting attribute (node) is an important step as this can make the tree either balanced or skewed. Moreover, we want to find the attribute with the minimum number of "don't care" values to minimize the duplications for "don't care". Thus, we select an attribute as splitting attribute which contains both 0 and 1 values and with minimum number of "don't care" (?) values. Any tie is broken arbitrarily.

Figure 3.4 shows pseudo-code of optimal algorithm.

Algorithm: *BuildSigTree(Signature Tree T)*
Let *B* be the current max # queries in a leaf node of the signature tree. Initially *B* = 0.
Stack *S*
S.push(*T*) //*T* is root node of *T*
While *S* not empty do
u=*S*.top()
if both children of *u* have been processed before or there is no splitting attribute
update *B* and current assignment *C //if* |*u*|>*B then B*=|*u*|
S.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 $u_{A=1}$)
S.push ($u_{A=0}$) //dfs to create tree for left child.
Else if $u_{A=1}$ not processed before
S.push ($u_{A=1}$) //dfs to create tree for right child
Return current assignment *C*

Figure 3.4 Pseudocode of Optimal Algorithm.

EXAMPLE 5: *If we consider Figure 3.1 in the setting of our vacation packages running example (where attribute a_1 is Beach, a_2 is Boating and so on), we create the signature tree as follows: We see that attribute a_3 (Casino) has the minimum number of "don't care" (?) values (which is 1) and it also contains both 0 and 1 values. So, as shown in Figure 3.5, we select a_3 (Casino) as the first splitting attribute. This splits Q into two groups: left child with 4 queries (q_2, q_4, q_5, q_6) and right child with 3 queries (q_1, q_3, q_5). As discussed before, at this step we only create the left child, which has maximum (4) queries. Now we follow the same process for these four queries and split it again until we create the first leaf node.*

After we get the first leaf node which contains queries q_2, q_4, and q_6, for which the "current maximum" = 3, we start moving upward from the leaf node in a depth-first manner. We check if the other child of the parent node contains more queries than the "current maximum" or not. The right (other) child of node a_2 (parent of leaf node) contains 2 queries (q_2 and q_5; with a_3 = 0 and a_2 = 1)) which is less than 3 ("current maximum"). So we do not create the right path from node a_2. Next we move to node a_3 which is the parent of node a_2. The right (other) child of node a_3 contains 3 queries (q_1, q_3, q_5; with a_3 = 1) which is also not more than 3 ("current maximum"). So we do not create right path from node a_3 as this will never lead to a leaf node with more than 3 queries. Finally, the leaf node with the "current maximum" is the winner leaf node.

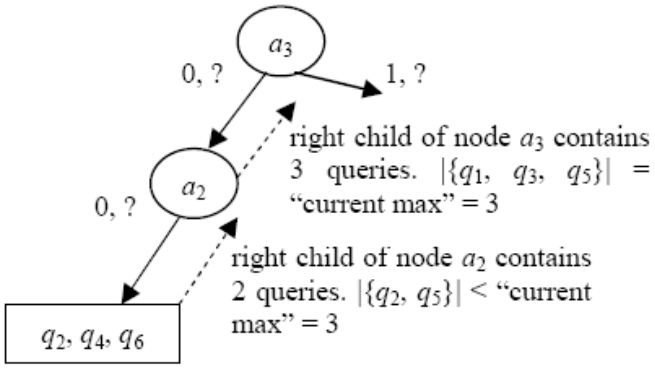


Figure 3.5 Signature Tree Creation and Updating.

After we get the winner leaf node (with queries q_2, q_4, q_6), we design the new tuple based on the queries in the leaf node. For example, for query q_2 (1?0?1?), we assign values 1 for attribute a_1 (Beach) and a_5 (Historical Site), and value 0 for attribute a_3 (Casino), and so on. After this process, we have the new tuple as 100110 (Beach = 1, Boating = 0, Casino = 0, Fishing = 1, Historical Site = 1, Museum = 0) which will be retrieved by maximum 3 queries (q_2, q_4, and q_6). □

3.3.4. Cost Analysis of the Optimal Algorithm

In the worst case, the *SigTreeSPD* algorithm is linear on the size of the query log and exponential on the number of attributes. That is, its cost is $O(S \cdot 2^M)$, where S is the number of queries and M is the number of attributes. It is linear on the size of query log because for each node of the tree we make a pass on the query log to count the satisfied queries. It is exponential on the number of attributes since in the worst case, the tree will have 2^M nodes. However, due to our pruning methods, the average case is much better than this worst case bound.

3.4 Scalable Approximate Algorithms for *SPD* and *MPD*

So far we have presented bounded approximate algorithms for *SPD* and *MPD* (Section 3.2) and an optimal algorithm for *SPD* (Section 3.3). The approximate algorithms of Section 3 are useful for theoretical purposes, but they are not very feasible in practice. This is because they are main memory algorithms based on semidefinite programming (*SDP*) relaxation, which do not scale for large instances. Moreover, the bounded approximation factors are too suboptimal to be useful in practice. In this section, we propose scalable approximate algorithms for *SPD* and *MPD*, which are shown to perform well in terms of scalability and approximation error in Section 3.5.

3.4.1. Approximate Algorithm for SPD

First we complement both the query log and query semantics as follows: *(a)* Complement the value of each attribute for each query in the query log, i.e., if an attribute has value 0 then convert it to 1 and vice versa. *(b)* Complement the conjunctive semantics to disjunctive semantics. Let ~*Q* denote the converted query log *Q*. At this point it becomes similar to *MINSAT* problem [30], which is also NP-complete. Given a set *U* of Boolean variables and a collection of disjunctive clauses over *U*, the goal of *MINSAT* problem is to find a truth assignment that minimizes the number of satisfied clauses. *SPD*, which has conjunctive clauses (queries) and is the complement of the *MINSAT* problem. Solving *MINSAT* on ~*Q*, we get an assignment that satisfies the minimum number of queries in ~*Q*; which corresponds to satisfying the maximum number of queries in the original query log *Q*.

Our heuristic operates as follows. Given any ordering of the variables, we greedily sequentially select an assignment for each variable to satisfy the smallest number of additional clauses (clauses in ~*Q* in *SPD*). Figure 3.6 displays the pseudocode of the algorithm. Note that the *For* loop in Figure 6 is inspired by a *MINSAT* heuristic proposed in [30].

Approx Algorithm: *HeuristicSPD*
Let *Q* be the query log, *A* ($a_1 \ldots a_M$) 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

Figure 3.6 Pseudocode of Approximate Algorithm for *SPD*

The above heuristic has an approximation ratio equal to the maximum number of attributes (literals) in any query (clause). Note that this ratio does not hold for *SPD* since in *SPD*

the solution is complemented, that is the number of satisfied queries is *S* minus the number of satisfied queries in *MINSAT*. Nevertheless, our experimental results in Section 6 show that the algorithm has a very small approximation error in practice.

3.4.2. Approximate Algorithm for MPD

As mentioned in Section 3.2, the solution for *MPD* involves a combination of solving the *SETCOVER* problem with the solution of multiple instances of *SPD*. We consider two variants for *SPD*: one using the optimal *SPD* algorithm of Section 3.3.3, and one using the heuristic of Section 3.4.1. In particular, the algorithms are as follows:

a) Apply algorithm for *SPD* and add solution assignment (package) 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 3.7 displays the pseudocode of two approximate algorithms for *MPD*. Note that the way that the idea of greedily removing satisfied queries is inspired by a SETCOVER heuristic [13].

As discussed in Section 3, the *SETCOVER* algorithm has *log(n)* approximation bound, where *n* is the size of the universe of elements. Hence, the *SigTreeMPD* algorithm has approximation bound *log(S)*, where *S* is the number of queries in *Q*, since the *SPD* component of the algorithm is optimal. There is no approximation bound for *HeuristicMPD*, but it is shown to perform well in Section 3.5.

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

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

Figure 3.7 Pseudocode of Algorithms for MPD

3.5 Experiments

Our main performance indicators are (a) the time cost of optimal and approximate algorithms, and (b) the approximation quality of approximate 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 used datasets of products and product queries. 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): We collected 237 queries for cell phones from users of UT-Arlington 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 we generated larger datasets 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, and 30).

Table 3.1 summarizes the query logs or datasets.

Table 3.1 Summery 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

3.5.1. Experimental Results for SPD

In addition to *SigTreeSPD* and *HeuristicSPD,* we consider the *NaïveSPD* algorithm, which creates the whole signature tree. That is, *NaïveSPD* does not employ the pruning ideas of *SigTreeSPD*.

Figures 3.8 and 3.9 show the performance and quality of the algorithms for the real query log (REAL). Note that *SigTreeSPD* has almost optimal quality. *NaïveSPD* is much slower than the other two algorithms and hence we do not plot in the same graph. Instead we report on the representative times: for the real query log (REAL) with 30 attributes in Figure 8, *NaïveSPD* takes more than 64000 seconds which too slow and infeasible compared to other two algorithms.

Figures 3.10 and 3.11 show the performance of the algorithms for varying query log size and number of attributes respectively, for REAL+ dataset. We randomly select a subset (of size 10, 15, ..., 30) of the attributes of the dataset. As we can see from the graphs, the approximate algorithm is much more efficient than optimal algorithm. As shown before for real query log (RAL), *NaïveSPD* is much slower than the other two algorithms for REAL+ as well. For 10 and 15 attributes in Figure 3.11, *NaïveSPD* takes 12 and 280 seconds respectively. For

larger number of attributes and for varying query log sizes it becomes infeasible. Hence we do not report on *NaïveSPD* in other graphs.

Figures 3.12 and 3.13 show the quality (number of queries satisfied in the query log) of the approximate algorithm for varying query log size and number of attributes respectively for REAL+ dataset. As we can see from the graphs, the approximate algorithm performs well. As we see in Figure 3.13, the number of satisfied queries decreases as the total number of attributes increases. The number decreases because as more attributes are added, the queries become more selective and harder to be satisfied. The approximate algorithm has quality close to the optimal algorithm.

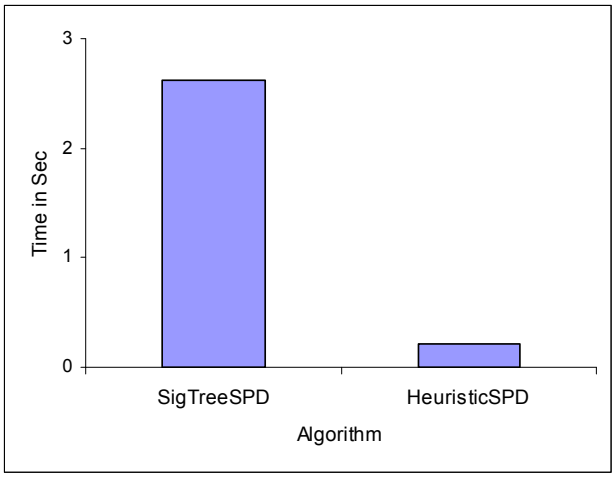


Figure 3.8 Time cost for REAL dataset

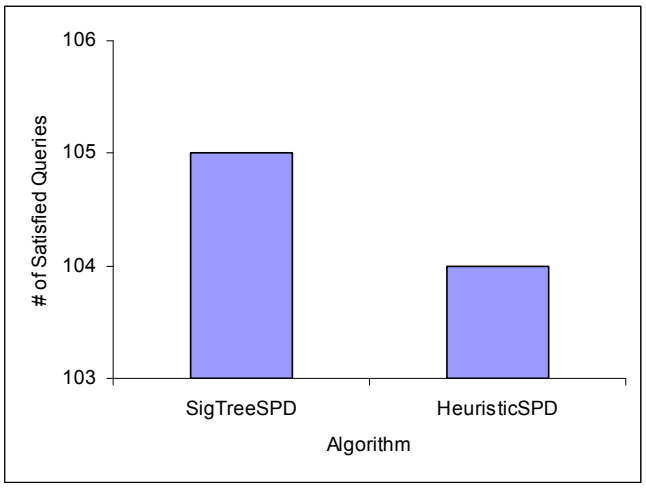


Figure 3.9 Quality for REAL dataset

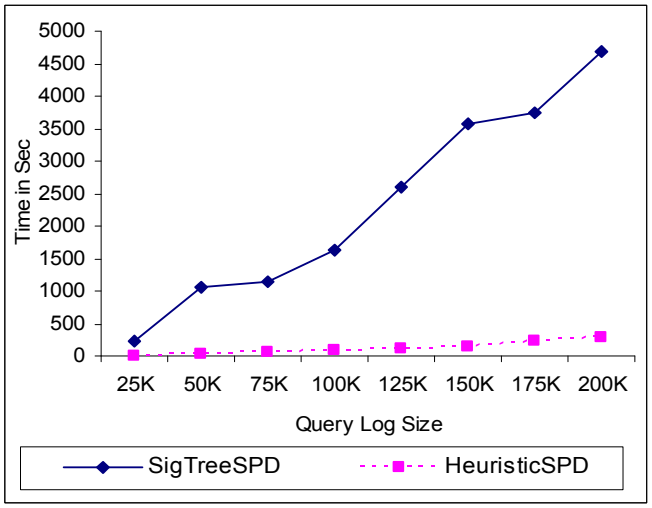


Figure 3.10 Time cost for varying query log size for REAL+_30

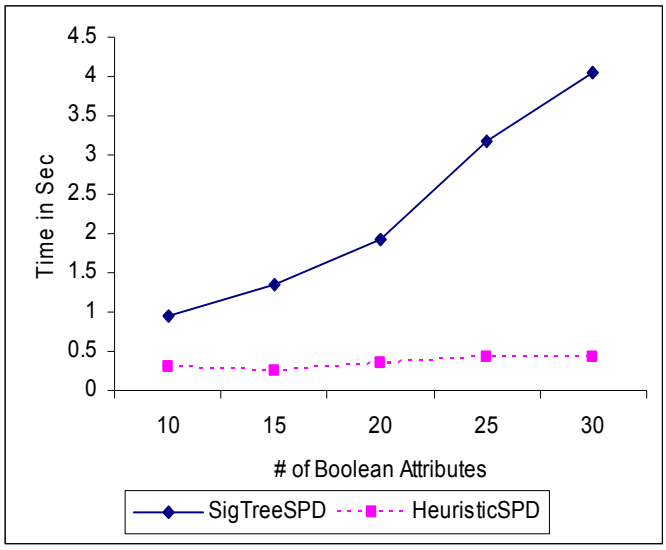


Figure 3.11 Time cost for varying # of attributes for REAL+_1000

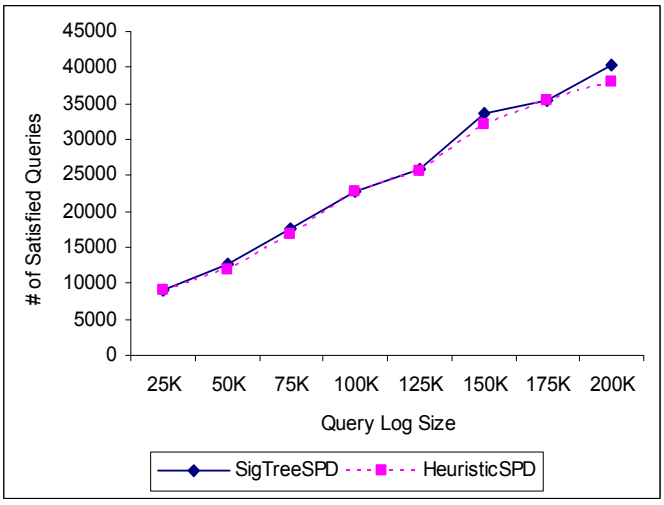


Figure 3.12 Quality for varying query log size for REAL+_30

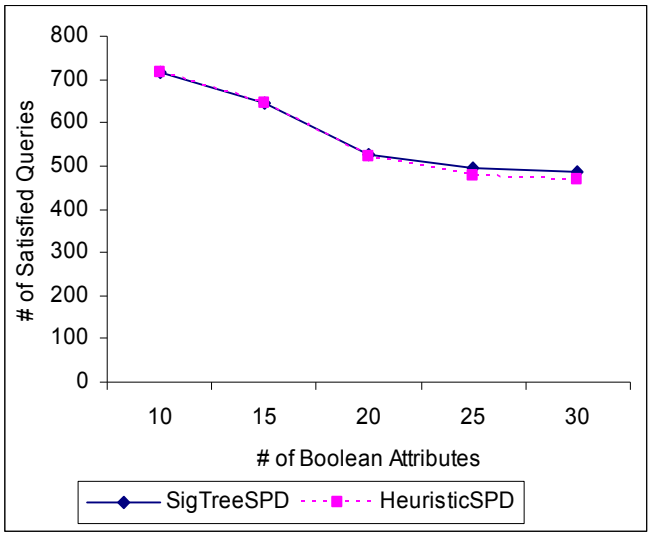


Figure 3.13 Quality for varying # of attributes for REAL+_1000

3.5.2. Experimental Results for MPD

We do not provide any experiment results for an optimal algorithm for *MPD*. This is because the optimal algorithm for *MPD* is infeasible as it is double exponential. First, the number of candidate packages is exponential on the number of attributes, and then the combination of packages is exponential on the number of packages. So we only provide the experiment results for approximate algorithms for *MPD*.

Figures 3.14 and 3.15 respectively show the performance and quality of the algorithms for the REAL dataset.

Figures 3.16 and 3.17 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 approximate algorithm is much more efficient than the optimal algorithm. The missing data in Figure 3.16 is due to the very slow speed of the optimal algorithm for large datasets. The running time of the optimal algorithm increases exponentially as the number of total attributes increases, which follows the asymptotic analysis of the algorithm in Section 3.3.4.

Figures 3.18 and 3.19 show the quality (number of products need to design to satisfy all queries in the query log) of the approximate algorithm for varying query log size and number of attributes respectively for REAL+. For the same reason as in Figure 3.16, Figure 3.18 also has missing data for optimal algorithm.

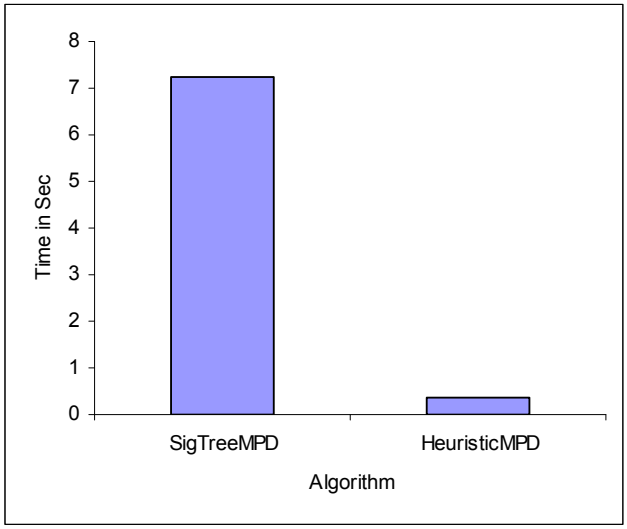


Figure 3.14 Time cost for REAL dataset

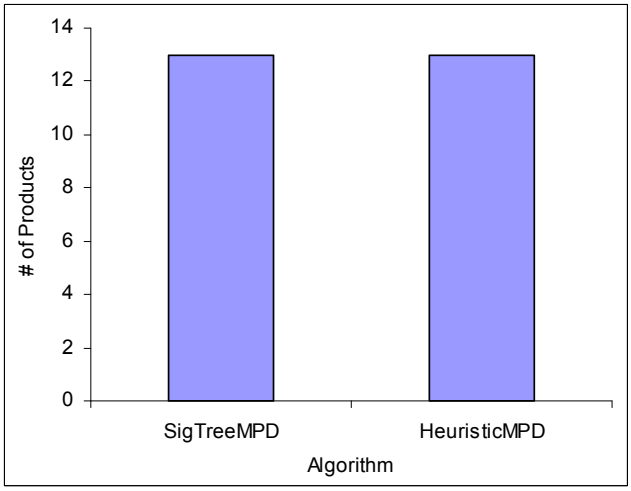


Figure 3.15 Quality for REAL dataset

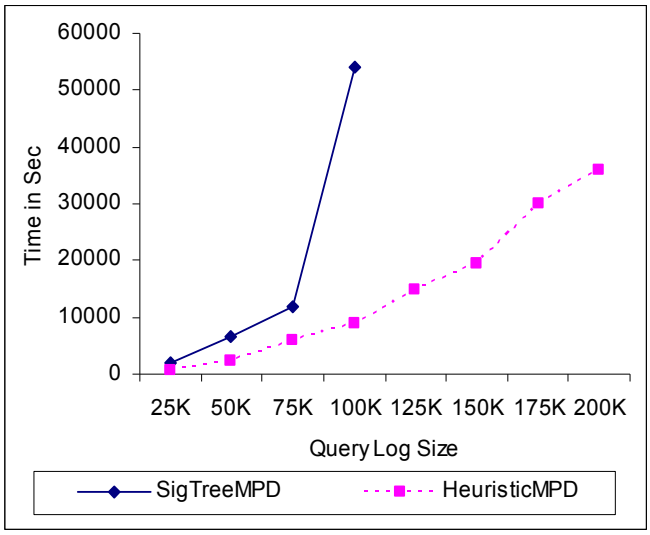


Figure 3.16 Time cost for varying query log size for REAL+_30

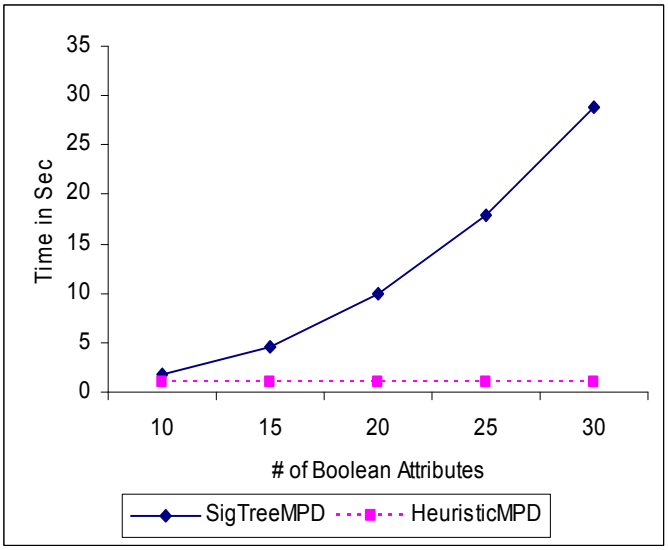


Figure 3.17 Time cost for varying # of attributes for REAL+_1000

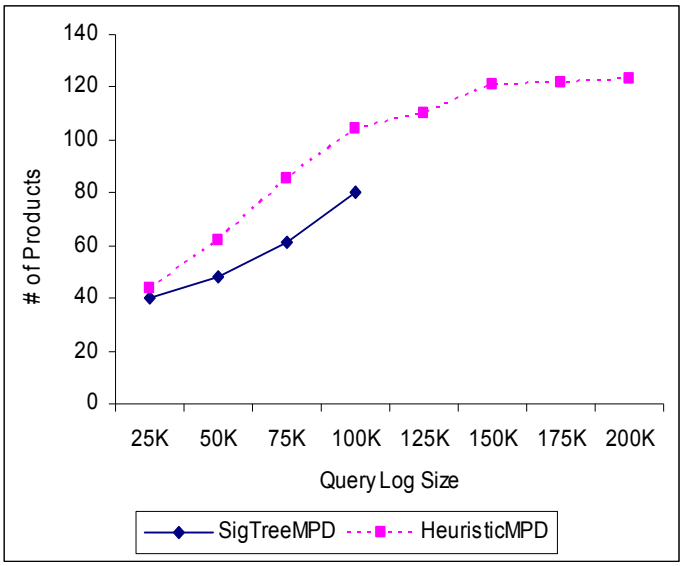


Figure 3.18 Quality for varying query log size for REAL+_30

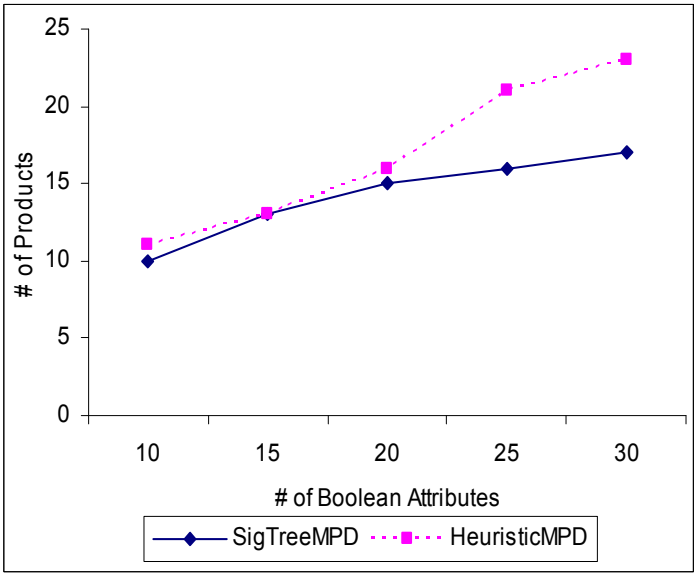


Figure 3.19 Quality for varying # of attributes for REAL+_1000

3.6 Other Interesting Variants

In this section we briefly discuss other interesting problem variants.

3.6.1. Profitability Constraint

In practice package design (especially product design) is often constrained by profitability concerns. We model profitability with a function *F()* of the attributes that returns true if an assignment is profitable and 0 otherwise. This modeling is a generalization of the modeling of [33], where the profitability function defines a hyperplane that divides the space to two subspaces. Hence, the problem becomes to find the assignment that maximizes the number of satisfied queries while achieving profitability. The profitability constraint becomes one more pruning condition in *SigTreeSPD* and *HeuristicSPD* algorithms.

3.6.2. Dependencies among Attributes

Another problem variant arises when there are dependencies among the attributes. E.g., if a cell phone has the *3G* feature, it must also have *internet* feature. We tackle this by removing the unsatisfiable queries from the query log and using the dependencies to optimize the heuristic algorithms.

3.6.3. Other Query Semantics

Instead of strict conjunctive Boolean query semantics, a user may be satisfied with partial matches, that is, tuples that only satisfy a percentage of the query conditions. The problem then becomes to design a new tuple in such a way that it partially satisfies (at least *x*% of the query conditions) the maximum number of queries in the query log. Efficiently handling this semantics is part of our future work.

3.6.4. Candidate Packages Given

Both *SPD* and *MPD* assume that all possible packages can be designed. Instead, we may be given a set of candidate packages. This variant of *SPD* is polynomial since we can naively consider each candidate package and pick the best. The *MPD* variant remains NP-hard.

3.6.5. Top-r MPD

Instead of trying to satisfy all queries of the query log, we may be interested to design a set of *r* packages to maximize the number of satisfied queries. An interesting observation of this variant is that the approximation bound of the *SETCOVER* algorithm changes from log(*S*) to 1-1/e [25], and hence the overall approximation bound in Section 3 changes from $(2^k/ck)\log(S)$ to $(2^k/ck)(1-1/e)$.

CHAPTER 4

RELATED WORK

4.1 Related Work for Attributes Selection

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 [40] ranking functions, like BM25 [39], as well as link-structure-based techniques like PageRank [6] if such links are present (e.g., the Web). In the database world, automatic ranking techniques for the results of structured queries have been proposed [1, 11, 44]. Also there has been recent work [14] on ordering the displayed attributes of query results.

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

Although the problem of choosing attributes is seemingly related to the area of feature selection [21], our work differs from the work on feature selection because our goal is very specific – to enable a tuple to be highly visible to the database users – and not to reduce the cost of building a mining model such as classification or clustering.

Kleinberg et al. [28] present a set of microeconomic problems suitable for data mining techniques; however no specific solutions are presented. Their problem closer to our work is identifying the best parameters for a marketing strategy in order to maximize the attracted customers, given that the competitor independently also prepares a similar strategy. Our problem is different since we know the competition. Another area where boosting an item's rank

has received attention is Web search, where the most popular techniques involve manipulating the link-structure of the Web to achieve higher visibility [17].

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 [42]). Integer programming is well-known to be NP-hard [15]; however carefully designed branch and bound algorithms can efficiently solve problems of moderate size. In our experiments, we use an of-the-shelf ILP solver available from http://lpsolve.sourceforge.net/5.5/download.htm.

Computing frequent itemsets is a popular area of research in data mining and some of the best known algorithms include Apriori [2] and FP-Tree [22]. Several papers have also investigated the problem of computing maximal frequent itemsets [6, 8, 18, 20, 26]. Almost all the popular approaches are designed for sparse datasets and do not work well for our unique problem of dense datasets. Apriori [2] employs a bottom-up, breadth first search that enumerates every single frequent itemset. In many applications (especially in dense data) with long frequent patterns enumerating all possible subsets of an M length pattern (M can easily be 50 or 60 or longer) is computationally unfeasible. Also, we are not interested in mining all frequent itemsets, but only maximal frequent itemsets in our algorithm. A known approach for mining maximal frequent itemsets is the complete random walk [20], which is a bottom-up approach. But in a dense dataset the maximal frequent itemsets usually lie on the top region of the lattice, and if a bottom-up approach is used to find maximal frequent itemsets, it will have to traverse a long portion of the lattice (i.e., numerous levels) and will be inefficient. To see this, consider a table with 50 attributes, and assume we need to determine a compressed tuple t' with 10 attributes. Now, we need to know the itemset of $\sim Q$ (complemented query log which is a dense dataset) of size 40 with maximum frequency. Due to the dense nature of $\sim Q$, the bottom-up approach will not be able to compute frequent itemsets beyond a size of 5-10. Likewise, other approaches for mining maximal frequent itemsets such as the Genetic Algorithm (GA)

based approach [26] is also mainly intended for sparse dataset and does not work well for dense dataset. In contrast, our proposed method works well for dense dataset.

The recent works [33] and [34] are related to our work. The former tries to find out the dominant relationship between products and potential buyers where by analyzing such relationships, companies can position their products more effectively while remaining profitable. The latter introduces skyline query types taking into account not only min/max attributes (e.g., price, weight) but also spatial attributes and the relationships between these different attribute types. Their work aims at helping manufacturers choose the right specs for a new product, whereas our work to choose the attributes subset of an existing product for advertising purposes.

In work [35], we tackled the main variant of the problem with Boolean conjunctive query semantics where a tuple satisfies a query if all the attributes present in query are also present in the tuple (Section 2.2). We extend the idea now. We consider both the database (existing products) and query log with various query semantics (conjunctive, top-k, skyline, negations, etc.).

Several techniques have been proposed for efficient skyline query processing [7, 32, 38, 45]. There has been recent work on categorical skylines [41] and skyline computation over low cardinality domains [36] that also considers skyline for Boolean data as well. One main difference of our work with the existing works is that our goal is not to propose a method for processing or maintaining the skylines, instead we use skylines as a query semantic where a new tuple can be visible for maximum number of queries.

Another related work is mining top-*k* frequent itemsets without minimum support threshold [23] which finds top-*k* closed frequent itemsets. This is inapplicable in our case because we are interested in finding out all the maximal frequent itemsets, and not just the top-*k* frequent itemsets. Also it is not proven that the top-*k* approach works well for dense dataset. The top-*k* approach without minimum support threshold [23] finds top-*k* frequent closed patterns

of length no less than *min_l*, where *min_l* is the minimal length of each pattern. In our problem, we do not have any *min-l* restriction.

4.2 Related Work for Package Design

Optimal product design or positioning is a well studied problem in Operations Research and Marketing. Shocker and Srinivasan [43] first represented products and consumer preferences as points in a joint attribute space. After that, several approaches and algorithms [3, 4, 5, 16, 19, 30] 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. Like our work, users in this domain in fact do not get to select the attributes or features they like and don't like. Instead of involving users directly in the process of designing new package, we use previous user search queries for the same package and it is easy to collect the preferences (search queries) for large number of Internet users nowadays. We also consider large query logs to design the new package and allow users to express their interests in attribute or feature level in terms of positive, negative and "don't care".

Recent works on dominant relationship [34] and dominating neighborhood [33] 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 our work users may have opposite preferences for the same attribute, and our algorithms can be used with or without a profitability plane. Li et al. [33] also considers spatial, non-preference attributes. Our algorithms can be modified to support skyline semantics; however, more efficient algorithms may be possible for this problem variant given its restrictive nature.

In [35] we tackled the *attribute selection* 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 our work in this paper. In this paper, we consider designing an object (a new tuple), that is, assign values for all attributes instead of selecting subset of attributes.

The *MPD* problem can be viewed as the segmentation problem [29] for the *SPD* problem. However, in *MPD* the size of each segment (package) is not given.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this work we introduced the problem of selecting the best attributes of a new tuple, such that this tuple will be ranked highly, given a dataset, a query log, or both, i.e., the tuple "stands out in the crowd". We presented variants of the problem for Boolean, categorical, text and numeric data, and showed that even though the problem is NP-complete in most cases; optimal algorithms are feasible for small inputs. Furthermore, we present greedy algorithms, which are experimentally shown to produce good approximation ratios. While the problems considered in this work 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, our focus is on deciding what subset of attributes to retain of a product. We do not attempt to suggest pricing of a product to be more profitable, which is a problem tackled in marketing research, e.g., [37]. 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.

In this work we also investigated the problem of designing a package, such that, given a query log, this package will be returned by the maximum number of queries in the query log where a query can have negations. We considered both single and multiple package design. We showed that even though the problems are NP-complete, optimal algorithm is feasible for

moderate inputs. Furthermore, we present approximate algorithms, which are experimentally shown to produce good approximation ratios for large databases. A future direction is to extend the problem to other data types, such as categorical, text and numeric and different query semantics like top-*k* and skyline retrieval.

REFERENCES

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

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

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

[4] Sönke Albers, Klaus Brockhoff, "Optimal Product Attributes in Single Choice Models", Journal of the Operational Research Society (1980) 31, 647–655.

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

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

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

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

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

[10] Moses Charikar, Konstantin Makarychev, Yury Makarychev, "Near-optimal algorithms for maximum constraint satisfaction problems", ACM-SIAM symp. on Discrete algorithms, 2007.

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

[12] Yangjun Chen, Yibin Chen, “On the Signature Tree Construction and Analysis”, IEEE Trans. Knowl. Data Eng. 18(9), 1207-1224, 2006.

[13] Cormen, T.H., C.E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, The MIT Press, 2001.

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

[15] 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.

[16] Bezalel Gavish, Dan Horsky, Kizhanatham Srikanth, “An Approach to the Optimal Positioning of a New Product”, Management Science, 29, 11 (Nov 1983), 1277-1297.

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

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

[19] Thomas S. Gruca, Bruce R. Klemz, “Optimal new product positioning: A genetic algorithm approach”, European Journal of Operational Research, 146, 3 (May 2003), 621-633.

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

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

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

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

[24] Gustav Hast, “Approximating Max kCSP – Outperforming a Random Assignment with Almost a Linear Factor”, ICALP 2005.

[25] Hochbaum, D.S., *Approximation Algorithms for NP-Hard Problems*, PWS Publishing Company, 1997.

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

[27] D. S. Johnson, “Approximation algorithms for combinatorial problems”, J. Comput. System Sci., 9 (1974).

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

[29] Jon Kleinberg, Christos Papadimitriou, Prabhakar Raghavan, “Segmentation Problems”, ACM Symposium on the Theory of Computing, 1998, 473-482.

[30] Rajeev Kohli, Ramesh Krishnamurti, Prakash Mirchandani, “The Minimum Satisfiability Problem”, Siam Journal on Discrete Mathematics, 7 (2), 275–283, 1995

[31] Rajeev Kohli, Ramesh Krishnamurti, “Optimal product design using conjoint analysis: Computational complexity and algorithms”, European Journal of Operational Research, 40, 2 (May 1989), 186-195.

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

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

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

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

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

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

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

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

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

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

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

[43] A.D. Shocker, V. Shrinivasan, “A consumer-based methodology for the identification of new product ideas”, Management Science, 20, 6 (Feb 1974), 921-937.

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

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

BIOGRAPHICAL INFORMATION

Md (Muhammed) Zahiduzzaman Miah received BS degree in Civil Engineering from Khulna University of Engineering and Technology, Bangladesh; MS degree in Computer and Information Science from University of New Haven, CT, USA; and MBA degree from Quinnipiac University, CT, USA. Before joining to the PhD program, Muhammed worked as a Lecturer for more than two years at North South University, Dhaka, Bangladesh. He has several years of industry experience in database and web development areas. His main research work includes data mining, information retrieval, e-commerce, and maximizing visibility of objects in search queries.