Querying Improvement Strategies

Guolei Yang
Department of Computer Science
Iowa State University
Ames, IA, USA
yanggl@iastate.edu

Ying Cai
Department of Computer Science
Iowa State University
Ames, IA, USA
yingcai@iastate.edu

ABSTRACT

Given a set of objects and a set of top-k queries on these objects, we are interested in adjusting some object's attribute values to meet some requirements under certain cost constraints. We call such an adjustment an *improvement strategy*. Searching for cost-efficient improvement strategies is crucial for applications like product marketing, where top-k queries are used to model users' preference. We propose two types of *Improvement Queries (IQs)*. A *Min-Cost* IQ finds the improvement strategy that makes selected objects hit a desired number of queries with the minimal cost, while a *Max-Hit* IQ searches for the improvement strategy that makes selected objects hit as many queries as possible with a given budget. We show that answering IQs is NP-hard and develop a suite of heuristic algorithms. Our key idea is to interpret objects as functions and treat each top-k query as an input to the functions. The geometric relationship among the function intersections is then leveraged for efficient query processing. We implement the proposed algorithms as an analytic tool and integrate it with a DBMS, and they exhibit excellent performance on both synthetic and real-world data in experiments.

1. INTRODUCTION

Top-k query [7, 6, 11, 26] is widely used in applications like e-commerce for users to find objects (e.g., products) that best match their preference. A user's preference is represented by a *utility function* which computes a "score" for each object, and a top-k query retrieves the k objects with the highest/lowest scores. When an object appears in a query result, we say the object *hits* the query. Given a set of objects and a set of top-k queries, adjusting an object's attribute values could result in changing the number of queries it hits. In this paper, we refer to such an adjustment as an *improvement strategy*. We are interested in querying the improvement strategies for objects of interested under some cost constraint. We consider two variations of *Improvement Query* (IQ):

- **Min-Cost IQ:** Given a *cost function*, this type of IQ finds the most cost-efficient improvement strategy for an object to hit a given minimum number of top-k queries. Here a cost function is defined by the query issuer to measure the cost of adjusting attribute values of objects. The idea of modeling costs as math functions is a common approach [19, 4]. We allow query issuers to define their own cost functions.

- **Max-Hit IQ:** Given a *cost function* and a *budget*, this type of IQ returns the improvement strategy for an object to hit the maximal number of top-k queries under the condition that the total cost does not exceed the budget.

The problem of finding improvement strategies arises from a variety of applications. For example, a camera manufacturer may want to improve its product for more market shares. Here an improvement is a change of the product's features such as camera's resolution and price. Likewise, in a presidential election, it is imperative for the candidates to evaluate their campaign strategies from time to time, and adjust if needed, in order to appeal themselves to more voters. In these examples, there are a set of objects (e.g., products, presidential candidates) and a set of top-k queries, each representing the preference of a user (e.g., customer, voter), and we want to improve one or more objects (called *targets*) to hit as many queries as possible. Existing queries such as reverse top-k query [21], maximal rank query [14], and reverse k-ranks query [25] have been developed to provide information concerning an object's competitiveness in top-k selection. These queries, however, do not allow one to identify an improvement strategy, the focus of this paper.

The problem of processing IQs can be formulated as constrained optimization problems and we prove it is NP-hard. As such, finding accurate query results is computation intensive even for moderate size datasets. We address this problem by proposing a suite of heuristic algorithms. At the core of the proposed algorithms is a novel indexing technique. Our key idea is to 1) *interpret each object as a function*, and 2) *treat each top-k query as an input to these functions*. The *intersection* of two functions formulates a hyperplane in their domain. Given a set of functions, their intersection hyperplanes partition the domain into a number of *subdomains*. We observe that the rank of an object must be the same for all queries that fall in one subdomain. Applying an improvement strategy to an object will cause the boundary of some subdomains to change, but it will affect the result of a top-k query only if the query falls into a different subdomain. This observation allows us to develop a highly

efficient algorithm for IQ processing. We summarize our main contributions as follows:

- To our knowledge, this is the first to study the problem of object improvement, defined as adjusting the attribute values of the objects of interest. We prove the inherent intractability of the minimal cost/maximal hit improvement strategy searching problem.
- We propose the notion of *Improvement Query (IQ)*, which supplements the existing top-k query with the key information needed to develop effective improvement strategies. We propose two types of IQs: Min-Cost IQ and Max-Hit IQ. Given a user-defined cost function, the former IQ finds the most cost-efficient improvement strategy that achieves desired number of hits, while the latter one finds the improvement strategy that hits the maximal number of top-k queries with a given budget. We design efficient IQ processing algorithms based on a novel query indexing technique and an important observation.
- We implement the proposed techniques as an analytic tool and integrate it with the Database Management System (DBMS). The tool is thoroughly evaluated over synthetic and real-world data. The results show that our techniques demonstrate good performance, and the tool is scalable for large-scale users and objects.

The rest of the paper is organized as follows. We discuss related work in Section 2. In Section 3, we formally define the problem and give an overview of our solution. The proposed techniques for basic cases and complex scenarios are presented in Section 4 and Section 5 respectively. In Section 6, we describe our system implementation and present experiment results. We conclude the paper in Section 7.

2. RELATED WORK

Our work is closely related to the top-k query and other rank-aware queries. We briefly discuss some representative works as follows.

Top-k query: Several indexing techniques have been proposed for efficient processing of top-k queries. View-based techniques (e.g., [11, 8]) employs materialized views to retrieve top k, where objects are ranked according to arbitrary utility functions. Layer-based technique ([6]) computes the convex hulls of data points, and organizes them in layers. Top-k queries are then processed from the outmost layer, which contains objects that are most likely to be in top-k. The state-of-the-art technique is [26], which exploits the dominant relationship between objects. More specifically, an object p_i is said to *dominate* another object p_j if there exists no linear utility functions that ranks p_i lower than p_j. Thus there is no way for p_j to be include in a query result unless p_i is included first. As such, objects can be organized into groups based on their dominant relationship. This allows efficient processing of top-k queries. These techniques, however, are all limited to linear utility functions. The problem of non-linear utility function top-k selection is studied in [24] as k-constrained optimization problem, and addressed with a state-space indexing technique.

Other rank-aware queries: Top-k query has inspired a rich family of rank-aware queries, which are closely related to our research. Given a set of objects and a set of top-k queries, a reverse top-k query [20, 21] retrieves the queries whose result contains a selected object. For less popular objects, a useful variant of reverse top-k query is the reverse k-ranks query, which can find the k queries whose rank of an object is the highest among all queries. A maximum rank query [14] computes the highest possible rank an object can achieve for any utility function. Unlike our work, the maximum rank is not achieved by adjusting attributes of the object itself, but by exploring different utility functions. It can find the maximum rank one object can get with respect to any query, but cannot provide information on how to increase the number of queries that an object hits. This makes it fundamentally different from our problem. These existing queries help one understand the current competitiveness of an object among its peers, but not improve the object to make it more competitive.

Another related work is [13]. It considers how to find the k objects from a dataset that can be upgraded with minimal cost. The goal of *upgrade* is to make the object appear on *skyline* of the dataset. An object is said to be on skyline if it is not worse in all dimensions than another object in the dataset. Each dimension is compared independently and no function is computed, therefore making an object to be on skyline is straightforward, and the major challenge addressed in [13] is how to efficiently find the k objects with lowest upgrading cost without traversing all objects. In contrast, finding optimal improvement strategy for even one object is NP-hard. Their proposed algorithm cannot solve our problem. A similar work [22] discusses how to efficiently create new products that appear on skyline of a given dataset. But it does not consider improving existing objects, thus less related to our work.

3. PRELIMINARIES

3.1 Problem Definition

Consider a dataset D with n objects. Each object p_i is a point in the d-dimensional space, where each dimension represents a numerical attribute of the object. We use $p_i^{(j)}$ to denote its j-th dimension's value. Each dimension can be continuous or discrete, finite or infinite. Let $Q = \{q_1, q_2, ..., q_m\}$ denote a set of m top-k queries. Each query q_i $(1 \leq i \leq m)$ specifies a k value (i.e., the number of object to return) and a utility function which computes a score for each object. Together they represent a user's preference. The number of top-k queries hit by p_i is denoted by $H(p_i)$. We define improvement strategy as follows:

DEFINITION 1 (IMPROVEMENT STRATEGY). *An improvement strategy s for an object p_i is a d-dimensional vector $s = \{s_1, s_2, ..., s_d\}$, where $s_i \in \mathbb{R}$ specifies how the i-th attribute is to be adjusted, i.e., applying s to p_i will replace p_i with a new object p_i', where $p_i'^{(j)} = p_i^{(j)} + s_j$ $(1 \leq j \leq d)$.*

To illustrate, consider a camera dataset showed in Figure 1. Each camera has three discrete attributes *resolution*, *storage*, and *price*. Together they determine the camera's rank for a given top-k query. Let $s = \{5, 2, -50\}$ be an improvement strategy. Applying s on a camera means to increase the camera's resolution by 5 Megapixel, increase its storage by 2 GB, and decrease its price by $50. For example, applying s on camera p_1 will result in a new object $p_1' = \{15, 4, 200\}$. Note that after the improvement, p_1''s

Cameras

ID	*resolution (Megapixel)*	*storage (GB)*	*price ($)*
p_1	10	2	250
p_2	12	4	340
...	...	...	...

$\Downarrow$ *Applying* $s = \{5, 2, -50\}$ *to* p_1

ID	*resolution (Megapixel)*	*storage (GB)*	*price ($)*
p_1'	15	4	200
p_2	12	4	340
...	...	...	...

Top-k queries represent users' preference for camera

ID	*Utility function*	*top-k*
q_1	5.0**resolution* + 3.5**storage* - 0.05**price*	$k = 1$
q_2	2.5**resolution* + 7.0**storage* - 0.08**price*	$k = 1$
...	...	...

Figure 1: Example of improvement strategy for cameras

rank becomes higher than that of p_2 for both queries q_1 and q_2.

For ease of presentation, we will simply use $p_i' = p_i + s$ to denote the improved object p_i' that is derived by applying s on p_i. An improvement strategy aims to make a target object appear in more query results. Given an improvement strategy s, we measure its effectiveness in improving object p_i as the *number of top-k queries hit by* $p_i' = p_i + s$, denoted by $H(p_i')$. A larger $H(p_i')$ means more effective that s is in improving p_i.

Improving an object requires resources such as time and money. We let the query issuer specify such resource requirements using a *cost function* $Cost_{p_i}(s)$, which computes the cost of applying strategy s to object p_i. There is rich literature on how to model product costs using math functions and interested readers are referred to [19, 4, 2] for details. Here we simply assume the cost functions are provided by the query issuer. Our research is aimed at finding two kinds of improvement strategies:

Definition 2 (Min-Cost Improvement Strategy). *Given an improvement goal that is to hit at least* $\tau \in \mathbb{I}$ *queries, an improvement strategy* s *for* p_i *is a* ***minimal cost improvement strategy*** *w.r.t. some cost function* $Cost_{p_i}$ *if* $H(p_i + s) \geq \tau$ *and* $Cost_{p_i}(s)$ *is minimized.*

Definition 3 (Max-Hit Improvement Strategy). *Given a budget* $\beta \in \mathbb{R}$*, an improvement strategy* s *for* p_i *is a* ***maximal hit improvement strategy*** *w.r.t. some cost function* $Cost_{p_i}$ *if* $Cost_{p_i}(s) \leq \beta$ *and* $H(p_i + s)$ *is maximized.*

Accordingly, we define two types of *Improvement Queries* (IQs). A *Min-cost IQ* let user query minimal cost improvement strategies for selected objects. Similarly, a *Max-Hit IQ* returns the maximal hit improvement strategies. We will show later in Section 4 that searching for the two types of improvement strategies are NP-Hard even for one target object, and the problem becomes more complex when trying to improving multiple target objects. As such, our goal is to develop highly efficient heuristic algorithms.

3.2 Interpreting Objects as Functions

Our key idea is to interpret each object *as a function* and treat each top-k query *as a function input*. This is different from existing works where queries are considered as utility functions and objects as their input. We use the most common *linear* utility functions[7, 6, 11, 26] as an example to explain our idea.

For linear utility functions, each query $q_i \in Q$ is a d-dimensional vector $q_i = \{q_i^{(1)}, q_i^{(2)}, ..., q_i^{(d)}\}$ that assigns a weight to each attribute of an object and computes the weighted sum. For simplicity, we use the same assumption as existing works that all queries are *normalized*, i.e., $q_i^{(j)} \in [0, 1]$ for any dimension j. In our solution, we treat each object p_i as a linear function f_i, where $p_i^{(j)}$ is the j-th coefficient. It takes a query q as input and computes the ranking score of p_i:

$$f_i(q) = \sum_{j=1}^{d} q^{(j)} p_i^{(j)} \quad (1)$$

Note that the ranking score is the same as the weighted sum. The difference is that a query is now treated as a function parameter while the object attribute values are treated as function coefficients. As such, the set of objects D is interpreted as a set of functions $D = \{f_1, f_2, ..., f_n\}$. When causing no ambiguity, we will use p_i and f_i interchangeably to refer to the same object. To evaluate a top-k query q, we compute $f_1(q), f_2(q), ..., f_n(q)$ and select the k functions with lowest output values.

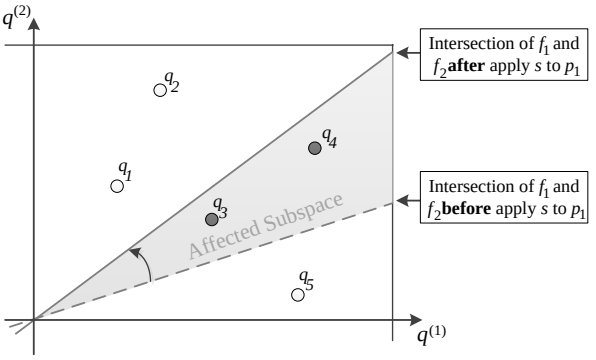
The intersection of two functions f_i and f_j creates a *hyperplane* in the d-dimensional domain space. The intersection partitions the domain into two *subdomains*, namely *above* and *below*. For any input q falls in the above subdomain, we have $f_i(q) \geq f_j(q)$, and for any input q in the below subdomain, we have $f_i(q) < f_j(q)$. The intersections of all functions partition the domain space D into a number of subdomains, and the functions can be strictly sorted in each of these subdomains. That is, if there exists a query point q in a subdomain such that $f_i(q) > f_j(q)$ (or $f_i(q) < f_j(q)$), then for any other query point p in the same subdomain, we have $f_i(p) > f_j(p)$ (or $f_i(p) < f_j(p)$). As a result, the rank of a function f_i remains the same for any two queries q_x and q_y as long as they fall in the same subdomain.

Applying an improvement strategy s to p_i will cause the intersections involving f_i to *tilt* towards some direction determined by s. The boundaries of some subdomains will also move. As showed in Figure 2, it may cause some query points to move to a different subdomain (e.g., move from above to below some intersections). We have two important conclusions.

Fact 1. *An improvement strategy* s *affects the result of a query* q *if and only if* q *is moved to a different subdomain after applying* s *to* p_i*. Thus, if no query point is moved to a different subdomain, we have* $H(p_i + s) = H(p_i)$*.*

Fact 2. *The rank of two functions* f_i *and* f_j *must be switched in the ranking result of some query* q*, if and only if* q *is moved from above (or below) to below (or above) of the intersection of* f_i *and* f_j*.*

The proofs of the two conclusions are straightforward. Due to limited space, we refer readers to [9, 16] for proof details. These facts suggest an efficient way to evaluate a given improvement strategy s. First, we apply s to p_i and



Query	Ranking results	
	Before applying s	After applying s
q_1, q_2	$[f_1, f_2]$	$[f_1, f_2]$
q_3, q_4	$[f_1, f_2]$	$[f_2, f_1]$
q_5	$[f_2, f_1]$	$[f_2, f_1]$

Figure 2: An improvement strategy affects subdomain boundaries and query results

find all the query points that are moved to a different subdomain. Then, for each query point found, check if p_i appears in its result and update $H(p_i + s)$ accordingly. The challenge now is, how to efficiently determine (without traverse all query points or subdomains) which query points are moved to which subdomains before and after applying an improvement strategy, and then compute their results. We discuss this approach in detail in the next section.

4. PROPOSED SOLUTION

We first introduce an *Efficient Strategy Evaluation* (ESE), which group query points by subdomains and index them using multidimensional data structures such as R-tree [10] or X-tree [3]. We will then discuss how to use ESE as a building block for efficiently processing of IQs. Here we consider only one target object with linear utility functions. Nevertheless, our techniques allow users to select multiple objects as targets, use different cost functions for each object, and query improvement strategies with non-linear utility functions, which we will discuss later in Section 5.

4.1 Efficient Strategy Evaluation (ESE)

Given an improvement strategy s for p_i, we need to compute its effectiveness in improving p_i, i.e., counting the number of top-k queries that include $p_i' = p_i + s$ in their result. For this purpose, existing solutions such as *Reverse top-k Threshold Algorithm* (RTA) [21] can be used. These schemes, however, support only linear utility functions. In particular, they are less efficient when a less number of queries include the object in their result. When $H(p_i + s)$ increases, their performance will drop significantly. Here we present an approach that works better for our purpose.

Given the intersection of two functions f_i and f_l:

$$\sum_{j=1}^{d} q^{(j)}(p_i^{(j)} - p_l^{(j)}) = 0 \quad (2)$$

Equation 3 represents the new intersection hyperplane after some improvement strategy s is applied to p_i.

$$\sum_{j=1}^{d} q^{(j)}(p_i^{(j)} + s_j - p_l^{(j)}) = 0 \quad (3)$$

The area bounded between the old and new intersection hyperplanes represented by Equation 2 and 3 formulates a subspace (e.g., the shadow area showed in Figure 2) inside the function domain space. We define this subspace as the *affected subspace* of s. It contains all the query points whose result are affected by applying s to p_i. To efficiently retrieve and evaluate such queries, we group all queries by their subdomains and index them with an R-tree.

Algorithm 1 *FindSubdomains*(I, Q)

```
1: d ← newSubdomain()
2: Subdomains.add(d)
3: for all q ∈ Q do
4:    q.subdomain ← d
5: end for
6: for all I_i ∈ I do
7:    for all Subdomain d ∈ Subdomains such that d overlaps I_i do
8:       d_above ← newSubdomain()
9:       d_above.boundaries.add(I_i, above)
10:      d_below ← newSubdomain()
11:      d_below.boundaries.add(I_i, below)
12:      for all q falls in d do
13:         if q falls above I_i then
14:            q.subdomain ← d_above
15:         else
16:            q.subdomain ← d_below
17:         end if
18:      end for
19:      if d_above contains query then
20:         Subdomains.add(d_above)
21:      end if
22:      if d_below contains query then
23:         Subdomains.add(d_below)
24:      end if
25:   end for
26: end for
27: Return Subdomains
```

Group query points by subdomain: Subdomains are partitioned using intersection hyperplanes of functions in D. Thus we need first to find the intersections created by the functions. This can be efficiently done using intersection discovery algorithms such as the plane sweeping algorithm [15]. We then partition the function domain into subdomains gradually, by considering function intersections one at a time.

Let $I = \{I_1, I_2, ..., I_m\}$ be the set of all function intersections. An intersection hyperplane I_i partitions the domain space into two subdomains: subdomain *above* and subdomain *below* the intersection. As such, it also partitions the query points Q into two groups, above and below. Note that queries fall on the intersection hyperplane can be treated as above it with no affect on the proposed algorithm. Whether a query point q falls above or below I_i is checked as follows. Let I_i be the intersection of some functions f_a and f_b. A query q falls above I_i if and only if $f_a(q) - f_b(q) \leq 0$. Otherwise q is below I_i. These two groups of queries can then be further partitioned by considering another inter-

section. We repeat this *binary space partitioning* process until no group can be further partitioned. At the end, for each query, we add an attribute *Subdomain* that contains a unique subdomain ID, recording the subdomain that contains the query point. If all query points in a sub-tree have the same *Subdomain* value, then we can mark this on the root-node of the sub-tree, instead of storing the same information for each query point. Note that we can also find which intersection serves as a boundary of a subdomain during this process. Finally, to save space, all the subdomains that contain no query point are simply discarded. A more formal description of this process is given in Algorithm 1.

Once the index is in place, computing $H(p_i+s)$ is straightforward. We only need to evaluate (or re-evaluate, if it is already evaluated) all queries falling in the affected subspaces. To check whether a query point q falls in the affected subspace, it is not necessary to solve the system of Equation 2 and 3. It is determined by two boundary conditions:

$$\sum_{j=1}^{d} q^{(j)}(p_i^{(j)} - p_l^{(j)}) \geq 0 \quad (4)$$

$$\sum_{j=1}^{d} q^{(j)}(p_i^{(j)} + s_j - p_l^{(j)}) < 0 \quad (5)$$

which is equivalent to a range query over the R-tree index, where the query range is the affected subspace (ruled by the boundaries of the function domain, if any). However, evaluating queries in the affected subspace may still be expensive if the affected subspace is large. Here we propose two methods to avoid complete re-evaluation of any query.

First, by Fact 2, if q falls in the affected subspace after s is applied, the new ranking result of q can be generated by simply switching the rank of f_i and f_l in the original ranking result. If q is not in the affected subspace, its result must remain the same. Additionally, if f_l was not in the top-k result of q, it indicates that after applying the improvement strategy, f_i cannot be in the top-k of the q because it only switches order with f_l. As such, we can rapidly eliminate unaffected queries.

Second, all query points fall in the same subdomain share exactly the same ranking result. Thus at most one query needs to be evaluated per subdomain. Recall that we have already grouped query points by their subdomains in the indexing step, and marked for each query which subdomain contains it. Let $TP(p_i) \subseteq Q$ denote the set of queries hit by p_i. The pseudocode of this ESE approach is given in Algorithm 2.

We first find all the affected subspace(s) for the given strategy s. This is done by checking all function intersections involving f_i among the intersections found in the indexing stage. For each query point that falls in an affected subspace of s, we check its query result. If the query has not been evaluated yet, then evaluate it and cache the result for future use (note that at most one query result needs to be cached per subdomain). Otherwise, use the aforementioned function-switching method to rapidly generate its result. For each subdomain, only one query needs to be evaluated, and the result can be shared for all other queries. In ESE, each top-k query needs to be evaluated for at most once, and the result of a large proportion of queries can be generated by re-using the result of their nearby queries, given that they fall in the same subdomain.

Algorithm 2 *EfficientStrategyEvaluation*(p_i, s)

```
1: H(p_i + s) ← |TP(p_i)|
2: for all f_l ∈ D intersects f_i and f_l ≠ f_i do
3:    Find the affected subspace
4:    for all q falls in the affected subspace do
5:       if q is not evaluated then
6:          evaluate q
7:       end if
8:       Switch the rank of f_i and f_l;
9:       for all q_j falls in the same subdomain as q do
10:         if q_j ∉ TP(p_i) and q_j ∈ TP(p_i + s) then
11:            H(p_i + v) + +;
12:         else if q_j ∈ TP(p_i) and q_j ∉ TP(p_i + s) then
13:            H(p_i + v) − −;
14:         end if
15:      end for
16:   end for
17: end for
18: Return H(p_i + s)
```

4.2 Improvement Strategy Searching

4.2.1 Min-Cost Improvement Strategy

Let p_i be the object to be improved. Given an improvement strategy s, we have the improved object $p_i' = p_i + s$. We use $p_{j,k}$ to denote the k-th ranked object of query q_j. In order for p_i' to be in the result of q_j, the following condition must hold:

$$f_i'(q_j) < f_{j,k}(q_j) \quad (6)$$

That is, the ranking score of p_i' must be less than that of $q_{j,k}$. Here $f_{j,k}$ is $p_{j,k}$'s corresponding function and f_i' that of p_i'. We have variable $x_j = 1$ if p_i' appears in the result of q_j and $x_j = 0$ otherwise. For the min-cost improvement strategy, the goal is to minimize the cost under the condition that p_i' can hit at least τ queries. This problem can be formulated as a constrained optimization problem:

$$\text{minimize} \quad Cost_{p_i}(s) \quad (7)$$

$$\text{subject to} \sum_{j=1}^{m} x_j \geq \tau \quad (8)$$

$$f_i'(q_j) < f_{j,k}(q_j) + (1 - x_j)C \quad \forall j \in [1, m] \quad (9)$$

$$x_j \in \{0, 1\} \quad \forall j \in [1, m] \quad (10)$$

where C denotes a very large number that exceeds the highest score of all objects. Constraint 8 guarantees that the improved object hits at least τ queries, while Constraint 9 ensures that Equation 6 is satisfied for each hit query. Note that the improvement strategy must also be *Valid*. That is, all attribute values of the improved object must not exceed the allowed range. For simplicity, here we assume p_i is defined on $\mathbb{R}^d$, thus the trivial condition $p_i + s \in \mathbb{R}^d$ is omitted in the above formulation. Nevertheless, in the case where this certain limitation on the value of the i-th attribute, additional constraints on s_i can be added to reflect such requirements for valid improvement strategies. For example, if the user does not allow value of the i-th attribute of the target object to be adjusted at all, we can simply add a constraint $s_i = 0$.

The formulated problem is an *integer linear programming* problem [23], which has been studied extensively and no efficient algorithm is known. The problem of searching for the min-cost improvement strategy actually is *NP-hard*. We prove it with a reduction from the Minimal Set Cover problem, which is known to be NP-hard.

DEFINITION 4 (MINIMAL SET COVER). *Given a set* $U = \{u_1, u_2, ..., u_n\}$ *and* $S = \{S_1, S_2, ..., S_m\}$ *where* $S_i \subseteq U$. *Find the minimal number of subsets in* S *whose union is* U.

Reduction from Minimal Set Cover to Min-cost Improvement Strategy: An instance of minimal set cover problem can be converted to an instance of the min-cost improvement strategy problem as follows: Create a top-1 query q_i for each element $u_i \in U$ with utility function:

$$u_i(p) = w_{i1} * p^{(1)} + w_{i2} * p^{(1)} + ... + w_{im} * p^{(m)} \quad (11)$$

and set weight w_{ij} to 1 if $u_i \in S_j$, and $w_{ij} = 0$ if otherwise. Suppose the objects are ranked by their utility scores in non-increasing order. Create two m-dimensional objects p_0 and p_1, such that all attributes of p_0 are set to 0 and all attributes of p_1 are set to $1/(m+1)$. Therefore $H(p_0) = 0$ and $H(p_1) = n$. The goal is to improve p_0 such that $H(p_0) = \tau = n$. We impose a simple linear cost function:

$$Cost_{p_0}(s) = s_1 + s_2 + ... + s_m \quad (12)$$

such that the cost of adjusting any attribute of p_1 is equally expensive. Additionally, each attribute of p_0 is discrete and can only be 0 or 1. Note that covering an element $u_i \in U$ is equivalent to hitting query q_i with p_0. In order to do so, an improvement strategy must adjust at least one attribute $p^{(j)}$ of p from 0 to 1 where $w_{ij} = 1$, which indicates that subset S_j should be selected to cover u_i. The total improvement cost is equal to the number of selected subsets. As such, a min-cost improvement strategy for the converted instance can be translated into a minimal set cover for the original instance.□

We now propose a heuristic algorithm (Algorithm 3) which leverages the proposed ESE algorithm to search for the sub-optimal strategy. The algorithm consists of multiple iterations. In each iteration, it first computes for each query $q_j \in Q$, a strategy s_j such that $p'_i = p_i + s_j$ can hit it with the minimal cost. This step generates a set of S of *candidate improvement strategies*. Then we apply to p_i the strategy $s \in S$ with the *minimal cost per hit query* $Cost_{p'_i}(s)/H(p'_i + s)$. Repeat this process until p'_i hits at least τ queries. In each iteration, we call the ESE algorithm as a subroutine to compute $H(p'_i + s_j)$.

Algorithm 3 $MinCostIQ(p_i, \tau, Cost_{p_i})$

```
1: p'_i ← p_i
2: while H(p'_i) < τ do
3:    S ← ∅
4:    for each query q_j ∈ Q and ∉ TP(p'_i) do
5:       s_j ← arg min Cost_{p'_i}(s) such that q_j ∈ TP(p'_i + s)
6:       Compute H(p'_i + s_j)
7:       S.add(s_j)
8:    end for
9:    Find s ∈ S with minimal Cost_{p'_i}(s)/H(p'_i + s)
10:   if H(p'_i + s) ≤ τ then
11:      p'_i = p'_i + s
12:   else
13:      Return s ∈ S with minimal Cost_{p'_i}(s) and H(p'_i +
         s) ≥ τ
14:   end if
15: end while
16: Return s = p'_i − p_i
```

Note that the algorithm requires to find the minimal cost strategy s_j that hits a query q_j. It formulates a single-constraint optimization problem:

$$\text{minimize } Cost_{p_i}(s) \quad (13)$$

$$\text{subject to } f'_i(q_j) < f_{j,k}(q_j) \quad (14)$$

which can be efficiently solved using standard math tools like [12].

The proposed algorithm can be considered a greedy one, since it always selects the improvement strategy with maximal efficiency-cost ratio at each step. The rational behind the algorithm is based on the following observation: The *average cost* per hit query is minimized in a min-cost improvement strategy, comparing with any other improvement strategies that hit the same number of queries. The proposed algorithm tries to minimize the average cost per hit query at each iteration. This greedy method reduces size of the searching space to $O(m)$ per iteration, and the number of iterations is bounded by τ. In comparison, exhaustive search takes at least $O(2^m)$ steps.

Similar to other greedy algorithms, our algorithm may terminate with a local optimum. Nevertheless, our experiment shows the algorithm is efficient enough to answer users' IQs interactively (i.e., a user hardly feels waiting time) with a regular desktop computer. Although the cost of the improvement strategy found may be sub-optimal, it greatly outperforms other methods such as simple greedy search (i.e., always try to hit the query with the least cost, repeat until hit enough queries) and random search (i.e., return a randomly generated improvement strategy), which we will discuss later. To sum up, this algorithm offers a good trade-off between improvement cost and feasibility.

Processing Min-Cost IQs: To issue a min-cost IQ, the query issuer first defines a cost function $Cost_{p_i}$ for the selected target p_i and specifies a desired τ. The system then uses Algorithm 3 to find the improvement strategy that satisfies the desired number of hits. For query issuers who indeed want the optimal strategy, we also provide them with the option of exhaustively strategy searching, which uses mathematical optimization tools (e.g., [12]) to solve the above optimization problem. However, due to the intractability of the problem, this algorithm is only feasible for very small datasets.

4.2.2 Max-Hit Improvement Strategy

Recall that the goal of maximal hit improvement strategy is to maximize the number of queries hit by the improved object with the constraint that the total cost does not exceed a given budget β. Similarly, we formulate the following optimization problem.

$$\text{maximize } H(p_i + s) \quad (15)$$

$$\text{subject to } Cost_{p_i}(s) \leq \beta \quad (16)$$

$$f'_i(q_j) < f_{j,k}(q_j) + (1 - x_j)C \quad \forall j \in [1, m] \quad (17)$$

$$x_j \in \{0, 1\} \quad \forall j \in [1, m] \quad (18)$$

The target function computes the hit number of the improved object, while Constraint 16 corresponds to the limited budget. The meaning of Constraint 17 is the same as the minimal cost improvement strategy problem. It is easy to see that searching for maximal hit improvement strategy is also NP-Hard, because the minimal cost improvement strategy problem reduce to it.

Reduction from Min-Cost Improvement Strategy to

Max-Hit Improvement Strategy: Let $MaxHit$ (p_i, β, $Cost_{p_i}$) be a subroutine that finds the maximal hit improvement strategy. We show how to find the minimal cost improvement strategy for p_i with desired hit τ by calling the subroutine. Let x_{max} be the cost required to hit all top-k queries, which can be treated as a constant. The minimal cost that we are looking for must fall in $[0, x_{max}]$, so we can search for the minimal cost strategy with a binary searching process. We start by setting β to an initial value x such that $x_{max} \geq x \geq 0$, and use the subroutine to find s such that $p_i + s$ hit the maximal number of queries. If $H(p_i + s) \geq \tau$, it means the minimal cost required to hit τ queries is no greater than x. Thus we refine the searching range by setting β to a new value in $[0, x]$ and repeat the process. Similarly, if $H(p_i + s) < \tau$, it indicates the minimal cost required must be larger than x and thus we set β to a new value in $[x, x_{max}]$. Regardless of the initial value, this binary searching process can find the minimal cost improvement strategy within $\log x_{max}$ attempts (i.e., by calling $MaxHit(p_i, \beta)$ for at most $\log x_{max}$ times, which is linear).□

The above proof demonstrates that the two improvement strategies, namely min-cost and max-hit, are closely related to each other. The two types of improvement strategies share a similar characteristic: the cost per hit query is minimized for a max-hit improvement strategy, comparing with any other improvement strategies with the same cost. As such, we modify the greedy searching Algorithm 3 to process max-hit IQs. The algorithm uses a similar searching method which looks for the most cost-efficient improvement strategy in each iteration, and the iterations terminate when all budget is used, or there is not enough budget to cover more queries.

Algorithm 4 $MaxHitIQ(p_i, \beta, Cost_{p_i})$

```
1: p'_i ← p_i
2: s* ← 0
3: while Cost_(p_i)(s*) < β do
4:    S ← ∅
5:    for each query q_j ∈ Q and ∉ TP(p'_i) do
6:       s_j ← arg min Cost_{p'_i}(s) such that q_j ∈ TP(p'_i + s)
7:       Compute H(p'_i + s_j); S.add(s_j)
8:    end for
9:    Find s ∈ S with minimal Costp'_i(s)/H(p'_i + s)
10:   if Cost_(p_i)(s*) + Cost_(p_i)(s) ≤ β then
11:      s* += s
12:   else
13:      for each s ∈ S, sorted by cost do
14:         if Cost_(p_i)(s*) + Cost_(p_i)(s) ≤ β then
15:            s* += s
16:         end if
17:      end for
18:      Break
19:   end if
20: end while
21: Return s*
```

Processing Max-Hit IQs: A max-hit IQ consists of target object(s), corresponding cost function(s), and a budget β. The improvement strategy that satisfies the budget constraint is then returned to the user by Algorithm 4. For convenience, we will refer to Algorithms 3 and 4 together as the **Efficient-IQ** algorithm. Similarly, we also provide the exhaustive search option in our implementation.

4.3 Data updating

Add/Remove a query: When a query point is added to or removed from Q, the R-tree needs to be updated. Adding or removing an indexed point on R-tree is easy. However, when a new query point is added, we need to find which subdomain contains it. We can use Algorithm 1 but only on the newly added query point to find its subdomain. This is usually not necessary. We observe that, if a new query point q falls closely to a group of other query points which are all in a subdomain d, then it is very likely that q also falls in d. Fortunately, we can quickly check if q falls in d by verifying the above/below relations between q and the boundary intersections of d as in Algorithm 1. Based on this observation, we propose to use the subdomain(s) of the k-Nearest Neighbour of q as candidate subdomain of q, and use Algorithm 1 only if q is not in any of these candidates.

Add/Remove an object: Adding or removing an object will cause the boundary of subdomains to change. Thus, similar to applying an improvement strategy, some query points may move to a different subdomain. We discuss how to update subdomain of affected queries as follows. When a new object is added, we first find all the newly created intersections and then rerun Algorithm 1 with these intersections to update the queries. Similarly, when an object is removed, we find all existing intersections that involve the object, and then locate all subdomains whose boundaries include one of the involved intersections. Then, if the subdomain is above the intersection, we merge it with the subdomain that is below it, and vice versa. This is to reflect the fact the once the object is removed, this intersection no longer exists, and the two subdomains that were separated by it should be merged as one subdomain. To facilitate this process, we implement a bloom filter to index the subdomains based on their boundaries, allowing us to quickly check if a subdomain uses an intersection as its boundary.

5. EXTENSION

5.1 Improving Multiple Target Objects

So far we have considered improving a single object. In this section, we extend our proposed techniques to enable users to query strategies that improve multiple objects. Here a user wants to select a set of objects $D_t \subseteq D$ as targets, and query the min-cost improvement strategy such that the total number of hits of the targets is no less than certain threshold τ, while the total improving cost is minimized. Each target can be associated to a different cost function, or share the same one. We assume that if one query is hit by two different target objects in D_t, the query is counted only once. We consider two **Combinatorial Object Improvement** problems.

DEFINITION 5. *Given a set of target objects $D_t \subseteq D$ and their corresponding cost functions, the **Combinatorial Min-Cost Improvement Strategy** for D_t is a set of improvement strategies S_t, where $s_i \in S_t$ is an improvement strategy for $p_i \in D_t$, such that $\sum_{p_i \in D_s} H(p_i + s_i) \geq \tau$ and $\sum_{p_i \in D_s} Cost_{p_i}(s_i)$ is minimized.*

DEFINITION 6. *Given a set of target objects $D_t \subseteq D$ and their corresponding cost functions, the **Combinatorial Max-***

Hit Improvement Strategy *for D_t is a set of improvement strategies S_t, where $s_i \in S_t$ is an improvement strategy for $p_i \in D_t$, such that $\sum_{p_i \in D_s} Cost_{p_i}(s_i) \leq \beta$ and $\sum_{p_i \in D_s} H(p_i + s_i)$ is maximized.*

The two problems are both NP-hard, since the single-object improvement strategy problems are their special cases. We can slightly modify the algorithms proposed in Section 4.2 to handle the combinatorial improvement strategy searching problems. To search for the combinatorial minimal cost improvement strategy, we can modify Algorithm 3 as follows: First finds the min-cost improvement strategies that can hit each query, and uses them as candidates. The algorithm then selects the candidate strategy with minimal cost per hit query. This process is repeated until at least the desired number of queries are hit. A more formal description is given as follows:

- Step 1: For each query q and each target object p_i, find the minimal-cost improvement strategy that makes p_i hits q. All such improvement strategies are used as candidates.
- Step 2: Find and apply the candidate strategy s with minimal cost per hit query. If the total number of hit queries after applying the strategy is larger than τ, then instead of s, we should apply the candidate strategy that hits at least τ queries with minimal cost. This is to avoid over-achieving the desired number of hits, and thus increase the total cost.
- Step 3: If the number of query hit by the improved objects is less than τ, repeat step 1 and 2.

Similarly, for max-hit IQ, we modify Algorithm 4 to make it applicable for multiple target objects.

- Step 1: For each query q and each target object p, find the minimal-cost improvement strategy that makes p_i hits q. All such improvement strategies are used as candidates.
- Step 2: Filter out the candidate strategies whose cost exceeds the remaining budget. If the candidate set is not empty, then select the candidate strategy with minimal cost per hit query, and apply it to the corresponding object. Update the remaining budget accordingly. If the candidate set is empty, then terminate.
- Step 3: If there is still available budget, repeat step 1 and 2.

5.2 Complex Utility Functions

We now discuss how to handle the case when the utility functions used in top-k queries are non-linear. Regardless of its complexity, a utility function $f(p_i)$ can always be seen as a function $f_{p_i}(q)$ for object p_i, in which the attribute values of p_i are treated as constants of the function, while the variable q consists of the other parameters of the top-k query (e.g., attribute weights as in linear utility functions). We explain the idea with a complex utility function example, applied on a Car dataset with three attributes (Table 1), where w_1 and w_2 are user-specified weights.

$$u(Car \quad c) = \sqrt{w_1 * c.Price} + w_2 \frac{c.Capacity}{c.MPG} \tag{19}$$

As showed in the table, each car object can be seen as a non-linear function, by treating its $Price$, MPG (Mileage Per Gallon gas), and $Capacity$ as constants. The function has input variables (w_1, w_2). The intersection of non-linear functions can take a more complex form. Generally, the intersection of two d-variable functions formulates a *surface* in the d-dimensional domain space. Nevertheless, our observation that these functions are sortable in subdomains partitioned by their intersection is still valid. Thus the proposed Efficient-IQ algorithm works as well over complex functions. Our concern is, however, for certain complex functions, the number of subdomains partitioned by intersections can be very large [1], which may result in a high indexing cost.

Table 1: *Car* dataset and the corresponding functions

ID	Price	MPG	Capacity	$u(w_1, w_2)$
1	15000	30	4	$\sqrt{15000w_1} + w_2\frac{4}{30}$
2	20000	28	6	$\sqrt{20000w_1} + w_2\frac{6}{28}$
3	8000	35	2	$\sqrt{8000w_1} + w_2\frac{2}{35}$

To mitigate this problem, we propose to *convert non-linear functions into linear functions* through variable substitution, i.e., replacing complex components of an equation with one variable to simplify the equation. After converting non-linear functions into linear ones, we can then apply the same techniques introduced in Section 4 for efficient processing of IQs. Consider an example of top-k queries with polynomial utility function, applied on a 4-dimensional dataset D:

$$u(p) = w_1(p^{(1)})^3 + w_2(p^{(2)} * p^{(3)}) + w_3(p^{(4)})^2 \tag{20}$$

which contains three high degree terms. It can be converted into an equivalent linear function:

$$u^*(p) = w_1 p^{(5)} + w_2 p^{(6)} + w_3 p^{(7)} \tag{21}$$

where $p^{(5)} = (p^{(1)})^3$, $p^{(6)} = p^{(2)} * p^{(3)}$, and $p^{(7)} = (p^{(4)})^2$ are used to substitute $p^{(1)}$-$p^{(4)}$. As such, each object becomes 7-dimensional. Nevertheless, in this example, attributes 1 4 are no longer used in the converted utility function, thus the dataset can be treated as 3-dimension. The value of each augmented attributed is computed using the original attribute values of the object, thus they do not need to be computed and stored in advance. Instead, we simple store the conversion process as math formulas, and compute their values on the fly to avoid storage redundancy.

Variable substitution can be used to convert other forms of complex functions into linear ones as well. Consider function:

$$u(p) = \sqrt{(w_1 - p^{(1)})^2 + (w_2 - p^{(2)})^2} \tag{22}$$

which computes the Euclidean distance between a data point and a given location $\{w_1, w_2\}$. We can make the following conversion:

$$u^*(p) = (w_1 - p^{(1)})^2 + (w_2 - p^{(2)})^2 \tag{23}$$

$$u^*(p) = (w_1^2 + w_2^2) - 2w_1 p^{(1)} - 2w_2 p^{(2)} \tag{24}$$

$$+ p^{(3)} + p^{(4)} \tag{25}$$

where $p^{(3)} = (p^{(1)})^2$ and $p^{(4)} = p^{(2)})^2$ are the two augmented attributes. Note that $u^*(p) = u(p)^2$. Since distance is always positive, the ranking result of the converted function

[1] For linear functions, the number of such subdomains is bounded by $O(n^d)$ where n is the number of objects and d the number of variables [17]. While for some high-degree functions, the number can be $O(2^n)$.

remains the same.

5.3 Heterogeneous Utility Functions

Since IQ allows users to apply complex utility functions, it is possible that each user defines a utility function with a completely different form. For example, to query the Car dataset (Table 1), some users may express their preference as a different utility function:

$$v(Car \quad c) = \frac{c.MPG}{w_1 * c.Price} + w_2(c.Capacity)^2 \tag{26}$$

In this case, we cannot simply use the value of (w_1, w_2) to differentiate different top-k queries. Because even for the same (w_1, w_2), the two functions 19 and 26 may compute different values, as they represent two evaluation methods over the same dataset. The default way to handle heterogeneous utility function is to add another column $v(w_1, w_2)$ to the Car dataset, and use function outputs in this column to sort the objects when considering the top-k queries with $v(Car \quad c)$. However, this will significantly increase the indexing cost, because we need to find subdomains for two different sets of functions, each has the same size of the object set.

To address this problem, we propose constructing a "generic" function in such a way that all the user-defined utility functions are special cases of this one function. Let's continue with the Car dataset example. Construct the following generic function for functions 19 and 26 by adding them up:

$$G(Car \quad c) = u(Car \quad c) + v(Car \quad c) \tag{27}$$

$$= \sqrt{w_1 * c.Price} + w_2 \frac{c.Capacity}{c.MPG} \tag{28}$$

$$+ \frac{c.MPG}{w_3 * c.Price} + w_4(c.Capacity)^2 \tag{29}$$

Now we can differentiate two queries by the value of (w_1, w_2, w_3, w_4) as in the linear case. Our solution works because if a query uses function 19 as utility function, it must set w_3, w_4 to 0. While for queries with function 26, w_1, w_2 is 0. As such, we unify the domain of the two functions into one domain space, and are able to interpret each object as only one function.

6. IMPLEMENTATION AND EVALUATION

6.1 System Implementation

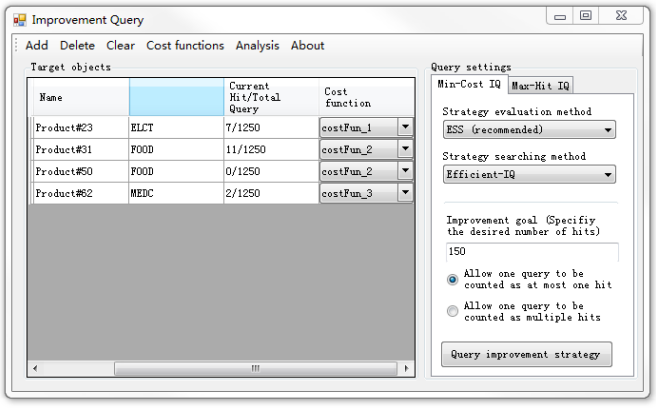


Figure 3: Graphic User Interface for Improvement Query

We have implemented the proposed techniques as an analytic tool and integrated it with the Database Management System (DBMS). The tool allows users to issue IQs in an interactive way via a Graphic User Interface (GUI) showed in Figure 3. Users can select target objects manually from the object dataset or via an SQL select statement. For the target objects, users specify which attributes can be adjusted and in what range, and also the cost function to be used for each object. Our system is implemented using C++ and C# on a Windows server with Intel Xeon 64-bit 8-core CPU running on 2.93GHz and 32GB RAM. An R-tree is used to index the queries. For comparison purpose, we implement four IQ processing schemes in our experiments.

- **Efficient-IQ:** This is the proposed heuristic algorithm, which uses the ESE algorithm for improvement strategy evaluation.
- **RTA-IQ:** This implementation uses the RTA algorithm, designed for reversed top-k query, to evaluate improvement strategies in each iteration, instead of the proposed ESE algorithm. Note that RTA supports only linear utility functions.
- **Greedy:** This implementation uses simple greedy algorithm. It always finds the query point that can be hit by any target object with the minimal cost, then repeats the process until the desired number of queries are hit (for Min-Cost IQs), or there is no budget left (for Max-Hit IQs).
- **Random:** This scheme randomly generates improvement strategies until it finds an improvement strategy that satisfies the improvement goal (i.e., hits the desired number of queries, or total cost less than the budget), and returns it as the answer to user's IQ.

6.2 Data Preparation

We test our system over four types of object datasets, namely Independent (IN), Correlated (CO), Anti-correlated (AC), and Real-world. IN, CO, and AC are synthetic datasets generated with the method described in [5]. Specifically, in IN, all attributes of an object are generated independently with a uniform distribution, while in CO and AC, attribute values of the an object is correlated or anti-correlated, respectively. Each generated object has 10 numerical attributes in range $[0, 1]$. We use two real-world datasets: VEHICLE and HOUSE. VEHICLE [1] contains 37051 vehicle models with attributes including year, weight, horse power, mileage per gallon (MPG), and annual cost. HOUSE is extracted from [18], including 100,000 records with four attributes house value, household income, number of person, and monthly mortgage payment. We normalize attributes of the real-world datasets to $[0, 1]$.

We generate two sets of top-k queries, namely UN and CL. Both sets of queries use polynomial utility functions, while the distribution of function coefficients (weights) are uniform and independent in UN but clustered in CL. Details of how to generate such queries are given in [21]. The degree of each term in the function is randomly chosen from $[1, 5]$ and the top-k value is randomly selected from $[1, 50]$. The default experiment setting is given in table 2.

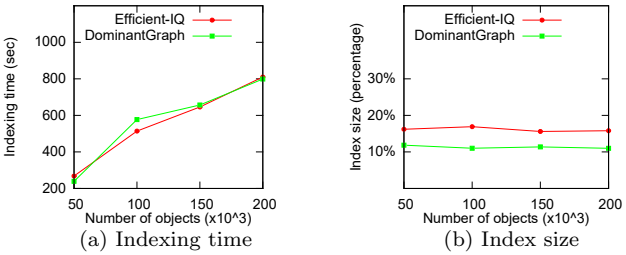
6.3 Experiment Results

Table 2: Experiment Setting

Parameter	Default	Range
$\|D\|$	100,000	50,000 - 200,000
$\|Q\|$	10,000	5,000 - 15,000
τ	250	100 - 500
β	50	10 - 100
Dimensionality	3	1 - 5

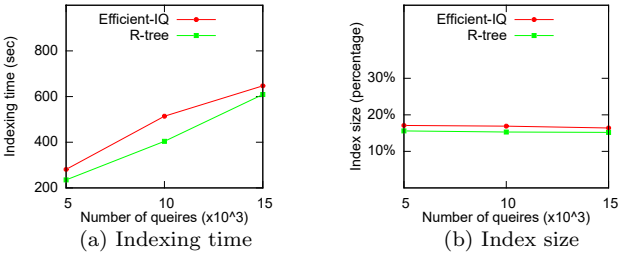
6.3.1 Data Indexing

We first evaluate the indexing cost of the proposed techniques, which involve the cost of building an R-tree over the query points and grouping them by subdomains. To better understand the scale of this cost, we compare indexing structure size (showed as percentage to the original dataset) and the total indexing time of the proposed technique (**Efficient-IQ**) with two benchmarks: 1) the cost of building only an R-tree on the query points (**R-tree**), and 2) the cost of building a Dominant Graph (**DominantGraph**) [26] for the objects, which is the state-of-the-art indexing technique for top-k query with linear utility functions.



(a) Indexing time (b) Index size

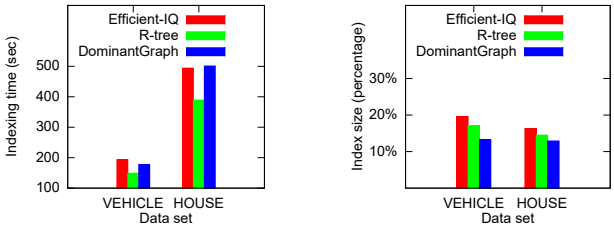
Figure 4: Scalability to the object set size



(a) Indexing time (b) Index size

Figure 5: Scalability to the query set size

We adjust the number of objects and report the corresponding indexing time and size of the proposed technique and DominantGraph (Figure 4). In order for Dominant Graph to work, we use only linear utility functions for top-k queries. For each test point, we generate 100 different utility functions and report the average indexing costs. We observe that the indexing cost on different types of synthetic data is almost the same, thus we report the average cost over all the types of datasets to save space. The dimension (i.e., number of variables) of the utility functions is uniformly picked in $[1, 5]$. The indexing time of DominantGraph is similar to our technique in general while Efficient-IQ incurs slightly higher storage overhead (less than 5% of the data size). However, our technique is unique in being able to support efficient processing of IQ.



(a) Indexing time (b) Index size

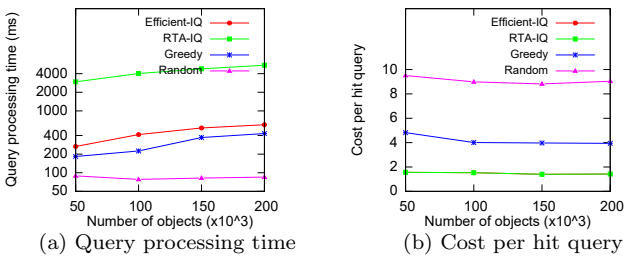
Figure 6: Indexing cost of real-world datasets

We then adjust the number of queries and compare the proposed technique with building only an R-tree (Figure 5). This time we allow non-linear utility functions. For the same set of queries, the proposed Efficient-IQ requires about 20% - 25% more indexing time comparing with building only an R-tree. The extra time is used to find subdomains for each query point, in order to facilitate the ESE algorithm. The final index size, nevertheless, is only about 10% larger than an R-tree. This is because many adjacent query points fall in the same subdomain and thus we do not need to store the subdomain information for each of them. In general, the propose technique shows good scalability, in terms of indexing cost, with respect to both the number of objects and queries. Experiment over real-world datasets is consistent with that on synthetic data.

6.3.2 IQ Processing

For query processing, we are interested in two metrics: 1) Average query processing time, and 2) Quality of the improvement strategy returned to the user. For Min-Cost IQ, the quality of an improvement query can be measured by its total cost. While for Max-Hit IQ, it's the total number of query hit by the improved objects. We use an unified quality measurement for both types of queries, i.e., the average cost per hit query of an improvement strategy, the lower the better. If multiple target objects hit the same query, we count them as only one hit. Our experiment shows that, even for the smallest dataset, exhaustive search takes more than 4 hours to process a query in average. Thus we compare only the 4 aforementioned schemes. For RTA-IQ to work, we limit the type of utility functions to linear with attribute weights normalized to 1. We use the following cost function for all objects:

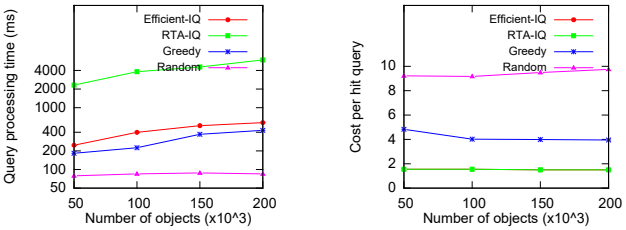
$$Cost(s) = \sqrt{\sum_{i=1}^{d} s_i^2} \tag{30}$$



(a) Query processing time (b) Cost per hit query

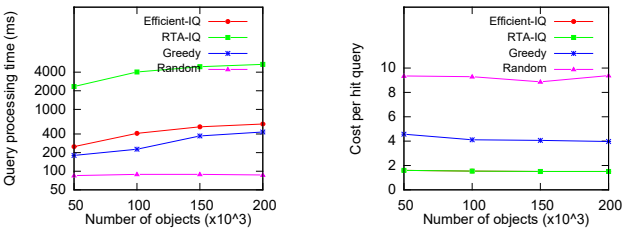
Figure 7: Query processing on the IN object dataset

We evaluate the scalability of the proposed techniques with regard to the size of D and Q respectively. The results on different data sets are showed in Figure 7-13. For



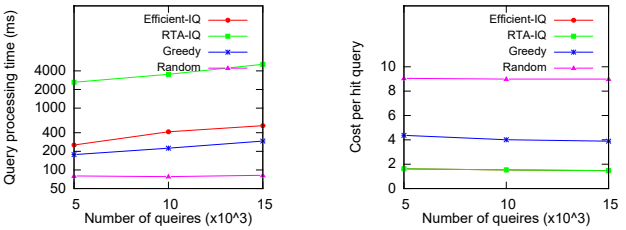
(a) Query processing time (b) Cost per hit query

Figure 8: Query processing on the CO object dataset



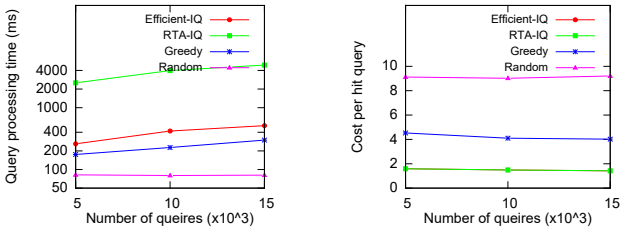
(a) Query processing time (b) Cost per hit query

Figure 9: Query processing on the AC object dataset



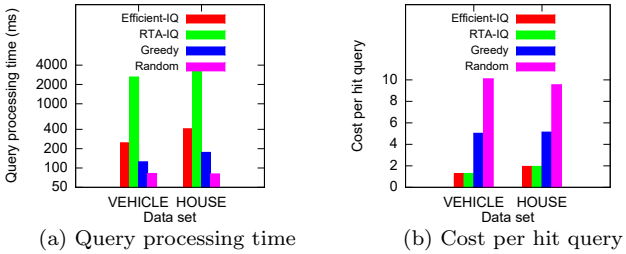
(a) Query processing time (b) Cost per hit query

Figure 10: Query processing on the UN query dataset



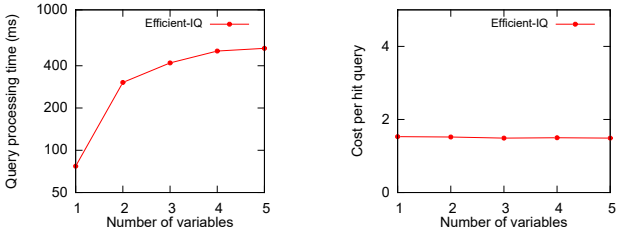
(a) Query processing time (b) Cost per hit query

Figure 11: Query processing on the CL query dataset



(a) Query processing time (b) Cost per hit query

Figure 12: Query processing on the real-world datasets



(a) Query processing time (b) Cost per hit query

Figure 13: Scalability to the number of variables in functions

each test point, we issue 100 Min-Cost IQs and 100 Max-Hit IQs, and report the average performance of the compared schemes. The parameters of these IQs are randomly and uniformly selected from the ranges given in Table 2. For each real-world dataset, we use a randomly generate query set that is one third of its size.

It is not surprising that Random is the fastest scheme in processing IQs, but it also yields the worst improvement strategy quality. The simple greedy algorithm has better strategy quality than Random, but is still very poor when compared with the proposed techniques. The Efficient-IQ achieves both good running time and high strategy quality. It outperforms RTA-IQ significantly in querying processing time, while achieving the best improvement strategy quality. (Note that RTA-IQ uses the same strategy-searching approach as Efficient-IQ, thus the quality of the strategies found by the two schemes is the same). The result shows that the good performance of the proposed technique is due to the combination of an efficient strategy searching method and a fast evaluation algorithm used in each searching iteration.

Finally, we evaluate the scalability of the proposed technique with respect to dimensionality of the functions (i.e., the number of variables in the interpreted functions). Since RTA only works on linear function, in this experiment we plot only the result of Efficient-IQ. The result (Figure) shows as the number of variables increases, the query processing time increases too, but in a sub-linear way. That means the query processing time becomes less sensitive to dimensionality as it increases, which is a desired feature.

7. CONCLUSION

We live in a society that is competitive in nature. Daily we face the challenges of improving something to make it more competitive against its peers. In this paper, we consider the problem of finding improvement strategies. We propose a new type of query called *Improvement Query* (IQ) that has two variants. A Min-Cost IQ retrieves the improvement strategy with minimal cost for some target object to hit a desired number of top-k queries, and a Max-Hit IQ tries to find an improvement strategy that maximize the number of hit queries with a given budget. Here the cost of an improvement strategy is modeled by a user-defined cost function. We show that finding the exact answers to both queries are NP-Hard and propose a suite of heuristic solutions. Our key idea is to interpret each object as a function and treat each top-k query as as its input. As such, the set of functions can be strictly sorted by their output in each subdomain partitioned by their intersections. The geomet-

rical relations among then function intersections can then be leveraged for efficient processing of IQs. We implement the proposed techniques as an analytic tool and integrated it with the DBMS. In our extensive evaluation, it demonstrates excellent performance.

8. REFERENCES

[1] Fueleconomy.gov vehicle data. http://www.fueleconomy.gov/feg/ws/index.shtml. Updated: Tuesday April 12 2016.

[2] J. Anderson. Determining manufacturing costs. *CEP*, pages 27–31, 2009.

[3] S. Berchtold, D. Keim, and H. Kriegel. An index structure for high-dimensional data. *Readings in multimedia computing and networking*, page 451, 2001.

[4] B. R. Binger et al. *Microeconomics with calculus*. Number 338.501 B613 1998. Addison-Wesley, 1998.

[5] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 421–430. IEEE, 2001.

[6] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: indexing for linear optimization queries. In *ACM SIGMOD Record*, volume 29, pages 391–402. ACM, 2000.

[7] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *VLDB*, volume 99, pages 397–410, 1999.

[8] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *Proceedings of the 32nd international conference on Very large data bases*, pages 451–462. VLDB Endowment, 2006.

[9] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 2000.

[10] A. Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[11] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. In *ACM SIGMOD Record*, volume 30, pages 259–270. ACM, 2001.

[12] L. G. Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, 1980.

[13] H. Lu and C. S. Jensen. Upgrading uncompetitive products economically. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 977–988. IEEE, 2012.

[14] K. Mouratidis, J. Zhang, and H. Pang. Maximum rank query. *Proceedings of the VLDB Endowment*, 8(12):1554–1565, 2015.

[15] J. Nievergelt and F. P. Preparata. Plane-sweep algorithms for intersecting geometric figures. *Communications of the ACM*, 25(10):739–747, 1982.

[16] F. P. Preparata and M. Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[17] L. Schläfli. *Theorie der vielfachen Kontinuität*, volume 38. Zürcher & Furrer, 1901.

[18] R. G. J. G. Steven Ruggles, Katie Genadek and M. Sobek. *Integrated public use microdata series: Version 6.0 [Machine-readable database]*. University of Minnesota, 2015.

[19] J. Viner. *Cost curves and supply curves*. Springer, 1932.

[20] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 365–376. IEEE, 2010.

[21] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Norvag. Monochromatic and bichromatic reverse top-k queries. *Knowledge and Data Engineering, IEEE Transactions on*, 23(8):1215–1229, 2011.

[22] Q. Wan, R. C.-W. Wong, I. F. Ilyas, M. T. Özsu, and Y. Peng. Creating competitive products. *Proceedings of the VLDB Endowment*, 2(1):898–909, 2009.

[23] L. A. Wolsey and G. L. Nemhauser. *Integer and combinatorial optimization*. John Wiley & Sons, 2014.

[24] Z. Zhang, S.-w. Hwang, K. C.-C. Chang, M. Wang, C. A. Lang, and Y.-c. Chang. Boolean+ ranking: querying a database by k-constrained optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 359–370. ACM, 2006.

[25] Z. Zhang, C. Jin, and Q. Kang. Reverse k-ranks query. *Proceedings of the VLDB Endowment*, 7(10):785–796, 2014.

[26] L. Zou and L. Chen. Dominant graph: An efficient indexing structure to answer top-k queries. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 536–545. IEEE, 2008.