

# Efficient Approximation of Optimization Queries Under Parametric Aggregation Constraints

**Sudipto Guha**  
University of Pennsylvania  
sudipto@cis.upenn.edu

**Divesh Srivastava**  
AT&T Labs-Research  
divesh@research.att.com

**Dimitrios Gunopoulos**  
University of California  
dg@cs.ucr.edu

**Michail Vlachos**  
University of California  
mvlachos@cs.ucr.edu

**Nick Koudas**  
AT&T Labs-Research  
koudas@research.att.com

## Abstract

We introduce and study a new class of queries that we refer to as OPAC (*optimization under parametric aggregation constraints*) queries. Such queries aim to identify sets of database tuples that constitute solutions of a large class of optimization problems involving the database tuples. The constraints and the objective function are specified in terms of aggregate functions of relational attributes, and the parameter values identify the constants used in the aggregation constraints.

We develop algorithms that preprocess relations and construct indices to efficiently provide answers to OPAC queries. The answers returned by our indices are approximate, not exact, and provide guarantees for their accuracy. Moreover, the indices can be tuned easily to meet desired accuracy levels, providing a graceful tradeoff between answer accuracy and index space. We present the results of a thorough experimental evaluation analyzing the impact of several parameters on the accuracy and performance of our techniques. Our results indicate that our methodology is effective and can be deployed easily, utilizing index structures such as R-trees.

## 1 Introduction

In today's rapidly changing business landscape, corporations increasingly rely on databases to help organize, manage and monitor every aspect of their business. Databases are deployed at the core of important business operations, including Customer Relationship Management, Supply Chain Management, and Decision Support Systems. The increasing com-

plexity of the ways in which businesses use databases creates an ongoing demand for sophisticated query capabilities. Novel types of queries seek to enhance the way information is utilized, while ensuring that they can be easily realized in a relational database environment without the need for significant modifications to the underlying relational engine. Indeed, over the years, several proposals enhancing the query capabilities of relational systems have been made. Recent examples include *preference queries*, which incorporate qualitative and quantitative user preferences [1, 3, 13, 8, 17] and *top-k queries* [10, 9, 2].

In this paper, we initiate the study of a new class of queries that we refer to as OPAC (*optimization under parametric aggregation constraints*) queries. Such queries aim to identify sets of database tuples that constitute solutions of a large class of optimization problems involving the database tuples. To illustrate this important class of queries, consider the following simple example.

**Example 1** Consider a large distributor of cables, who maintains a database relation  $R$  keeping track of the products in stock. Cable manufacturers ship their products in units, each having a specific weight and length. Assume that relation  $R$  has attributes  $uid$  (a unit identifier), manufacturer, weight, length and price, associated with each cable unit. A sample relation  $R$  is depicted in Figure 1.

Commonly, "queries" select cable units by imposing constraints on the total length and total weight of the units they are interested in, while optimizing on total price. Thus, the desired result is a set of tuples collectively meeting the imposed aggregate constraints and satisfying the objective function. Note that this is considerably different from selecting cable units (tuples) based on their individual attribute values.

For example, one query could request the set of cable units having the smallest total price, with total length no less than  $L_c = 90$  and total weight no less than  $W_c = 50$ . A straightforward solution to this query involves computing the total weight and length of each possible subset of cable units in  $R$ , identifying those that respect the constraints on length and weight, and returning the one with the lowest price. Clearly, such a brute force evaluation strategy is not desirable. In the example of Figure 1, the answer set for this query would be

---

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Uid	Manufacturer	Weight	Length	Price
1	Optical Co.	30	40	50
2	Optical Co.	20	50	50
3	Optics Inc.	30	70	80
4	Opticom Co.	20	20	10
5	Optics Inc.	20	20	20

Figure 1: Sample Relation  $R$

$\{uid_2, uid_4, uid_5\}$ , with a total price of 80.

A different query could seek to maximize the total price for a number of cable units requested, of total length no more than  $L_c = 90$  and of total weight no more than  $W_c = 50$ . In this case, the answer set for this query would be  $\{uid_1, uid_2\}$  or  $\{uid_3, uid_5\}$  each with a total price of 100.

Finally, observe that  $L_c$  and  $W_c$  are parameters of these two OPAC queries, and different users may be interested in these queries, but with different values specified for each of these parameters. ■

Instances of OPAC queries are ubiquitous in a variety of scenarios, including simple supplier-buyer scenarios (as illustrated by our example), that use relational data stores. They easily generalize to more complex scenarios involving Business to Business interactions in an electronic marketplace. Any interaction with a database, requesting a set of tuples as an answer, specifying constraints over aggregates of attributes values, seeking to optimize aggregate functions on some measure attribute in the result set, is an instance of an OPAC query.

OPAC queries have a very natural mathematical interpretation. In particular, they represent instances of optimization problems with multiple constraints [7], involving the tuples and attributes of a database relation. Although such problems have been extensively studied in the combinatorial optimization literature, there has been no work (to the best of our knowledge) exploring the possibility of using database technology to efficiently identify the set of tuples that constitute solutions to OPAC queries, when the relevant data resides in a database relation.

In this paper, we begin a formal study of the efficient execution of OPAC queries over relational databases. Our work is the first to address this important problem from a database perspective, and we make the following contributions:

- We introduce the class of OPAC queries as an important novel query type in a relational setting.
- We develop and analyze efficient algorithms that preprocess relations, and construct effective indices (R-trees), in order to facilitate the execution of OPAC queries. The answers returned by our indices are not exact, but approximate; however, we give quality guarantees, providing the flexibility to trade answer accuracy for index space.
- We present the results of a thorough experimental evaluation, demonstrating that our technique is effective, accurate and efficiently provides answers to OPAC queries.

This paper is organized as follows. In section 2, we present definitions and background material necessary for the rest of

the paper. Section 3 formally defines the problems we address in this paper. In section 4, we present our techniques for preprocessing relations to efficiently answer OPAC queries. In section 5, we experimentally evaluate our techniques varying important parameters of interest. Section 6 reviews related work and finally section 7 summarizes the paper and discusses avenues for further research.

## 2 Definitions

Let  $R(A_1, \dots, A_n, P)$  be a relation, with attributes  $A_1, \dots, A_n, P$ . Without loss of generality assume that all attributes have the same domain. Denote by  $S$  a subset of the tuples of  $R$  and by  $S_{A_i}, 1 \leq i \leq n$ , and  $S_P$ , the (multiset of) values of attribute  $A_i, 1 \leq i \leq n$  and  $P$  in  $S$ , respectively. Let  $f_i, 1 \leq i \leq n$  and  $\mathcal{F}$  denote aggregate functions (e.g., sum, max). We consider atomic aggregation constraints of the form  $f_i(S_{A_i}) \theta c_i$ , where  $\theta$  is an arithmetic comparison operator (e.g.,  $\leq, \geq$ ), and  $c_i$  is a constant [16], and complex aggregation constraints that are boolean combinations of atomic aggregation constraints; we refer to them collectively as aggregation constraints, denoted by  $\psi$ .

**Definition 1 (General OPAC Query Problem)** Given a relation  $R(A_1, \dots, A_n, P)$ , a general OPAC query  $Q$  specifies (i) a parametric aggregation constraint  $\psi_{\vec{v}}$ , (ii) an aggregate function  $\mathcal{F}$ , with optimization objective  $m$  (min or max), and (iii) a vector of constants  $\vec{c}$ . It returns a subset  $S$  of tuples from  $R$  as its result, such that (i)  $\psi_{\vec{v}=\vec{c}}(S_{A_1}, \dots, S_{A_n}) \equiv \text{TRUE}$ , and (ii)  $\forall S' \subseteq R, (\psi_{\vec{v}=\vec{c}}(S'_{A_1}, \dots, S'_{A_n}) \equiv \text{TRUE}) \Rightarrow (\mathcal{F}(S'_P) \leq_m \mathcal{F}(S_P))$ . ■

Intuitively, the result of a general OPAC query  $Q$  is a subset  $S$  of tuples of  $R$  that satisfy the parametric aggregation constraint  $\psi_{\vec{v}}$  (with the parameters  $\vec{v}$  instantiated to the vector of constants  $\vec{c}$ ), such that its aggregate objective function is optimal (i.e., maximal under  $\leq_m$ ) among all subsets of  $R$  that satisfy the (instantiated) parametric aggregation constraint.

It is evident that the result of a general OPAC query involves the solution of an optimization problem involving a (potentially) complex aggregation constraint on relation  $R$ . Depending on the specifics of the aggregate functions  $f_i, \mathcal{F}$ , the nature of the aggregation constraint, and the optimization objective, different instances of the OPAC query problem arise. For suitable choices of these it might be feasible to efficiently obtain a solution. In the general case, however, the problem is computationally infeasible (NP-hard).

In this paper, we consider the important instance of the problem when the aggregate functions  $f_i, \mathcal{F}$  return the sum of the values in their input multisets, the aggregation constraints are conjunctions of atomic aggregation constraints of the form  $f_i(S_{A_i}) \leq c_i$ , and the objective function seeks to maximize  $\mathcal{F}(S_P)$ .

This formulation of an OPAC query gives rise to a well-known optimization problem, namely the *multi-attribute knapsack* problem [7, 18]. Given this relationship between the specific form of the OPAC query on which we focus our presentation and the multi-attribute knapsack problem, we will refer to values of the function  $\mathcal{F}(S_P)$  as the *profit* for the set of tuples  $S$ . It is well-known that solving the knapsack problem, even in the simple instance involving a constraint on only one attribute (e.g.,  $\sum_{x_i \in S_{A_1}} x_i \leq c_1$  and maximize  $\sum_{x_j \in S_P} x_j$ )

is NP-complete. However, this problem is solvable in pseudo-polynomial time with dynamic programming [6]. The multi-attribute knapsack problem has been extensively studied in the literature (e.g., see [7, 18] and references therein) and many approaches have been proposed for its solution. For example, the pseudo-polynomial algorithm solving the knapsack problem in the single attribute case can serve as a basis for a solution of the multi-attribute problem as well. In particular, one could generate all solutions for one attribute, and pick the solution  $S$  that maximizes  $\mathcal{F}(S_P)$  among all solutions that satisfy the constraints on all attributes. The form of the solution that is reported could vary; for example, the solution could be the set of tuple identifiers from  $R$ . Several other approaches for the solution of the multi-attribute knapsack problem are available in the literature (e.g., [4, 18] and references therein).

It is evident that every OPAC query  $Q$  determines an instance of a multi-attribute knapsack problem on relation  $R$ . Since the relation  $R$  can be very large, in the general case, solving the multi-attribute knapsack problem from scratch every time an OPAC query is posed is not at all pragmatic. Such an approach would be far from being interactive and, more importantly, it would be entirely DBMS agnostic, missing the opportunity to utilize the underlying DBMS infrastructure for query answering. We wish to alleviate these shortcomings and provide efficient answers to OPAC queries utilizing DBMS concepts and techniques.

We conclude this section by briefly introducing the following concepts, from the optimization literature, that will be useful in what follows. In optimization problems involving multiple objective functions, the concept of the *Pareto* (or, *dominating*) set has been proposed as the right solution framework for optimization problems in general, and multi-attribute knapsack problems in particular. In this setting, the Pareto set is the set of optimal solutions that are mutually incomparable, because improving one objective would lead to a decrease in another.

In our setting, we consider a single optimization objective, but we allow the user to dynamically specify the aggregation constraint parameters. Thus, we can adapt the Pareto framework to the OPAC query problem.

**Definition 2 (Pareto Set)** *The Pareto set  $\mathcal{P}$  for an OPAC query defined on a relation  $R$  is the set of pairs  $\{(\bar{c}, S)\}$  of all  $n$ -dimensional vectors  $\bar{c} = (c_1, \dots, c_n)$  and associated solutions  $S$ , such that (a) there exists a solution  $S \subseteq R$  with  $f_i(S_{A_i}) = c_i, 1 \leq i \leq n$ , and (b) there is no other pair  $(\bar{c}', S')$ , such that  $f_i(S'_{A_i}) = c'_i, c'_i \leq c_i, 1 \leq i \leq n$ , and  $\mathcal{F}(S'_P) > \mathcal{F}(S_P)$ .* ■

This is an appealing concept since the Pareto set will contain all the “interesting” solutions. What makes such solutions interesting is that they are *optimal* both in terms of the parameters realizing them and the profit obtained. For any element  $(\bar{c}, S)$  of the Pareto set, there is no other solution with higher profit achieved by parameters at most as large in all dimensions as  $\bar{c}$ . Identifying such a set would be very informative as it contains valuable information about maximal profits.

**Example 2** *Given the relation  $S$  in Table 1, the Pareto points are the round points in Figure 2. For example,  $(24, 24)$  is a Pareto point with profit 220, realized by the entire set of tuples.*

Relation S	a1	a2	Profit
t1	9	11	100
t2	11	9	100
t3	4	4	20

Table 1: Relation S

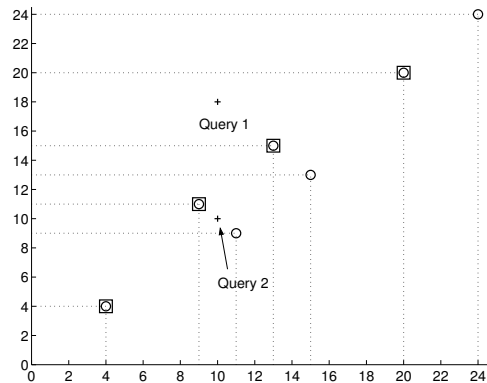


Figure 2: The Pareto set (round points), and an  $\epsilon$ -Pareto set (rectangular points) for the relation  $S$ .

*The vector  $(13, 15)$  with profit 120 is also a Pareto point, the corresponding set of tuples being  $\{t_1, t_3\}$ , because no vector  $(c_1, c_2)$  with  $c_1 \leq 13$  and  $c_2 \leq 15$  has profit more than 120.* ■

The notion of Pareto sets are defined for arbitrary classes of constraint problems and functions, not only for the multi-attribute knapsack. The constraint problems can be discrete and, in most such cases, the Pareto set can have exponentially many elements. This happens because linear programs can be posed in the Pareto framework, and the convex hull of the solution space for linear programs can have exponentially many (in the number of objects/variables) vertices.

The size of the Pareto set for an instance of the multi-attribute knapsack problem can be exponential in the number of tuples, even if the number of attributes is a small constant. Consider, for example, the case where there is only one attribute,  $A_1$  and a profit attribute  $P$ , and tuple  $i$  in relation  $R$  has the form  $(2^i, 2^i)$ . In this case, any subset of the tuples in  $R$  defines a unique cost and profit vector and no other set can achieve at least as small a cost and a higher profit. Therefore, all the subsets of tuples define dominating points.

To circumvent this problem, the concept of approximating the Pareto set has been introduced [14]. The  $\epsilon$ -Pareto set, is a set of “almost” optimal solutions defined as: for every optimal solution  $c$ , the  $\epsilon$ -Pareto set contains a solution that optimizes each of the optimization criteria within a fraction of  $\epsilon$ . They also show that the  $\epsilon$ -Pareto set for a multi-objective knapsack problem can be computed efficiently and it is polynomial in size [14]. It is therefore a very powerful way to argue about approximate solutions to multi-objective optimization problems.

Given a relation  $R$ , functions  $f_i$  and  $\mathcal{F}$ , and  $\epsilon > 0$ , the  $\epsilon$ -Pareto set,  $\mathcal{P}_\epsilon$  is a set of solutions that almost dominate any other solution.

**Definition 3 ( $\epsilon$ -Pareto set)** *The  $\epsilon$ -Pareto set for an OPAC*

query is a set of pairs  $\{(\bar{c}, S)\}$  of  $n$ -dimensional vectors  $\bar{c} = (c_1, \dots, c_n)$  and solutions  $S$ , such that, (a) there exists a solution  $S \subseteq R$  with  $f_i(S_{A_i}) \leq c_i, 1 \leq i \leq n$ , and (b) there is no other pair  $(\bar{c}', S')$ , such that  $f_i(S'_{A_i}) \leq c'_i, 1 \leq i \leq n, c'_i \leq (1 + \epsilon)c_i, 1 \leq i \leq n$  and  $\mathcal{F}(S'_P) > (1 + \epsilon)\mathcal{F}(S_P)$ . ■

**Example 3** If we have  $\epsilon = 0.25$ , the set of the rectangular points in Figure 2 is an  $\epsilon$ -Pareto set. For example, point  $(13, 15)$  is in the  $\epsilon$ -Pareto set because there is no vector with coordinates less than  $1.25 * (13, 15)$  that has profit more than  $120 * 1.25$ . ■

The concept of an  $\epsilon$ -Pareto is very useful. Assuming that the size of this set is manageable, one could materialize it and seek to utilize it for query answering. Following the treatment of [14] for  $\epsilon$ -Pareto sets, we can show the following:

**Theorem 1** The size of the  $\epsilon$ -Pareto set for an OPAC query instance defined on a relation  $R$  is polynomial in  $|R|$  (the size of  $R$ ) and  $\frac{1}{\epsilon}$ , but may be exponential in the number of attributes.

**Proof:** (Sketch) Assume that the  $n$  attributes of  $R$  are integers. Since  $f_i$  and  $\mathcal{F}$  are polynomial functions (and more specifically sums), the domain of each of these functions cannot be more than  $a^{|R|}$  for some constant  $a > 1$ . We can cover the space of  $[1, a^{|R|}]$  with a set of geometrically increasing intervals with step  $1 + \epsilon$ . To cover each domain we need  $O(p(\frac{|R|}{\epsilon}))$  intervals (for some polynomial  $p$ ). Taking the Cartesian product we get a total of  $O(p(\frac{|R|}{\epsilon})^n)$  hyper-rectangles. Clearly, taking one solution from the interior of each hyper-rectangle (if such a solution exists) results in an  $\epsilon$ -Pareto set. ■

### 3 Problem Statement

We will provide the description of a technique suitable for efficiently answering OPAC queries over a database. Our technique will be approximate but will provide guarantees for its accuracy, and expose useful tradeoffs.

Assume for a moment that we had complete knowledge of the collection of OPAC queries one would be interested in. In that case, a straightforward approach could precompute the answer of each query, assuming space was not an issue. In that scenario, any query  $Q$  could be answered efficiently by using the vector of constants  $\bar{c}$  provided by  $Q$  to retrieve the corresponding solution. Clearly, such a strategy is not feasible because exact knowledge of queries is not commonly available and the space overhead associated with such an approach could be prohibitive.

Our solution is to preprocess relation  $R$ , constructing index structures enabling efficient answers to arbitrary OPAC queries. For a query  $Q$ , we wish to provide either the exact answer, or an answer that is guaranteed accurate, for suitably defined notions of accuracy. Moreover, our construction will expose a tradeoff between accuracy and space, providing the flexibility to fine tune the accuracy of our answers. We quantify the accuracy of answers to an OPAC query as follows:

**Definition 4 ( $\epsilon, \epsilon'$ -Accurate Answers)** Let  $Q$  be an OPAC query specifying a vector of constants  $\bar{c} = (c_1, \dots, c_n)$ , having an answer  $S$  with profit  $P$ . For any  $\epsilon, \epsilon' > 0$ , an  $\epsilon, \epsilon'$ -Accurate answer to  $Q$ , is a vector  $\bar{c}' = (c'_1, \dots, c'_n)$  and an

answer set  $S'$ , such that  $\forall i, 1 \leq i \leq n, c'_i \leq (1 + \epsilon)c_i$  and  $P'(1 + \epsilon') > P$ , where  $P'$  is the profit of an OPAC query specifying vector  $\bar{c}'$  of constants. ■

Assume that  $Q$  is a query specifying a vector  $\bar{c}$  of constants and that the answer to  $Q$  is a set  $S \subseteq R$  with maximum profit  $P$ . An  $\epsilon, \epsilon'$ -accurate answer to  $Q$  is an answer set  $S'$  that is either the exact answer set  $S$  or it is an answer set corresponding to a query  $Q'$ . Query  $Q'$  specifies a vector of constants having values in each dimension less than or equal to  $1 + \epsilon$  of the corresponding values specified by  $Q$ . Moreover, the profit of  $Q'$  is strictly higher than a fraction of  $1 + \epsilon'$  of  $P$ . In the definition, without loss of generality, we assume the same  $\epsilon$  fraction is used for all constant values. Different values for  $\epsilon$  can be used for each of the values, if this is desirable,  $\epsilon$  being defined as a vector in this case. In fact, we do specify a different approximation factor,  $\epsilon'$ , for the profit, to differentiate between the aggregate functions  $f$  and  $\mathcal{F}$ .

We will preprocess relation  $R$ , constructing an index providing  $\epsilon, \epsilon'$ -accurate answers to OPAC queries. Our preprocessing will consist of solving the multi-attribute knapsack problem exactly, for a select subset of the candidate query space of all possible OPAC queries. We will then utilize these solutions towards providing  $\epsilon, \epsilon'$ -accurate answers to any candidate OPAC query on  $R$ . This gives rise to the main problem we address in the paper:

**Problem 1 (Efficient OPAC Query Answering)** Given a relation  $R$ , an OPAC query without the vector of constants  $Q$  and  $\epsilon, \epsilon'$ , preprocess  $R$  constructing an index being able to efficiently provide  $\epsilon, \epsilon'$ -accurate answers to any OPAC query on  $R$  that provides the parameters (values) to the constant vector  $Q$ . ■

**Example 4** Consider the example of Figure 2 again. Assume  $\epsilon = \epsilon' = 0.25$ . Assume that we are given the query  $(10, 18)$ , that is, find a set of objects that satisfy these conditions and maximize the Profit. The set  $\{t_1\}$  is an  $\epsilon, \epsilon'$ -accurate answer, because it satisfies the constraints, and there is no other set that has higher profit even if we relax the constraints by  $\epsilon$ .

If the query was  $(10, 10)$ , the set  $\{t_1\}$  is again an  $\epsilon, \epsilon'$ -accurate answer. Although the set does not satisfy the query constraints, it satisfies the relaxed constraints  $((9, 11) \leq 1.25 * (10, 10))$ , and has the highest profit among all solutions that satisfy these constraints. ■

### 4 Efficient Answers to OPAC Queries

We will now present our solutions and main technical results for providing efficient answers to OPAC queries. We will describe our approach in the following steps:

- We will first present a technique to preprocess a relation  $R$ , evaluating solutions to a multi-attribute knapsack problem on  $R$ , for only a select number of vectors of constants.
- Following this preprocessing, we will then show how to utilize known indices (R-trees), to provide efficient  $\epsilon, \epsilon'$ -accurate answers to OPAC queries on  $R$ .
- Finally, we will discuss issues related to the correctness and completeness of our strategy.

## 4.1 Preprocessing $R$

For a relation  $R(A_1, \dots, A_n, P)$ , assume that the range of the  $\sum$  function applied on elements of each attribute  $A_i$  has range  $[0 \dots D]$ .<sup>1</sup>

Any candidate query  $Q$  specifies an  $n$ -dimensional vector of constants  $\bar{c} \in [0 \dots D]^n$ . We will preprocess the space  $[0 \dots D]^n$  of all vectors of constants that can be specified by a possible query, creating a number of partitions that aim to cover the space of all possible queries. The partitions will be constructed in a way such that, for all possible queries inside a partition, one can reason collectively about the properties and values of function  $\mathcal{F}$ . Moreover, it will allow us to derive an  $\epsilon, \epsilon'$ -accurate answer for any query falling inside a partition.

We start by examining the relationship between the vectors of constants and the values of function  $\mathcal{F}$ . We first define the following property between  $n$ -dimensional vectors of constants:

**Definition 5** Let  $\bar{c} = (c_1, \dots, c_n), \bar{c}' = (c'_1, \dots, c'_n)$  be two  $n$ -dimensional vectors of constants. We say that  $\bar{c}$  is dominated by  $\bar{c}'$ , ( $\bar{c} \ll \bar{c}'$ ) if  $c_i \leq c'_i, 1 \leq i \leq n$ . ■

We then make the following observation:

**Observation 1** Let  $Q, Q'$  be two queries on  $R$ , specifying vectors of constants  $\bar{c}, \bar{c}'$ , having result sets  $S, S'$  respectively. If  $\bar{c} \ll \bar{c}'$  then  $\mathcal{F}(S_P) = \sum_{x_j \in S_P} x_j \leq \mathcal{F}(S'_P) = \sum_{x_j \in S'_P} x_j$ . ■

Thus, if a vector of constants  $\bar{c}$  is dominated by a vector  $\bar{c}'$ , the profit one can achieve for  $\bar{c}$  is less than or equal to the profit one can achieve using the vector  $\bar{c}'$ . A consequence of observation 1 is the following: Consider a sequence of queries, with vectors of constants,  $\bar{c}_1 \ll \bar{c}_2 \dots \ll \bar{c}_m$ . Observing the evolution of the values of  $\mathcal{F}$  in each answer obtained starting from  $\bar{c}_m$  moving towards  $\bar{c}_1$ , function  $\mathcal{F}$  is monotonically non increasing. Our technique will trace the evolution of function  $\mathcal{F}$  along such sequences of dominated vectors. In order to be able to provide  $\epsilon, \epsilon'$ -accurate answers, one has to identify vectors of constants that cause the value of function  $\mathcal{F}$  to change by an  $\epsilon'$  fraction. At the same time, the coordinates of such vectors have to be related by  $\epsilon$  as required by  $\epsilon, \epsilon'$ -accurate answers.

Let  $[0 \dots D]^n$  be the domain of all possible vectors of constants and consider one of these vectors,  $\bar{c} \in [0 \dots D]^n$ . Let  $S_{\bar{c}}$  be the solution to the query with vector of constants  $\bar{c}$  and  $\mathcal{F}(S_{\bar{c}})$  be the associated profit. We will aim to identify the vector of constants  $\bar{c}'$  by manipulating the coordinates of vector  $\bar{c}$  by fractions of  $1 + \epsilon$ , such that (a)  $\bar{c}' \ll \bar{c}$ , (b)  $(1 + \epsilon')\mathcal{F}(S_{\bar{c}'}) > \mathcal{F}(S_{\bar{c}})$ , where  $S_{\bar{c}'}$  the solution to the OPAC query with vector of constants  $\bar{c}'$  and (c) vector  $\bar{c}'$  is minimal. Consider the hyper rectangle defined by vectors  $\bar{c}'$  and  $\bar{c}$ . By definition, any query with vector of constants inside the hyper rectangle has  $\bar{c}'$  as an  $\epsilon, \epsilon'$ -answer.

Algorithm *GeneratePartitions* is shown in Figure 3. Given the domain of possible vectors of constants  $[0 \dots D]^n$ ,

it starts exploring the space by considering the vector corresponding to the upper right corner of the space. This is a vector  $\bar{c}$  that dominates all other vectors and consequently according to observation 1, corresponds to an OPAC query having the maximum profit. This vector is inserted to a queue and the algorithm iterates while the queue is not empty. For each vector  $\bar{c}$  in the queue, the algorithm aims to construct an  $\epsilon, \epsilon'$ -accurate answer for it (and subsequently for each vector dominated by  $\bar{c}$ ). The algorithm invokes function *LocateSolutions* with parameter  $\bar{c}$  (line (3) in Algorithm *GeneratePartitions*). This function returns a hyper rectangle  $r$  corresponding to a region of space of  $[0 \dots D]^n$ , a solution  $S$ , a vector  $\bar{C}$  and a profit  $p$ . The semantics associated with this result is that  $S$  is the solution to a query specifying vector of constants  $\bar{C}$  having profit  $p$  and forms an  $\epsilon, \epsilon'$ -accurate answer for each vector of constants inside  $r$ . The hyper rectangle is inserted in a multidimensional data structure, such as an R-tree, along with the associated profit, vector  $\bar{C}$  and solution  $S$ . Along with the leaf index entry for  $r$ , the profit  $p$  and vector  $\bar{C}$  are stored as well as a pointer, pointing to the solution (set of tuple identifiers)  $S$  on disk. Consequently, the index acts as a secondary structure pointing to  $\epsilon, \epsilon'$ -accurate answers on disk. Finally, (in line (6)) function *GeneratePartitions* generates a set of  $n$  vectors of constants. This set is constructed in a way such that that no vector in *Vec* dominates another; namely  $\forall \bar{c}', \bar{c}'' \in \text{Vec}, \bar{c}' \not\ll \bar{c}''$ . This set of vectors is constructed by calling function *CreateFront*, which accepts as parameters the coordinates of the newly formed hyper rectangle  $r$  and the queue  $Q$ . Given a hyper-rectangle  $r$  with lower left corner  $\bar{c}' = (c'_1, \dots, c'_n)$ , and upper right corner  $\bar{c} = (c_1, \dots, c_n)$ , *CreateFront* creates a set of  $n$  vectors that together dominate the entire space dominated by  $\bar{c}$ , excluding the space spanned by  $r$ . This is the set of vectors  $\bar{c}_i = (c_1, \dots, c_{i-1}, c'_i, c_{i+1}, \dots, c_n), 1 \leq i \leq n$ . Each one of these vectors is inserted in the queue  $Q$ , unless it is already in the queue.

Function *LocateSolution* accepts as a parameter a vector  $\bar{c}$  and identifies a hyper rectangle  $r \subset [0 \dots D]^n$ . Let  $r$  correspond to a hyper rectangle defined by vectors  $(\bar{c}', \bar{c})$  (where  $\bar{c}'$  the lower left and  $\bar{c}$  the upper right corners);  $\bar{c}$  is the vector provided to *LocateSolution* at input and  $\bar{c}'$  a vector corresponding to a query having a profit no less than an  $1 + \epsilon'$  fraction of the profit of the query specifying vector of constants  $\bar{c}$ . Given a specific  $\bar{c}$ , the search for a  $\bar{c}'$  with the aforementioned properties is performed in function *LocateSolution*. Initially (line (1)) function *MultiKnapsack* (employing any pseudo polynomial algorithm for solving the multi attribute Knapsack problem) is called to determine a solution  $S$  (set of tuples) and the profit  $p$  to the query with vector of constants  $\bar{c}$ . Then, the vector  $\bar{c}'$  is formed by decreasing each coordinate of  $\bar{c}$  by an  $1 + \epsilon$  fraction and function *MultiKnapsack* is called again, this time with vector  $\bar{c}'$ , to determine the solution  $S'$  with profit  $p'$ , to a query with constant vector  $\bar{c}'$ . Two cases of interest arise:

- If the profit  $p'$  of the query with vector  $\bar{c}'$  is larger than an  $1 + \epsilon'$  fraction of the profit of the query with vector  $\bar{c}$ , then algorithm *LocateSolutions* attempts to minimize vector  $\bar{c}'$  by successively reducing its coordinates by a  $1 + \epsilon$

<sup>1</sup>Without loss of generality, assume all attributes have the same domain.

Algorithm **GeneratePartitions**( $\epsilon, \epsilon', D$ )

Initialize:

$Q$ : Queue of multidimensional constraint vectors

$\mathcal{R}$ : R-tree

$s, c, c'$ : constraint vectors

each coordinate of  $s$  is initially set to be equal to  $D$  and, and  $s$  is added to  $Q$

- (1) while  $Q$  not empty
- (2)  $\bar{c} = \text{headof}(Q)$
- (3)  $(r, \bar{C}, p, S) = \text{LocateSolution}(\bar{c})$
- (4) if there is no rectangle  $r'$  in the R-tree  $\mathcal{R}$  that contains rectangle  $r$  and  $r$  not NULL
- (5) Insert  $(r, p, \bar{C}, S)$  to the R-tree  $\mathcal{R}$  by storing  $(r, p, \bar{C})$  in a leaf index entry and maintaining a pointer to the set of tuple identifiers in the solution  $S$  on disk
- (6) CreateFront( $Q, r$ )
- (7) endif
- (8) end-while

Algorithm **LocateSolution**( $\bar{c}$ )

**Input:** constant vector  $\bar{c} = (c_1, \dots, c_n)$

**Output:**  $(r, \bar{C}, p, S)$

- (1)  $(p, S) = \text{MultiKnapsack}(\bar{c})$
- (2) if  $S$  is NULL return (NULL, NULL, 0, NULL)
- (3) for  $i = 1$  to  $n$
- (4)  $c'_i = \frac{c_i}{1+\epsilon}$
- (5)  $(p', S') = \text{MultiKnapsack}(\bar{c}')$
- (6) if  $(S'$  is NULL) return (NULL, NULL, 0, NULL)
- (7) if  $((1 + \epsilon')p' > p)$
- (8) while  $(p \geq \frac{p'}{1+\epsilon'})$
- (9)  $\bar{c}_i = \bar{c}'_i; p_i = p'_i; S_i = S'_i$
- (10) for  $i = 1$  to  $n$
- (11)  $c'_i = \frac{c'_i}{1+\epsilon'}$
- (12)  $(p', S') = \text{MultiKnapsack}(\bar{c}')$
- (13) end-while
- (14) return (FormRect( $\bar{c}_i, \bar{c}$ ),  $\bar{c}_i, p_i, S_i$ )
- (15) else
- (16) return (FormRect( $\bar{c}'_i, \bar{c}$ ),  $\bar{c}, p, S$ )

Figure 3: Algorithm *GeneratePartitions*

fraction, updating the solution  $S'$  and profit  $p'$  attainable (lines (5)-(12)). If it succeeds, the algorithm forms a hyper rectangle using the minimal vector  $\bar{c}'$  and  $\bar{c}$ , returning it along with vector  $\bar{c}'$ , the profit  $p'$  and solution  $S'$  (line (14)) of  $\bar{c}'$ .

- If, on the other hand, the profit of the query with vector  $\bar{c}'$  is smaller than an  $1 + \epsilon'$  fraction of the profit of the query with vector  $\bar{c}$ , then algorithm *LocateSolutions* does not attempt to reduce vector  $\bar{c}'$  further; it forms a hyper rectangle consisting of the  $(\bar{c}', \bar{c})$  returning it along with the associated profit  $p$ , vector  $\bar{c}$  and solution  $S$  of  $\bar{c}$  (line (16)).

Function *MultiKnapsack* with parameter  $\bar{c}$  is guaranteed to return a non empty solution, if a solution that satisfies the constraints exists. The Function will return a null solution if there is no subset of relation  $R$  satisfying the constraints imposed by vector  $\bar{c}$ . To formalize this notion we define the *feasible region* of relation  $R$ :

**Definition 6 (Feasible Region)** Let  $R(A_1, \dots, A_n, P)$  be a relation with tuples  $(A_1, \dots, A_n)$ . Assume that the  $\Sigma$  function applied on elements of each attribute  $A_i$  and  $P$  has range  $[0 \dots D]$  as well. The feasible region of  $R$  is the set of all vectors  $\bar{c}$  dominated by vector  $(D, \dots, D)$ , that dominate at least one tuple of  $R$ . ■

Algorithm *GeneratePartitions* progressively reduces the values in each dimension of vectors from the queue and eventually vectors generated by *CreateFront* will be outside the feasible region. As soon as a vector falls outside the feasible region of  $R$ , function *MultiKnapsack* returns null and progressively the number of elements in the queue decreases.

The following example illustrates the operation of the algorithm:

**Example 5** Consider the relation in Figure 4. There are only two tuples,  $t_1$  and  $t_2$ , and the range of the  $\Sigma$  function on both attributes is  $[0, 15]$ . Assume that  $\epsilon = \epsilon' = 0.25$ .

The algorithm starts at  $(15, 15)$  and finds that the solution at vector  $(15, 15)$  is  $\{t_1, t_2\}$  with Profit 200. The next vector that is investigated by *LocateSolution* is  $\frac{1}{1.25}(15, 15) = (12, 12)$  The best solution at  $(12, 12)$  is either  $\{t_1\}$ , or  $\{t_2\}$ , both with Profit 100. Since  $100 * 1.25 < 200$ , the algorithm does not extend this rectangle further. It adds the rectangle  $R1 = [(15, 15), (12, 12)]$  in the R-tree, and associates with this rectangle the solution at the top corner, namely  $\{t_1, t_2\}$ , with Profit 200.

*CreateFront* then adds the following 2 points in the queue:  $(15, 12)$  and  $(12, 15)$ . At this point, the top of the queue is  $(15, 12)$ . *LocateSolution* finds that a solution for this point is either  $\{t_1\}$  or  $\{t_2\}$ , both with Profit 100. The algorithm finds the bottom corner, which is  $\frac{1}{1.25}(15, 12) = (12, 10)$  (rounding up the numbers to simplify the example) which still has a solution with profit 100, so it extends this to  $\frac{1}{1.25}(12, 10) = (10, 8)$  which still has the same solution  $\{t_1\}$ , and then to  $(8, 7)$ , which does not have any feasible solution. So at this point it backs up, and adds the rectangle  $R2 = [(15, 12), (10, 8)]$  with solution  $\{t_1\}$  and Profit 100 to the R-tree. *CreateFront* then adds the points  $(15, 8)$  and

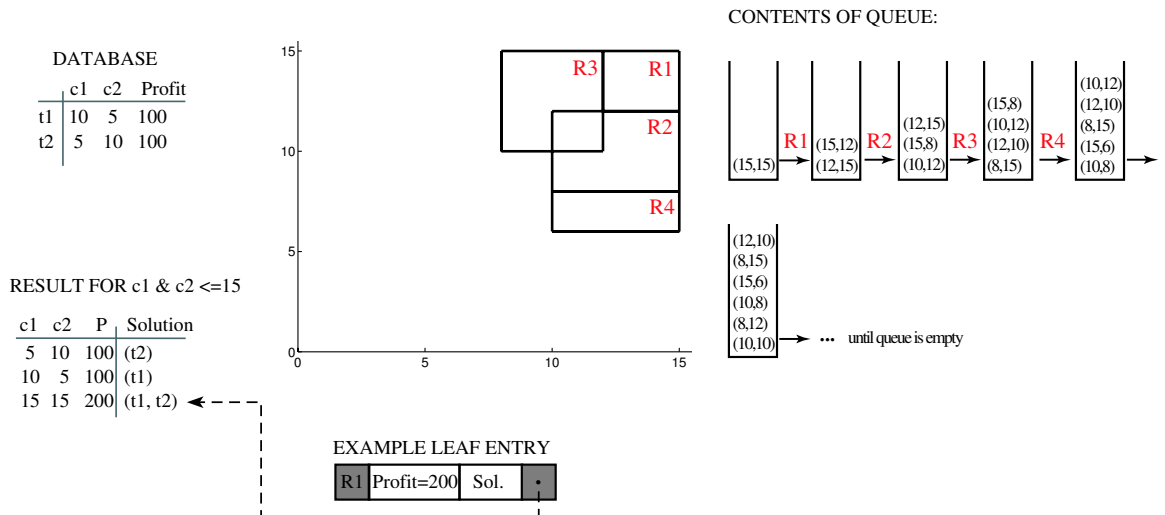


Figure 4: The operation of algorithm *GeneratePartitions*



Figure 5: Example partitioning of the space of two uniformly distributed attributes into a set of rectangles for a dataset with Gaussian profit distribution.

(10, 12) to the queue. Similarly, considering vector (12, 15) leads to the insertion of rectangle R3 in the R-tree, and vector (15, 8) leads to the insertion of rectangle R4. The next vector at the top of the queue is (10, 12), which is contained in rectangle R3. So no new rectangle is created, but the vectors (10, 10) and (8, 12) are inserted in the queue by *CreateFront*. The operation of the algorithm for this example is given in Figure 4. ■

Figure 5 presents an example of the partitioning generated by algorithm *GeneratePartitions*.

## 4.2 Query Answering

The outcome of algorithm *GeneratePartitions*, is a multi-dimensional index (e.g., an R-tree), providing access to a collection of hyper rectangles. For a query  $Q$  specifying a vector of constants  $\bar{c}$ , we obtain an  $\epsilon, \epsilon'$ -accurate answer as follows: We use  $\bar{c}$  and search the multidimensional index seeking hyper rectangles containing  $\bar{c}$  and let  $Ans$  be the set of hyper rectangles identified. Each of these hyper rectangles obtained from the leaves of the index, has a profit and a vector associated with it and points to a set of tuple identifiers on disk. Any

of these solutions is an  $\epsilon, \epsilon'$  answer and returning the vector, profit, and set of tuple identifiers associated with any hyper rectangle in  $Ans$ , suffices. In the case  $Ans$  is empty, then there is no feasible answer to  $Q$  in relation  $R$  and thus there is no possible  $\epsilon, \epsilon'$ -accurate answer.

**Example 6** Consider again the example of Figure 4. Assume we are given query (14, 14). This query vector falls in rectangle  $R1 = [(15, 15), (12, 12)]$ . We answer this query with the solution  $\{t_1, t_2\}$ , which is associated with R1, and which has Profit 200. This is a  $\epsilon, \epsilon'$ -accurate answer: From the definition of the  $\epsilon, \epsilon'$ -accurate answer, we have to return a solution which satisfies constraints  $c'_i < (1 + \epsilon)c_i$ , and profit  $P'(1 + \epsilon') > P$ . For the constraint vector (14, 14), the best solution is either  $\{t_1\}$  or  $\{t_2\}$ , with profit 100. Since  $15 \leq (1 + \epsilon)14 = 1.25 * 14$ , the solution  $\{t_1, t_2\}$  satisfies the relaxed constraints, and, since  $200 * 1.25 > 100$ , the profit constraint is satisfied as well. Intuitively, what happens is that, we do not give an exact answer; but we give an answer that is at least as good (and may in fact be much better) if we are willing to relax the constraints by a factor of  $\epsilon$ .

Let's assume a query (14, 9). This vector falls in rectangle R2. The solution we return is the one associated with rectangle r2:  $\{t_1\}$ , with Profit 100. In this case,  $c'_i < 1.25c_i$ , and  $P' = P$ , so this is a  $\epsilon, \epsilon'$ -accurate answer as well. ■

## 4.3 Correctness and Completeness

Algorithm *GeneratePartitions* guarantees that every feasible member of the space of all possible constant vectors will be contained in at least one hyper rectangle. The algorithm will cover the space of candidate query vectors using hyper rectangles. In particular:

**Theorem 2** Let  $\bar{c}$  be the constant vector associated with an OPAC query  $Q$ . Assume there exists a subset of tuples that satisfy the constraints  $\bar{c}$ . Then algorithm *GeneratePartitions*, creates at least one hyper rectangle containing vector  $\bar{c}$ .

**Proof:** Algorithm *GeneratePartitions* covers the entire feasible space: The first vector dominates the entire space.

Each iteration of the algorithm takes a vector from the queue, uses this vector to form the upper right corner of a new hyper rectangle, and adds a new set of vectors in the queue that together dominate the space that the original vector dominated with the exception of the space of the hyper-rectangle. Since the algorithm terminates when no vectors are in the queue, it follows that the entire feasible space is covered by hyper-rectangles. ■

The following result demonstrates that the answer to any OPAC query,  $Q$ , specifying a vector of constants  $\bar{c}$ , obtained from the index, is an  $\epsilon, \epsilon'$ -accurate answer.

**Theorem 3** *Let  $Q$  be a query specifying vector  $\bar{c}$  as a constant vector. Let  $r$  be a hyper rectangle containing  $\bar{c}$ , generated by algorithm *GeneratePartitions*. The answer to  $Q$  returned from the index, consisting of a vector, a set of tuples, and a profit is an  $\epsilon, \epsilon'$ -accurate answer.*

**Proof:** Let  $p_2$  be the profit of the lower left corner vector  $\bar{c}_2$ , and  $p_1$  be the profit of the upper right corner vector  $\bar{c}_1$  of multidimensional rectangle  $r$ . Assume a vector  $\bar{c}$  located inside  $r$ .

If  $p_2(1 + \epsilon') \geq p_1$ , then the lower left corner is an  $\epsilon, \epsilon'$ -accurate answer: since  $\bar{c}_2$  is dominated by  $\bar{c}_1$ , the optimal profit for  $\bar{c}$  is at most  $p_1$ , and therefore at most an  $1 + \epsilon'$  fraction higher than  $p_2$ . In this case algorithm *GeneratePartitions* stores in the index entry, along with  $r$ , a vector with coordinates equal to  $\bar{c}_2$  and a profit  $p_2$  and thus the answer returned is an  $\epsilon, \epsilon'$ -accurate answer.

If on the other hand  $p_2(1 + \epsilon') < p_1$ , then the profit of the lower left corner may be more than a fraction of  $1 + \epsilon'$  smaller than the profit of the query. However, by the construction of the algorithm *GeneratePartitions* this can only happen if all the values of the vector  $\bar{c}$  are within an  $1 + \epsilon$  fraction of the values of the upper right vector  $\bar{c}_1$ . In this case,  $\bar{c}_1$  provides an  $\epsilon, \epsilon'$ -accurate answer since it gives a much better profit with just an  $\epsilon$  relaxation of the constraints. Algorithm *GeneratePartitions* will associate the vector  $\bar{c}_1$  and its profit with the index entry, along with  $r$ . ■

Restricting our attention to monotone classes of aggregation functions, we can improve the computational aspects related to the construction of an  $\epsilon$ -Pareto set. The following theorem shows that the total number of hyper-rectangles represented in our index is polynomial.

**Theorem 4** *The number of vectors that create new multidimensional rectangles at any step of the execution of algorithm *GeneratePartitions*, is polynomial to  $\frac{1}{\epsilon}$ .*

**Proof:** Assume that the range of the  $\Sigma$  function applied on the  $A_i$  attribute values of a non-empty subset of the tuples is  $[1 \dots D]$  for each  $i$ . Here we assume that the attributes have non zero values; to deal with zero values we have to add the interval from zero to the smallest non zero value as one additional interval in the partition.

Then we can partition the range in  $O(\frac{\log D}{\log(1+\epsilon)}) = O(\frac{\log D}{\epsilon})$  intervals (for  $0 < \epsilon < 1$ ), geometrically increasing with step  $1 + \epsilon$ . Taking the Cartesian product of the  $n$  attributes, we create  $O((\frac{\log D}{\epsilon})^n)$   $n$  dimensional points. Note that, by the construction of the algorithm, every vector inserted in the queue

corresponds to one of these points. It follows that the number of hyper rectangles in the index is polynomial to  $1/\epsilon$  and to the size of the range of the  $\Sigma$  function applied on the attribute values, and is exponential to the number of the attributes. ■

In section 5 we will experimentally evaluate the effects data distribution has on the execution time of algorithm *GeneratePartitions*.

## 5 Experimental Evaluation

In this section we present the results of a comprehensive set of experiments, aiming to experimentally investigate the properties of algorithm *GeneratePartitions*. We seek to quantify the tradeoffs in terms of construction time and accuracy of our proposed techniques.

In our experiments we evaluate the impact of the parameters  $\epsilon$  and  $\epsilon'$  on the execution time of algorithm *GeneratePartitions*. We present scalability experiments, varying the number of tuples of the underlying relation, the sizes of the attribute domains and the number of attributes (dimensionality of the problem). We experimentally evaluate the accuracy of our approach. Finally we experimentally evaluate the efficiency of the technique, measuring the size of the index, and the query response time. All experiments were performed on an Athlon 1.3Ghz with 1Gb of memory and 60GB disk space.

### 5.1 Description of datasets

Our experimental test bed includes datasets with three distinct distributions in the profit attribute, namely *Uniform*, *Gaussian* and *Zipfian*. The rest of the attributes ( $A_1 \dots A_n$ ) on which constraints are posed are independently and normally distributed. We vary the number of these attributes from two (2D data sets) to three (3D data sets), effectively constructing three and four dimensional data spaces. For each profit attribute distribution we tested, we also produced data sets, introducing correlations between the attributes  $A_1 \dots A_n$ . For correlated attributes the correlation coefficient ranged between 0.7-0.8. Therefore, we had at our disposal a large collection of diverse datasets, that helped us understand and quantify the effect of the various parameters on the performance of our techniques.

### 5.2 Index construction time

With this set of experiments, we evaluate the impact of the parameters  $\epsilon$  and  $\epsilon'$ , the dataset distribution, attribute correlation, and dataset size on the total index construction time. This time consists of two distinct components:

- **MultiKnapsack Execution Time.** This is the total time required by all invocations to the MultiKnapsack function in algorithm *GeneratePartitions*. This time depends on the distributional characteristics of the data sets and their dimensionality.
- **Partition Generation Time.** It includes the time required to cover the domain space with hyper-rectangles of solutions. This step is directly affected by the parameters  $\epsilon$  and  $\epsilon'$ , as well as the dimensionality of the underlying data space.

Figures 6 and 7 report the total index construction time for the 2D datasets (two attributes  $A_1, A_2$  and a profit attribute

$P$ ), using correlated and uncorrelated  $A_1 \dots A_n$  attributes for Zipfian distributions. The values of the attributes in all the experiments are between 1 and 30. The MultiKnapsack execution time is depicted by transparent bars that start from zero and span downwards, while the colored bars represent the partition generation time and are shown above the zero level. The total time, representing the total index construction time is the sum of the two parts in each bar. It is evident that the majority of time is spent in the execution of the MultiKnapsack function. In the presence of correlations between the attributes the running time does not exhibit a substantial increase. We note that the execution time of function MultiKnapsack as well as its performance trends for various data sets, are pertinent to the specific method we used to solve this optimization problem. Any of the known and efficient techniques in the literature [7] can be used to implement this function.

We note from figures 6 and 7 that the construction time is polynomial to  $1/\epsilon$  and to  $1/\epsilon'$ . We also note that the construction time is similar in the case of correlated and uncorrelated attributes. Due to space limitations, in the remainder of this section, we present our results only for datasets with independently and uniformly distributed values in attributes  $A_1 \dots A_n$ , varying the distributions in the profit attribute  $P$ , and assuming  $\epsilon = \epsilon'$ .

Figures 8 and 9 report the results of the experiments varying  $\epsilon$  for 2D and 3D data sets (2 or 3 attributes and a profit attribute) respectively. The execution of function MultiKnapsack generally dominates the total construction time. However, when the value of  $\epsilon$  becomes very small the time required to generate the partitions becomes significant. The previous experiments evaluated the impact of distributional characteristics of the data sets in the overall index construction time. In Figures 10 - 13, we report on experiments that investigate the impact of quantitative data set characteristics on the overall index construction time. In particular, we examine how the MultiKnapsack execution time and the partition generation time are impacted by: (a) the cardinality of the underlying relations (Figures 10 and 11), (b) the attribute domain size (Figures 12, 13), and (c) the number of attributes of the relation. For this reason, we keep the parameters  $\epsilon$  and  $\epsilon'$  equal to 0.1 and vary the above parameters observing their impact.

We make the following experimental observations:

1. The MultiKnapsack execution time is independent of  $\epsilon$ . This happens because in the implementation we ran the dynamic programming once, at the beginning of the index construction algorithm, and used the partial results during the index construction (Figures 8, 9).
2. The Partition Generation time is proportional to  $O((\frac{1}{\epsilon})^{n+1})$  where  $n$  is the number of attributes, as was expected from the analysis (Figures 8, 9).
3. The MultiKnapsack execution time is linear to the size of the dataset (number of tuples, Figures 10 and 11) and the size of the domain (Figures 12, 13). This is also expected, since the dynamic programming algorithm we are using is linear to both of these variables.
4. The Partition generation time is constant to the size of the dataset (Figures 10 and 11).
5. The Partition generation time increases with an increase of the domain size. Larger attribute domains increase the search space of the algorithm we adopted for the solution of the MultiKnapsack problem, therefore increasing

the overall time required for the solutions of the multiple knapsack problem. Moreover, the space that algorithm *GeneratePartitions* has to cover increases, impacting the time required to generate the partitioning (Figures 12, 13).

These experiments quantify the time required by this pre-processing step for index generation and construction. The advantage is that any subsequent user query, requires a simple probing into the multidimensional index (e.g., an R-tree) to obtain a solution. Without the presence of such an index the only alternative would have been the execution of the MultiKnapsack algorithm for every new query, incurring a very large overhead for any query.

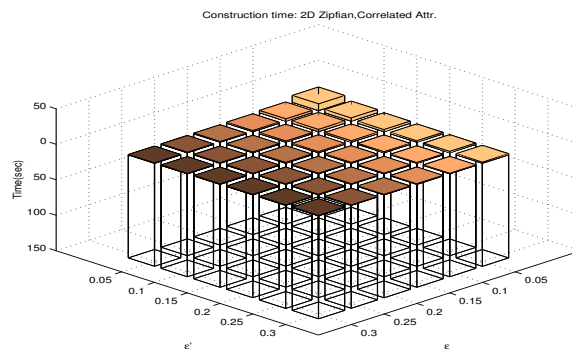


Figure 6: 2 constraints, correlated attributes, Zipf Distribution

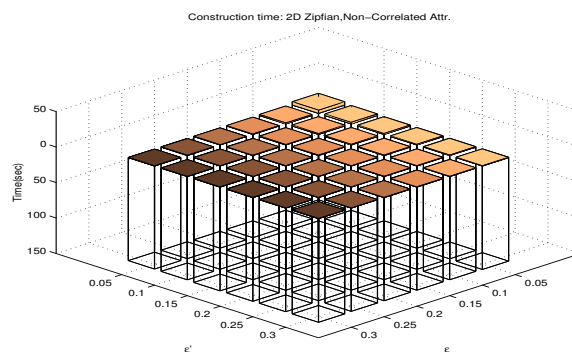


Figure 7: 2 constraints, uncorrelated attributes, Zipf Distribution

### 5.3 Evaluating the index accuracy and performance

Our index provides the guarantee of  $\epsilon, \epsilon'$ -accurate answers. In this section we demonstrate this fact empirically, and we investigate the effects of the accuracy guarantees on the index size.

We issue 25000 queries uniformly at random on the space of all possible queries and we report the average error of the profit returned by our index over the actual query profit (calculated by the MultiKnapsack algorithm). We define the average accuracy of a set of random queries as:

$$AvgAccuracy = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \left(1 - \frac{|Profit_{knapsack} - Profit_{index}|}{Profit_{knapsack}}\right)$$

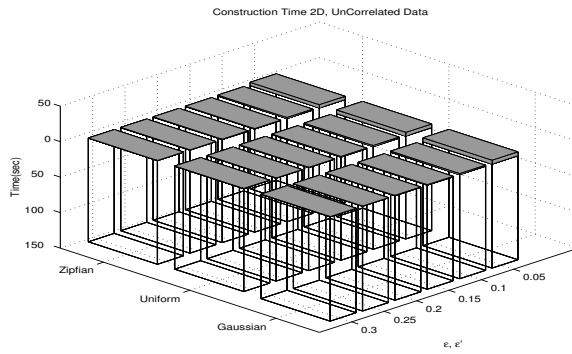


Figure 8: Scalability: Changing  $\epsilon$ . 2D datasets

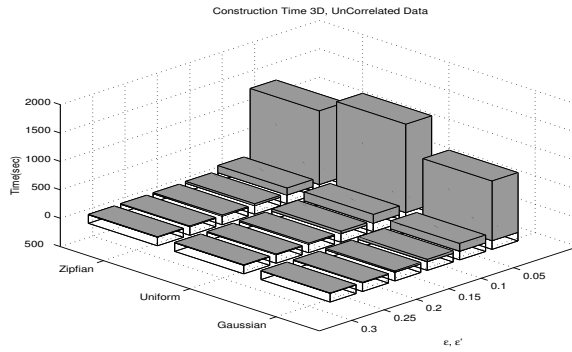


Figure 9: Scalability: Changing  $\epsilon$ . 3D datasets.

The expectation is that  $AvgAccuracy > 1/(1 + \epsilon')$  in accordance to the accuracy guarantees provided by our construction.

Figures 14 to 17 present the  $AvgAccuracy$  of the index for various values of  $\epsilon'$  and for different dataset (Figures 14 and 15) and domain (Figures 16 and 17) sizes. The lower manifold in each of these figures gives the size of the index built in each of the cases. The results are consistent with our theoretical analysis and the average reported error is below the error expected by the  $\epsilon'$  value specified at index construction time. It is interesting to observe that in all cases, for each  $\epsilon'$  value the ratio of the profit of the answer returned by our index to that of the actual query profit, is well above the worst case bound of  $1/(1 + \epsilon')$ , consistently across all profit distributions tested and data sets of increasing number of attributes. This empirically signifies, that the profit of the answer returned by our index, is close to the actual query profit.

With respect to the index size, we can make the following observations:

1. The size of the index is polynomial to  $1/\epsilon$ . This is clearly expected from the analysis, a smaller  $\epsilon$  requires more rectangles to cover the feasible space.
2. The index size depends on the domain size. The main reason is that we need more rectangles to cover the space. Also, the size of the  $\epsilon$ -Pareto set may increase in a larger domain. (Figures 16 and 17).
3. The size of the index does not generally depend on the number of tuples in the relation, as long as the domain size is constant (Figures 14 and 15).

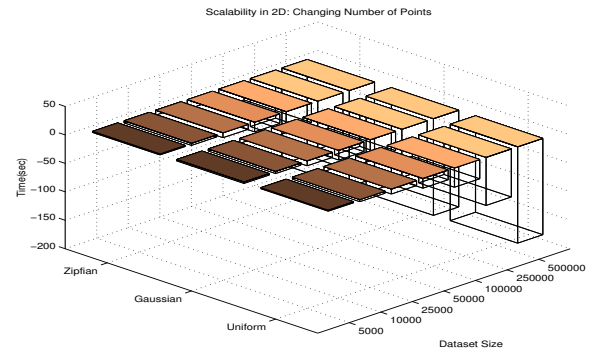


Figure 10: Scalability experiments: Changing the number of tuples. 2 constraints, uncorrelated attributes. The top bar is the time for the Partition generation, and the bottom bar the time for the MultiKnapsack procedure.

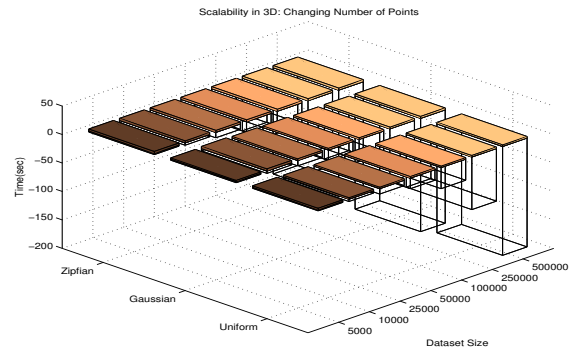


Figure 11: Scalability experiments: Changing the number of tuples. 3 constraints, uncorrelated attributes. The top bar is the time for the Partition generation, and the bottom bar the time for the MultiKnapsack procedure.

Finally, Figures 18, 19 give the average query response time for 25000 uniform random queries. We vary the dataset distributions, number of attributes,  $\epsilon$  values, and domain sizes. The experimental results show that the query response time depends mainly on the index size, and the main variables that affect that, namely the domain size and the number of attributes. Nevertheless, the query response time is small, and the reason is that the indices we build do not contain many overlapping hyper rectangles. Thus, the average number of rectangles intersected by a query in our experiments was less than two.

## 6 Related Work

We are not aware of work directly related to the work presented herein on OPAC Queries. Such queries are introduced as a novel query type seeking to provide greater query flexibility on top of relational data sources. Recently proposed, but not directly related, query types include preference queries [1, 12, 3, 13, 8, 17] and top-k queries [10, 9, 2].

The notion of *Pareto* optimality is discussed in the context of preference queries in database systems in [11]. A specialized form of Pareto optimality is introduced in which a user seeks the tuple with the highest values in a collection of select attributes, among all possible tuples in the database. Thus, in this specialized form, an answer is Pareto optimal, if it returns

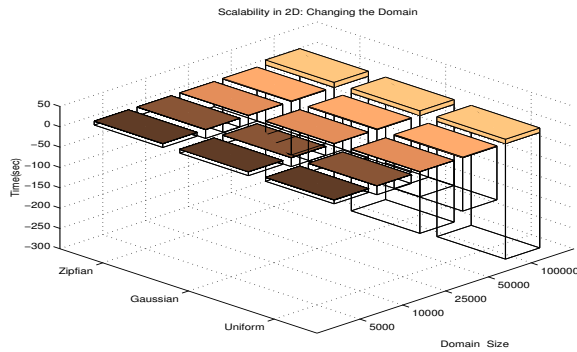


Figure 12: Scalability experiments: Changing the size of the attribute domain. 2 constraints, uncorrelated attributes. The top bar is the time for the Partition generation, and the bottom bar the time for the MultiKnapsack procedure.

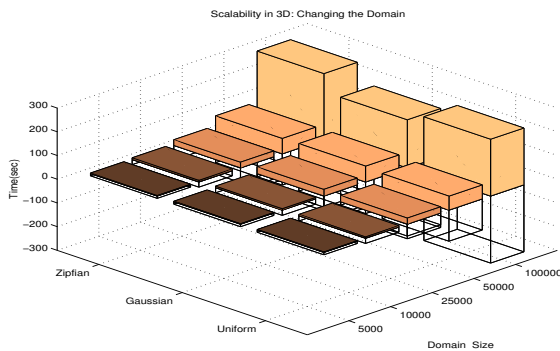


Figure 13: Scalability experiments: Changing the size of the attribute domain. 3 constraints, uncorrelated attributes. The top bar is the time for the Partition generation, and the bottom bar the time for the MultiKnapsack procedure.

the tuple that dominates all other tuples in the database, in a set of specified attributes.

There is a rich theory and literature related to solutions of optimization problems in the presence of constraints [7, 5]. [14] investigated the notion of  $\epsilon$ -Pareto, under arbitrary constraint problem classes, including the multi-attribute Knapsack. They characterized the conditions under which the  $\epsilon$ -Pareto set is polynomial in the number of variables/objects. In particular, they showed that for linear constraints the size of the Pareto set is polynomial in the number of variables, albeit exponential in the number of constraints. They gave an algorithm to construct a polynomial size  $\epsilon$ -Pareto set. However, due to its generality this technique can be very expensive regardless of the characteristics of the underlying data distributions. Subsequently, this concept was applied to query optimization [15].

## 7 Conclusions

We introduced a new class of queries, Optimization under Parametric Aggregation Constraint (OPAC) queries. Such queries aim to identify sets of database tuples constituting solutions to a large class of optimization problems involving the database tuples. We introduced algorithms that preprocess relations and construct indexes to efficiently provide answers to such queries. We analytically quantified the accuracy guar-

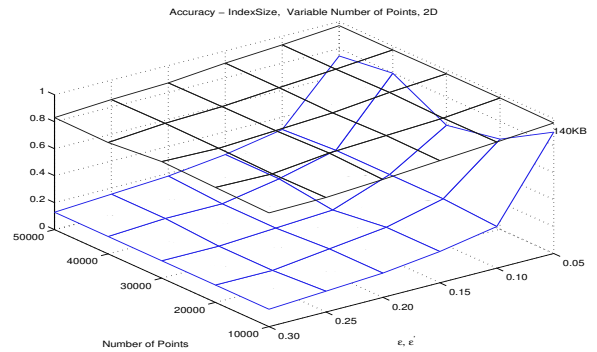


Figure 14: Accuracy/Index Size experiments: 2D data sets. Variables:  $\epsilon$ , dataset size. The top manifold gives average accuracy. The bottom manifold gives index size. The index size scale is on the right.

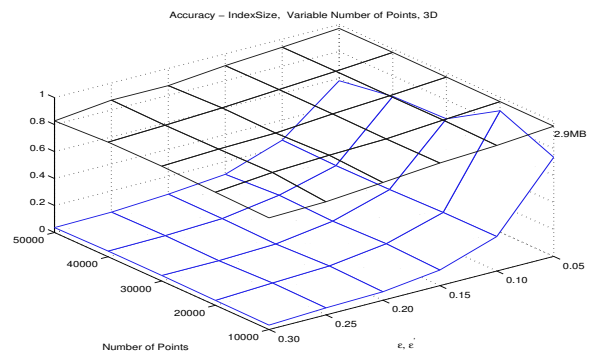


Figure 15: Accuracy/Index Size experiments: 3D data sets. Variables:  $\epsilon$ , dataset size. The top manifold gives average accuracy. The bottom manifold gives index size. The index size scale is on the right.

antees of our construction and presented a thorough evaluation highlighting the impact of various parameters on the performance of our schemes. Although we considered the case where the vector of constants of an OPAC query was fully specified our framework can handle various special cases as well. For example, the special case of an one dimensional vector of constants can easily be realized in our framework using B-trees. More generally, even when only some of the dimensions in the vector of constants are specified our framework can proceed by manipulating the hyper plane defined by this vector; we omit details due to lack of space.

This work raises new questions and opens avenues for additional work in this area. Considering this type of queries, in conjunction with other relational operators and addressing efficient processing and optimization issues is an intriguing research direction.

## References

- [1] R. Agrawal and E. Wimmers. A Framework For Expressing and Combining Preferences. *Proceedings of ACM SIGMOD*, pages 297–306, June 2000.
- [2] N. Bruno, L. Gravano, and A. Marian. Evaluating Top-k Queries Over Web Accessible Databases. *Proceedings of ICDE*, Apr. 2002.

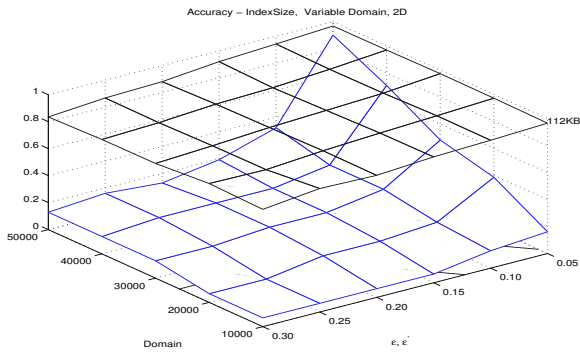


Figure 16: Accuracy/Index Size experiments: 2D data sets. Variables:  $\epsilon$ , domain size. The top manifold gives average accuracy. The bottom manifold gives index size. The index size scale is on the right.

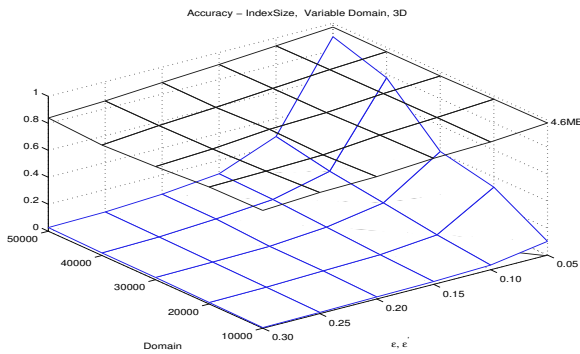


Figure 17: Accuracy/Index Size experiments: 3D data sets. Variables:  $\epsilon$ , domain size. The top manifold gives average accuracy. The bottom manifold gives index size. The index size scale is on the right.

[3] Y. Chang, L. Bergman, V. Castelli, C. Li, M. L. Lo, and J. Smith. The Onion Technique: Indexing for Linear Optimization Queries. *Proceedings of ACM SIGMOD*, pages 391–402, June 2000.

[4] C. Chekuri and S. Khanna. A PTAS For The Multiple Knapsack Problem. *Proceedings of SODA*, 2000.

[5] J. Climacao. Multicriteria Analysis. *Springer Verlag*, 1997.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw Hill and MIT Press, 1990.

[7] M. Ehrgott. Multicriteria Optimization. *Springer*, 2000.

[8] R. Fagin and E. Wimmers. Incorporating User Preferences in Multimedia Queries. *ICDT*, pages 247–261, Jan. 1997.

[9] L. Gravano and S. Chaudhuri. Evaluating Top-k Selection Queries. *Proceedings of VLDB*, Aug. 1999.

[10] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Efficient Execution of Multiparametric Ranked Queries. *Proceedings of SIGMOD*, June 2001.

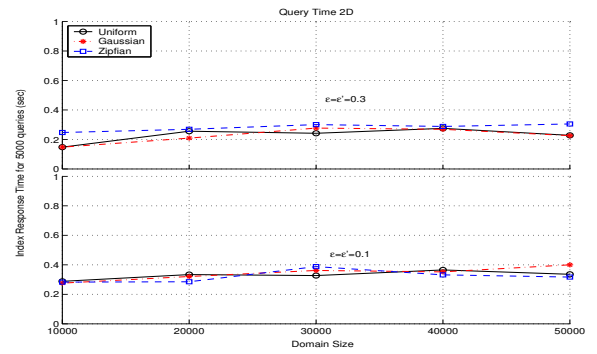


Figure 18: Query response time, 2D datasets

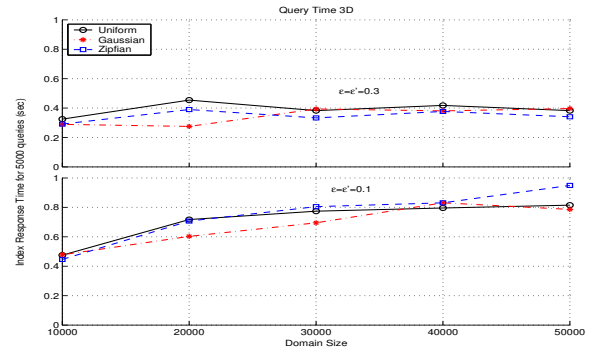


Figure 19: Query response time, 3D datasets

[11] W. Kiebling. Foundations of Preferences in Database Systems. *Proceedings of VLDB*, Aug. 2002.

[12] W. Kiebling and G. Kostler. Preference SQL: Design, Implementation, Experiences. *Proceedings of VLDB*, Aug. 2002.

[13] A. Natsev, Y.-C. Chang, J. Smith, C.-S. Li, and J. S. Vitter. Supporting Incremental Join Queries on Ranked Inputs. *Proceedings of VLDB*, Aug. 2001.

[14] C. Papadimitriou and M. Yannakakis. On the approximability of trade-offs and optimal access to web sources. *Proceedings of FOCS*, 2000.

[15] C. Papadimitriou and M. Yannakakis. Multiobjective Query Optimization. *Proceedings of ACM PODS*, June 2001.

[16] K. A. Ross, D. Srivastava, P. Stuckey, and S. Sudarshan. Foundations of aggregation constraints. *Theoretical Computer Science*, 193(1-2):149–179, 1998. An early version appeared in Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, 1994, LNCS 874.

[17] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava. Ranked Join Indices. *Proceedings IEEE ICDE*, Mar. 2003.

[18] E. Zitzler. Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications. *ETH Zurich PhD Thesis*, 1999.