2017

Execution and authentication of function queries

Guolei Yang
Iowa State University

Execution and authentication of function queries

by

Guolei Yang

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Iowa State University

Ames, Iowa

2017

DEDICATION

To all those who find inspiration in this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I have received tremendous help from many people, without whom I doubt I would have the courage and capability to complete my study at Iowa State University. They are my advisor, my collaborators, my instructors, and my teammates, but most importantly, they are all my friends.

I would like to express my greatest gratitude to my advisor Dr. Ying Cai, who I believe is one of the best guides a student could ever ask for. I am particularly touched by his passion for great ideas and deeply influenced by his meticulous attention to detail. Through the years we worked together, he showed me his way of inspiring students that I rarely see anywhere else. Dr.Cai showed me that he has the strongest faith in his students as well as their work even in the hardest times. It is his faith that made me who I am today.

I am truly honored to have a group of excellent researchers as my collaborators. They are Dr. Neil Zhenqiang Gong and Dr. Forrest Bao at Iowa State University, Dr. Andreas Züfle at George Mason University, Dr. Chandan K. Reddy at Virginia Tech, Yuqiu Qian at HongKong University, my schoolmates Minghong Fang and Jinyuan Jia, and my research teammates Shixin Tian, Zhenbi Hu, and Sen Wang. I appreciate them for sharing their invaluable ideas with me and lend me their support when I needed it the most.

I am greatly thankful to my Program of Study Committee, including Dr. Neil Zhenqiang Gong, Dr. Yong Guan, Dr. Wallapak Tavanapong, and Dr. Wensheng Zhang, for their support and insightful comments, which helped me in improving my dissertation.

Finally, I want to say that the unconditional support and love from my family is the most important reason that I am able to receive my education and achieve anything in my life. To them, I own the deepest gratitude.

ABSTRACT

We introduce a new query primitive called *Function Query* (FQ). An FQ operates on a set of math functions and retrieves the functions whose output with a given input satisfies a query condition (e.g., being among top-k, within a given range). While FQ finds its natural uses in querying a database of math functions, it can also be applied on a database of discrete values. We show that by interpreting the database as a set of user-defined functions, FQ can retrieve the information like existing analytic queries such as top-k query and scalar product query and even more. Our research addresses the challenges of FQ execution and authentication. The former is how to minimize the computation and storage costs in processing an FQ, whereas the latter, how to verify that the result of an FQ returned by a potentially untrustworthy server is indeed correct. Our solutions are inspired from the observations that 1) the intersections of a set of continuous functions partition their domain into a number of *subdomains*, and 2) in each of these subdomains, the functions can be sorted based on their output. We prove the correctness of the proposed techniques and evaluate their performance through analysis, prototyping, and experiments using both synthetic and real-world data. In all settings, our techniques exhibit excellent performance. In addition to FQ, our research has developed another query primitive called *Improvement Query*, which we also include in this dissertation.

CHAPTER 1. OVERVIEW

We introduce a new query primitive called *Function Query* (FQ). An FQ operates on a set of continuous functions with a same set of variables and retrieves the functions whose output under a given input satisfies a certain condition. Let $F = \{f_1(X), f_2(X), \cdots, f_n(X)\}$ be a set of n functions, where $X = \{x_1, x_2, \cdots, x_m\}$ is a set of input variables. We are interested in three kinds of FQs, which are different in output conditions:

- A *top-k FQ* returns the functions whose output with a given input is among the top k.
- A *range FQ* returns the functions whose output with a given input is within a certain range.
- A *kNN FQ* returns the k functions whose output with a given input is nearest to some value.

Math functions are often the most compact and intuitive way to represent continuous data such as the trajectory of a moving object and temperature observed over time. Indeed, in many scientific, financial, and sensor network applications, the data they store are naturally a set of functions (Sistla et al. (1997); Thiagarajan and Madden (2008)). Our FQ finds its natural use in these applications. Consider a database where each record is a function $temp_i(t)$ that outputs the temperature at time t at location i. FQs let users retrieve information such as the k hottest/coldest locations at noon time, or all locations where the temperature at noon time is in between 10 to 20 C^0, or the 3 hottest locations at 10am. Another way of using FQs, which we believe is inspiring, is to apply them on a discrete database for analysis-based data retrieval. Here each record in the database is originally a set of discrete values, but *interpreted* as a continuous function.

As an example, consider Table 1.1. Each record r_i in the table describes a rental property, but can be interpreted as a function $CashFlow_i(d, r, n) = Income - Expense - Price \cdot (1-d) \cdot \frac{r(1+r)^n}{(1+r)^n - 1}$, where *Price*, *Income* and *Expense* are the values of the record's corresponding attributes. The function computes the cash flow of investing the property, based on an investor's down payment

Table 1.1: Rental Properties for Sale

PID	$Address$	$Price$	$Income$	$Expense$	$CashFlow(d,r,n) = Income - Expense - Price \cdot (1-d) \cdot \frac{r(1+r)^n}{(1+r)^n-1}$
1	...	99000	8000	1450	$CashFlow_1(d,r,n) = 8000 - 1450 - 99000 \cdot (1-d) \cdot \frac{r(1+r)^n}{(1+r)^n-1}$
2	...	145000	12000	2000	$CashFlow_2(d,r,n) = 12000 - 2000 - 145000 \cdot (1-d) \cdot \frac{r(1+r)^n}{(1+r)^n-1}$
3	...	138000	11500	1200	$CashFlow_3(d,r,n) = 11500 - 1200 - 138000 \cdot (1-d) \cdot \frac{r(1+r)^n}{(1+r)^n-1}$
4	...	258000	29000	4450	$CashFlow_4(d,r,n) = 29000 - 4450 - 258000 \cdot (1-d) \cdot \frac{r(1+r)^n}{(1+r)^n-1}$

percentage d and mortgage over a term of n months at a monthly interest rate r. Various FQs can then be performed on these functions. For example, an investor can issue a range FQ to retrieve the properties whose cash flow is no less than \$1000 per month with input ($d = 0.15, r = 0.005, n = 180$).

We will call $CashFlow(d,r,n) = Income - Expense - Price \cdot (1-d) \cdot \frac{r(1+r)^n}{(1+r)^n-1}$ a *function definition*, which serves as a template to interpret each record in a database as a function. For application needs, a same database can be interpreted as different sets of functions. For example, each record r_i in Table 1.1 may also be seen as a function $ROI_i(d,r,n) = (Income - Expense - Price \cdot (1-d) \cdot \frac{r(1+r)^n}{(1+r)^n-1})\ /(Price \cdot d)$, which computes the Return on Investment (ROI) for the property.

To our knowledge, the problem of querying a collection of functions has not been studied in literature. Our work is different from the track of research (e.g., Thiagarajan and Madden (2008); Guo et al. (2013); Katsis et al. (2015); Anagnostopoulos and Triantafillou (2017)) which is aimed at finding a model (e.g., a function) to represent some continuous physical phenomena and then using the model to answer queries over the phenomena. Take *FunctionDB* (Thiagarajan and Madden (2008)) as an example. Given the temperatures observed at different times for a location, the proposed system constructs a function $temp(t)$ to model the temperature at time t. Users can then write a query like $SELECT\ AVG(temp)\ WHERE\ 9am < t < 10pm$ to compute the average temperature in a time period. Such queries are different from our FQs. In our case, we are given a set of functions and we want to allow users to perform queries over these functions. The output of a FQ is a subset of the functions, instead of a data point represented by a function or the aggregated result of the data represented by the function.

Nevertheless, when applied on a discrete database for analysis-based data retrieval, FQs are closely related to existing *analytic queries* such as *top-k query* (Chauduri and Gravano (1999); Chang et al. (2000); Hristidis et al. (2001); Das et al. (2006); Zou and Chen (2008)) and *scalar product query* (Khan et al. (2014)). Analytic queries allow users to apply a *scoring* function on a database to retrieve those records whose score satisfies certain condition. These queries are special cases of our FQ. For example, top-k query and scalar product query are equivalent to our top-k FQ and range FQ, respectively. Existing research investigates analytic queries separately. Different queries require different techniques and allow only primitive types of scoring functions (e.g., weight-based linear function) and output conditions (e.g., either among top k or no greater than a given threshold, but not both). FQ does not have these limitations. Users can interpret a database as a set of functions of virtually arbitrary types and query the database with a variety of output conditions (e.g., top k, range, and kNN), with a single unified solution.

In this dissertation, we propose making FQs a new query primitive. We are mainly interested in two research problems. The first one is **execution**. A straightforward way to processing an FQ is to compute each function based on the user-supplied function input and then check the result against the output condition. This approach is simple to implement, but it requires to compute all functions in the database. To materialize the potentials of FQ, it is crucial to minimize the computation and storage costs in query processing. The second problem is **authentication**. Having a cloud server to process FQs can be cost-effective to many data owners. However, the server, which is managed by a third-party, may or may not be trustworthy in query processing. This calls for a solution that allows users to verify that the query results they receive from the server are indeed correct. The work of this dissertation is to address these problems. Our key insight is, the intersections of a set of functions partition their input domain into a number of *subdomains*, and in each of these subdomains, the functions are sortable. In light of these facts, we develop a suite of novel solutions. For efficient processing of FQ, we propose a new data structure called *Intersection-tree* (I-tree). The I-tree indexes the subdomains partitioned by the intersections without computing their boundaries and can be used to sort the functions for a subdomain by simply traversing the tree from the root to the subdomain. For users to authenticate query results, we propose to sort the functions in each subdomain and digitally sign the functions to create a *signature mesh*. We prove the correctness of the proposed techniques and evaluate their performance through analysis, prototyping, and experiments using both synthetic and real-world data. In all settings, our techniques exhibit excellent performance. In addition to FQ, our research has developed another query primitive called *Improvement Query*, which we also include in this dissertation.

CHAPTER 2. REVIEW OF LITERATURE

To the best of our knowledge, the notion of FQ has not been investigated in the literature. In this chapter, we discuss some closely related works.

2.1 Analytic Queries

Analytic queries are common in having a utility function in data retrieval, but they are actually vastly different in nature. Existing research has studied them separately. Each query type has its own set of processing techniques, and each technique has a unique indexing structure. In particular, most techniques support only weight-based utility functions. We discuss them as follows.

Top-k Query: A *top-k* query (also called preference query in some literature) retrieves the k records with the highest rank under a user-defined utility function, where users assign weights (i.e., their preferences) on different attributes and aggregate the weighted sum. The techniques developed for efficient processing of top-k query can be classified into several categories. Sorted lists-based techniques (e.g., Fagin et al. (2003); Nepal and Ramakrishna (1999); Balke and Kießling (2000)) sort the objects in each attribute and find their ranks by scanning each list in parallel until the top-k results are found. View-based techniques (e.g., Hristidis et al. (2001); Das et al. (2006)) employs materialized views to retrieve top-k, where a view is generated on the fly for each query. Layer-based technique (e.g., Chang et al. (2000)) plots data points in high-dimensional space and then computes the convex hulls of the data points. To faciliate query processing, these convex hulls are organized and indexed by layers because objects on the outer layers are more likely to be in top-k compared with those on the inner layers. To process a top-k query, data points on the convex hulls are scanned one by one starting from the out-most layer unitl the top-k results are found. Dominant group technique (Zou and Chen (2008)). The technique in the last category explore the dominant relationship between objects. A object O_i is said to dominate another object O_j if there exists no set of weights that O_i can be ranked lower than O_j. In this case, there is no way for O_j to be included in a query result unless O_i is included first. As such, the objects can be organized into groups based on their dominant relationship to facilitate efficient query processing.

The above techniques require the utility function to be weight-based. The work by Zhang et al. (Zhang et al. (2006)) addressed this problem with a variant of top-k query called *k-constrained optimization query*. The proposed technique supports non-linear utility functions through state-space indexing. Specifically, the objects are indexed with a quad-tree, where each tree node corresponds to a subspace in the object value space. A node in the quad-tree stores the objects that fall in the corresponding subspace, along with the range of each attribute value of these objects. Given a utility function and a user input, it computes which nodes on the quad-tree are most likely to contain objects with the top-k highest score. This is done by constructing the *landscape* of the utility function with respect to each user input, which shows peaks and valleys of the function output. However, constructing the landscape is time consuming for complex and/or high-dimensional functions. Another variant of top-k query is *k-hit query*, which supports a probabilistic utility function. By specifying the distribution of the weights in the utility function, users can retrieve the k objects with the highest probability to be ranked among top-k. In order to efficiently answer such queries, the authors proposed the concept of convex cones, which generalizes convex hulls to the probabilistic space. By constructing convex cones of data points, the proposed technique can efficiently estimate the maximal probability that a data point will be in top-k.

Reverse Top-k Query: (Vlachou et al. (2010, 2011)) Given a set of objects and a set of top-k queries, a *bichromatic reverse top-k query* retrieves the top-k queries whose result contains an object of interest. A *monochromatic reverse top-k query*, on the other hand, finds all weight combinations that can rank an object of interest among top-k under a linear utility function. An indexing structure is proposed in (Vlachou et al. (2010)) to efficiently process bichromatic reverse top-k queries. Their technique partitions the weight space evenly into grids, such that every top-k query can be seen as a point located in some grid. Then, whether a top-k query will rank an object top-k can be determined by checking the boundary of the gird that contains the query. As such, it is possible to eliminate some top-k queries without having to process them. An interesting variant of reverse top-k query is *reverse k-ranks query* (Zhang et al. (2014b)). Given a set of top-k queries Q and an object of interest p, such a query finds a set of m top-k queries which give p the highest rank among all queries in Q. A tree-based pruning algorithm was proposed for reverse k-ranks query. The algorithm uses an R-tree to index the query points in the weight space. Then, the same dominating relationship between objects can be found between different nodes on the R-tree, i.e., a tree node that locates closely to a hyperplane that represents an object may dominate a nodes that are far away from it, meaning that queries in the closer node must rank the object higher. Using this dominating relationship, the pruning algorithm attempts to eliminate the queries that are less likely to given p a high rank.

Other Top-k Related Queries: The work by Zhang el al. (Zhang et al. (2014a)) supplies a top-k query with a *global immutable region* (GIR). The GIR for a top-k query includes all possible weight settings for which the query result remains unchanged. The technique proposed for GIR treats objects as an arrangement of hyperplanes in the weight space. It constructs a GIR's boundaries by finding the largest subspace in the weight space portioned by the hyperplanes in which the result of a given query remains unchanged. A *maximum rank query* (Mouratidis et al. (2015)) computes the highest possible rank an object of interest can achieve for any weight setting in a weight-based utility function. The basic idea to find the maximal rank of an object is to search the weight-space and follow the paths that can increase the ranking of the object. In order to construct such paths, the objects are also treated as an arrangement of hyperplanes which partition the weight spaces into several half-spaces. These half-spaces are then indexed using a Quad-tree to facilitate the path-finding process.

Scalar Product Query: In a scalar product query, proposed in (Khan et al. (2014)), the utility function is in the form of the scalar product between some database attributes and a set of parameters whose value is provided by a user. The query condition, unlike top-k queries, is that the utility function output of an object satisfies an open ended inequality (i.e., object whose scalar product is no greater than a given threshold b). The authors shown that this type of utility functions can also support a special kind of top-k nearest neighbour query (i.e., finding the top-k objects whose scalar product is no greater than b and also minimal). Each scalar product query can be seen as a hyperplane in the object value space. Based on this observation, a planner-index technique is proposed to accelerate scalar product query process. The basic idea is to pre-process a set of scalar product queries with a wide range of query parameters. The results of these queries are stored in advance. Then, when a new query comes, the proposed technique attempts to identify which part of the result of the new query must be the same with one of the stored queries. Then, this part of result can be reused and only the different part needs to be computed for the new query. Scalar product query extends range query over single attribute of relational database, but the proposed technique is limited to scalar product functions and cannot support top-k or k Nearest Neighbour (kNN) retrieval over the computation results.

It is worth mentioning that some DBMS (e.g., DB2 10.5 and PostgreSQL 9.4) have recently supported expression-based indexes over numeric attributes. Nevertheless, they do not allow queries with parameters whose values are unknown in advance, thus are less related to our work.

2.2 Query Authentication

There are three parties involved in data outsourcing: data owner, cloud server, and data user. The data owner gives a database to the cloud server, and users send their queries to the cloud server and receive query results. The server is administered by a third party that may or may not be trustworthy. As such, data users, who submit queries, want to verify the query results they receive from the server are indeed correct. A query result is said to be correct if it is *sound* (i.e., every data item received comes from the original database and satisfies the query condition) and *complete* (i.e., all data items in the original database that satisfy the query condition are received).

The work by Devanbu et al. (Devanbu et al. (2003)) was among the first to study the problem of query authentication. In this work, the outsourced data is a list of data items and the query is a range query $q(l, u)$, i.e., retrieving the data items whose values are in the range of l and u. In the proposed solution, the data owner first sorts the data items and computes the hash value for each data item using a one-way hash function. A Merkle Hash tree (MH-tree) (Merkle (1990)) is then built on top of these hash values, where each internal node is a hash result of the concatenation of two children nodes. Figure 2.1 shows 4 data items, $r_1 \leq r_2 \leq r_3 \leq r_4$, and a corresponding MH-tree, where $H(\cdot)$ denotes the hash function and "|" the concatenation of two nodes. The root node is signed with the data owner's private key and then made known to all users. When processing a query q, the server returns not only the query result, but also a *verification object* $VO(q)$, which contains the data items immediately beyond the left and right query boundaries and some other tree nodes for the user to reconstruct the root digest. For example, if the query result $R(q)$ is $\{r_3\}$, then $VO(q)$ includes r_2, r_4, N_1, and the signed root digest. With these data, the user can reconstruct the root digest and compare it with the root digest published by the data owner. If the two digests match, the user can be assured that r_2, r_3, and r_4 are not tempered and their order is continuous in the original list, i.e., the query result is sound and complete.

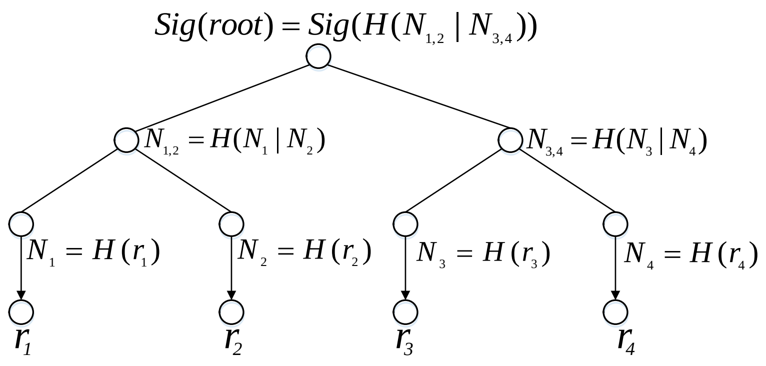

$Sig(root) = Sig(H(N_{1,2} \mid N_{3,4}))$
$N_{1,2} = H(N_1 \mid N_2)$
$N_{3,4} = H(N_3 \mid N_4)$
$N_1 = H(r_1)$
$N_2 = H(r_2)$
$N_3 = H(r_3)$
$N_4 = H(r_4)$
r_1
r_2
r_3
r_4

Figure 2.1: Merkle hash tree.

In a different work (Pang et al. (2005)), the authors proposed building a signature chain for authentication. The idea is to sort the data items and then create a signature for each record in the database. The signature for a record r_i is computed based on the digest of itself and the digests of its two immediate left and right neighbors, r_{i-1} and r_{i+1} , i.e., $Sig(r_i) = Sig(H(H(r_{i-1})|H(r_i)|H(r_{i+1})))$. Figure 2.2 shows a signature chain example. A query result $R(q)$

is accompanied with a $VO(q)$ that contains the signatures of all data in $R(q)$. This chain serves as the proof that for any two consecutive data r_i and r_{i+1} in $R(q)$, no data r_x exists in the original database such that $r_i < r_x < r_{i+1}$. The chain also allows the user to verify if all data in the query range is returned by checking the two boundary records.

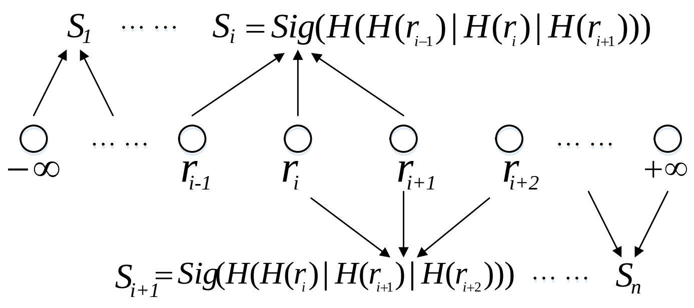

S_1
$S_i = Sig(H(H(r_{i-1}) | H(r_i) | H(r_{i+1})))$
$-\infty$
r_{i-1}
r_i
r_{i+1}
r_{i+2}
$+\infty$
$S_{i+1} = Sig(H(H(r_i) | H(r_{i+1}) | H(r_{i+2})))$
S_n

Figure 2.2: Signature chain.

The two techniques have their own advantages and disadvantages. With MH-tree, only the root node is signed, so the computation cost is low. However, $VO(q)$ needs to contain the entire search paths to all data in $R(q)$, so the size is larger, resulting in higher communication cost. Moreover, it is difficult to handle dynamic data. When a data is changed, the whole tree may need to be rebuilt. On the other hand, using signature chain incurs more computation cost to both the data owner and query issuers, but has a smaller $VO(q)$ and thus less communication overhead. It is more flexible to data update and in particular, allows a user to perform boundary checking without having to expose the user the two records that are immediate left and right to the sorted data in $R(q)$.

The two techniques have since inspired a series of research on query authentication. The work (Cheng et al. (2006)) considers multi-dimensional queries. The proposed technique maps the whole region into a spatial data structure (KD-tree or R-tree). The technique proposed in (Yang et al. (2008)) and (Yang et al. (2009)) combines R*-tree and MH-tree to deal with spatial data. Some other works (Li et al. (2006), Pang et al. (2009), Tang et al. (2013), and Tang et al. (2014)) consider the problem of data freshness, where the outsourced data may keep changing. A few more recent works (Hu et al. (2012)) and (Chen et al. (2013)) study the problem of privacy preservation in during the authentication process. All these techniques consider only simple queries over discrete data, where the raw data can be compared directly against the query condition. Complex queries such as top-k aggregation query and moving top-k spatial keyword query were considered in (Wu et al. (2015)) and (Choi et al. (2012)).

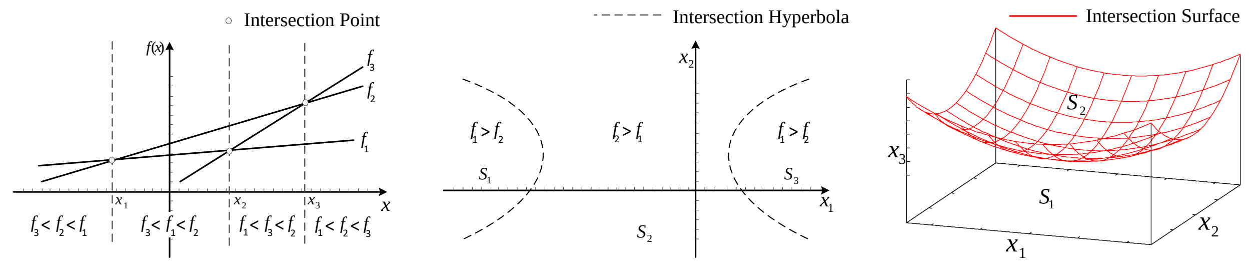
To our knowledge, authentication of function queries has not been studied in literature. In our case, the outsourced data is essentially a set of math functions. A query is associated not only a query condition, but also a function input. This input is specified by query issuers, which is not known to the data owner when uploading the data to the server.

CHAPTER 3. EFFICIENT EXECUTION OF FUNCTION QUERIES

3.1 Basic Idea

Consider a database, where each record r_i represents a function f_i. When causing no ambiguity, we will simply refer to the database as a set of functions $F = \{f_1, f_2, \cdots, f_n\}$. Given an FQ with an input value X and an output condition C (i.e., range, top-k, or kNN), one can compute every function in F with X as input and check if its output satisfies C. This simple solution requires to compute every function.

Our research has developed a more efficient solution. To facilitate our presentation, we first introduce some geometric terms. We say two functions f_i and f_j *intersect* on $X \in \mathbb{X}$ if $f_i(X) = f_j(X)$, where $\mathbb{X}$ is the function domain. We call the input X an *intersection point* of f_i and f_j. The set of all intersection points, referred to as the *intersection* of f_i and f_j, forms a hyperplane in the domain space. Given a set of functions, their intersections together partition $\mathbb{X}$ into a set of *subdomains*, each being a subspace in $\mathbb{X}$ bounded by the boundaries of $\mathbb{X}$ and/or intersection points.



(a) One dimensional intersections
$f_1=0.5x+3, f_2=1.2x+5, f_3=2x$

(b) Two dimensional intersections
$f_1=2x_1^2+3x_1 - x_2,\ f_2=x_1^2+4x_2^2-5$

(c) Three dimensional intersections
$f_1=4x_1-x_1^2 + x_2^2-3x_3+5,\ f_2=x_1^2+2x_1-x_2$

Figure 3.1: Examples showing functions are sortable in each of the subdomains partitioned by their intersections

We now present an important observation: *In each of the subdomains partitioned by the function intersections, the functions can be sorted based on their output.* Figure 3.1-(a) shows the intersections of three univariate linear functions f_1, f_2, and f_3. Here each intersection is a point on X-axis, and the three intersections x_1, x_2, and x_3 partition the domain into four subdomains, $(-\infty, x_1)$, $[x_1, x_2)$, $[x_2, x_3)$, and $[x_3, +\infty)$. In each of these subdomains, the functions are sortable, e.g., $f_1(x) \leq f_3(x) \leq f_2(x)\ \forall x$ in $[x_2, x_3)$. Complex functions may intersect in a more complicated way (as illustrated in Figure 3.1-(b) and (c)), but it remains true that they are sortable in each of the subdomains. Formally, we have the following theorem:

Theorem 1 *Let F be a set of functions defined on domain D. Let $S = \{S_1, S_2, ..., S_m\}$ be the set of subdomains partitioned by the intersections of the functions in F, where $\cup_{i=1}^{m} S_i = D$ and $\forall i \neq j, S_i \cap S_j = \emptyset$. For any subdomain $S_i \in S$ and $\forall f_i, f_j \in F$, if $\exists X_0 \in S_i$ such that $f_i(X_0) \leq f_j(X_0)$, we have $\forall X \in S_i$, $f_i(X) \leq f_j(X)$.*

Proof: : For contradiction, assume $\exists X_c \in S_i$ such that $f_i(X_c) > f_j(X_c)$. Let X_0X_c denote any path from X_0 to X_c. Due to continuity, there must be a point X_i on X_0X_c such that $f_i(X_i) = f_j(X_i)$. Hence X_i is an intersection point of f_i and f_j. Due to the property of intersections, the order of f_i and f_j must be different on the two sides of X_i on X_0X_c. Let $\epsilon > 0$ denote an arbitrarily small value. Let X_i^+ and X_i^- be two points on the two sides of X_i on X_0X_c, and the distance between X_i^+ and X_i^- is smaller than ϵ. Then $f_i(X_i^-) \leq f_j(X_i^-)$ indicates $f_i(X_i^+) > f_j(X_i^+)$, and vice versa. The total number of intersection points on X_0X_c must be an odd number, because otherwise the order of f_i and f_j should be the same at X_0 and X_c. Recall that any intersection point must locate on the boundary of some subdomain. Since moving from X_0 to X_c requires to cross subdomain boundaries for an odd number of times with any path, X_0 to X_c must locate in different subdomains, which contradicts the assumption that $X_c \in S_i$.$\square$

The theorem allows us to come up the following approach to support efficient FQ processing:

- Find all subdomains created by the intersections of the functions in F;

- Sort the functions by their output in each subdomain and store the sorted function lists.

To process an FQ with input X and output condition C, we can 1) locate the subdomain that contains X, 2) retrieve the function list sorted for the subdomain, and 3) perform a binary search on the list for the functions whose output satisfies C. Implementing this solution, however, remains challenging. First, the number of the subdomains can be very large. A set of n k-variable linear functions can have up to $m = n * (n - 1)$ intersections, which together can partition the domain space into $B_k^m = O(m^k)$ subdomains (Schläfli (1901)). Second, computing the boundary of each subdomain is computation-intensive when the number of functions is large. Moreover, finding the subdomain that contains a given input X is not trivial, since the subdomains are k-dimension polygons. Finally, sorting the functions and storing them for each subdomain will result in significant computation and storage overhead for large-scale datasets.

Our research addresses all these challenges. We first consider the case when the functions in F are linear. We propose a novel indexing structure, termed *Intersection-Tree* (I-tree), for efficient indexing of a large number of subdomains, without having to compute their boundaries. It supports not only efficient searching of the subdomain that contains a given input, but also efficient sorting of the functions for the subdomain. As one traverses from the tree root to a subdomain, the functions are sorted on the fly. Thus we can achieve a query processing time that is logarithmic of the number of functions. Built on top of this basic solution, we address the challenges of supporting FQs on more complex functions and enabling one to interpret one database as different sets of functions.

3.2 Proposed Solution

We first consider $F = \{f_1, f_2, \cdots, f_n\}$ being a set of *k-variable linear functions.* Let A_1, A_2, $\cdots$, and A_l be numerical attributes for a database. A function definition that interprets each record in the database as a k-variable linear function has the following general form:

$$F(x_1, x_2, \cdots, x_k) = C_1x_1 + C_2x_2 + \cdots + C_kx_k, \tag{3.1}$$

where coefficient C_i $(1 \leq i \leq k)$ can be any computation defined on the numerical attributes. For example, given a definition $F(x_1, x_2) = A_1x_1 + A_2/(A_3 + A_4)x_2$, we have $C_1 = A_1$ and $C_2 = A_2/(A_3 + A_4)$.

3.2.1 Structure of I-tree

The intersection of two k-variable linear functions is a k-dimensional hyperplane $\{X|f_i(X) - f_j(X) = 0\}$. This hyperplane partitions the input domain into two subdomains, namely *above* and *below*. The *above* consists of all inputs X such that $f_i(X) - f_j(X) \geq 0$, whereas *below*, all inputs X such that $f_i(X) - f_j(X) < 0$. As such, a set of n k-variable linear functions have up to $m = O(n^2)$ intersections and these intersections together partition the domain into $O(m^k)$ subdomains. The data structure of our I-tree, explained as follows, is designed to capture the essence of this *binary space partitioning* (Winder (1966)).

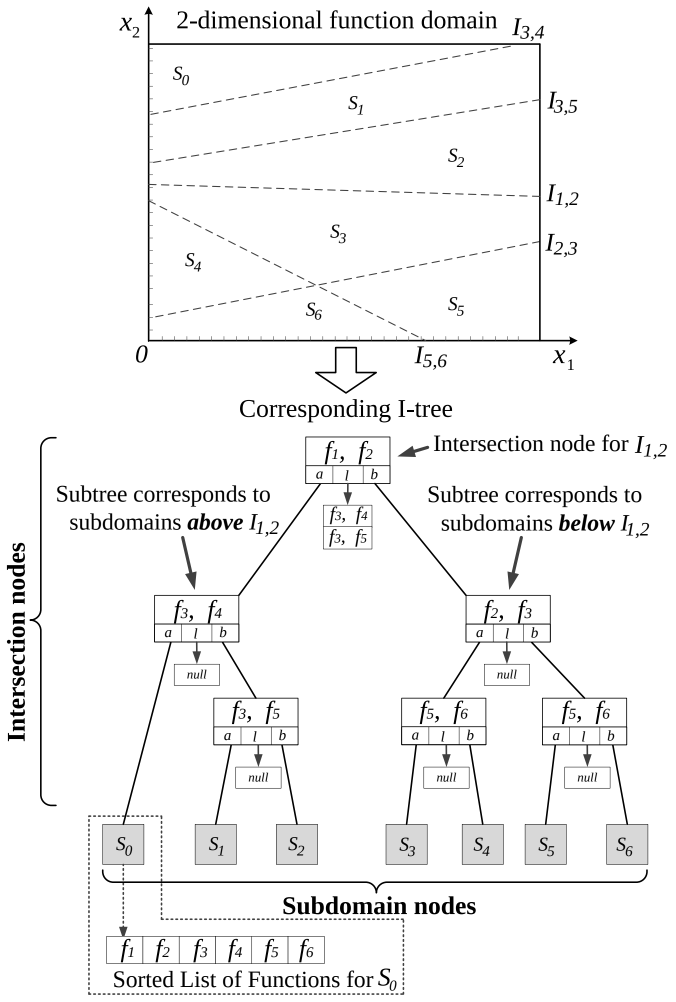

x_2
2-dimensional function domain
$I_{3,4}$
S_0
S_1
$I_{3,5}$
S_2
$I_{1,2}$
S_3
$I_{2,3}$
S_4
S_6
S_5
0
$I_{5,6}$
x_1
Corresponding I-tree
f_1, f_2
Intersection node for $I_{1,2}$
a l b
f_3, f_4
f_3, f_5
Subtree corresponds to subdomains **above** $I_{1,2}$
Subtree corresponds to subdomains **below** $I_{1,2}$
Intersection nodes
f_3, f_4
f_2, f_3
null
f_3, f_5
f_5, f_6
f_5, f_6
S_0 S_1 S_2 S_3 S_4 S_5 S_6
Subdomain nodes
f_1 f_2 f_3 f_4 f_5 f_6
Sorted List of Functions for S_0

Figure 3.2: An example of I-tree

An internal node in an I-tree has a form of (f_i, f_j, a, l, b) and is called an *intersection* node. It records the fact that two functions f_i and $f_j \in F$ intersect in some domain $\mathbb{X}$. We do not need to know $\mathbb{X}$, except for the root node, whose corresponding $\mathbb{X}$ is the entire domain specified by users. The intersection of f_i and f_j, denoted as $I_{i,j}$, partitions $\mathbb{X}$ into two subdomains *above* and *below*, which are represented by two pointers a and b, respectively. If *above* is further partitioned by another intersection $I_{p,q}$, then a links to the intersection node representing $I_{p,q}$. Otherwise, *above* is a subdomain where the functions can be strictly sorted according to their output. In this case, a links to a *subdomain node*. A subdomain node is a tuple of (S_i, L_i), where S_i refers to a subdomain and L_i the list of functions sorted for S_i. The same rule applies to subdomain *below*: If it is further partitioned, b links to another intersection node; Otherwise it links to a subdomain node. All subdomain nodes are leaf nodes. Finally, l in an internal node is a list that stores a set of function pairs. If a pair of functions f_p and f_q appears in l, it indicates that their intersection $I_{p,q}$ within $\mathbb{X}$ falls entirely above $I_{i,j}$. The list is used for efficient function sorting, which we will explain shortly.

Figure 3.2 shows an I-tree that indexes a domain of $[0, x_1^{max}] \times [0, x_2^{max}]$ partitioned by six 2-variable linear functions. The root note $N_{1,2}$ represents $I_{1,2}$, the intersection of f_1 and f_2. The two subdomains created by this intersection are represented by two subtrees linked by $N_{1,2}.a$ and $N_{1,2}.b$, respectively. The subdomain above $I_{1,2}$ is then partitioned by intersection $I_{3,4}$, while the subdomain below $I_{1,2}$ is partitioned by $I_{2,3}$. This information is recorded by the two nodes at layer 2 (i.e., $N_{3,4}$ linked by $N_{1,2}.a$ and $N_{2,3}$ linked by $N_{1,2}.b$). Among the four subdomains created by the three intersections, one of them is not further partitioned, so the corresponding pointer (i.e., $N_{3,4}.a$) links to a subdomain node (i.e., representing subdomain S_0). The remaining three are further partitioned by other intersections and therefore each of their corresponding pointers links to an internal node. These four nodes form layer 3. All nodes in layer 4 are subdomain nodes, each representing a subdomain where the functions are sortable. For simplicity, Figure 3.2 illustrates the subdomain nodes (except the leftmost one) with a subdomain ID only.

Note that each intersection node has an l pointer that links to a list of function pairs. A function pair (f_p, f_q) in the list linked by $N_{i,j}.l$ means their intersection hyperplane falls completely in the above subdomain created by the intersection of f_i and f_j. The list is null if no such function pair exists. If there are multiple function pairs in the list (e.g., for root node in Figure 2), they are sorted in top-down order: the pair of (f_u, f_v) is positioned higher than that of (f_p, f_q) if and only if hyperplane $f_u(X) - f_v(X) = 0$ falls entirely above $f_p(X) - f_q(X) = 0$. If two hyperplanes in the list intersect, their orders do not matter.

3.2.2 Search Operation

Searching on I-tree is straightforward. Let X be the function input in an FQ. To locate the subdomain node (S_i, L_i) where $X \in S_i$, we start by setting the current node N to be the tree root. If N is an intersection node (f_i, f_j, a, l, b), compute $f_i(X) - f_j(X)$. If $f_i(X) - f_j(X) \geq 0$, set N to $N.a$, because the subdomain containing X must fall above $I_{i,j}$. Otherwise, we set N to be $N.b$. This process is repeated until N becomes a subdomain node, which is then returned.

3.2.3 I-tree Construction

To build an I-tree, we first find each pair of functions that intersect. Two functions f_i and f_j intersect if $f_i(X) - f_j(X) = 0$ has a (real) root in the given domain. Without checking each pair of functions in F, we use PlaneSweep algorithm (Nievergelt and Preparata (1982)) to find function intersections. Given two functions f_i and f_j that intersect, we insert their intersection $I_{i,j}$ into the I-tree as follows. If the tree is empty, create a new intersection node (f_i, f_j, a, l, b) as root N_{root}, where $N_{root}.a$ and $N_{root}.b$ link to two new subdomain nodes and $N_{root}.l$ is initialized as an empty list. The corresponding sorted function list for each subdomain node will be generated later. If the tree is not empty, create a queue with the root node as its first element and then process each node N in the queue according to its type. If N is an intersection node (f_p, f_q, a, l, b), we check if there exists X such that $f_p(X) - f_q(X) \geq 0$ and $f_i(X) - f_j(X) = 0$. Checking this condition is equivalent to finding a real root of $f_i(X) - f_j(X) = 0$ in the closed haldspace define by $f_p(X) - f_q(X) \geq 0$, which can be done with standard root searching algorithms (Brent (2013)). If such an X exists, it indicates $I_{i,j}$ falls (partially or entirely) above hyperplane $f_u(X) - f_v(X) = 0$. In this case, we put $N.a$ in the queue. Likewise, we put $N.b$ in the queue if there exists an input X such that $f_p(X) - f_q(X) < 0$ AND $f_i(X) - f_j(X) = 0$. If N is a subdomain node, we replace N with a new intersection node $N' = (f_i, f_j, a, l, b)$, where $N'.a$ and $N'.b$ link to two new subdomain nodes, respectively, and $N'.l$ is an empty list. The above process is repeated until the queue is empty. A more formal description is given in Algorithm 1. As a binary search tree, an I-tree can be balanced using standard tree rotation algorithms (e.g., Stout and Warren (1986)) and we will not elaborate.

Algorithm 1 *Insert*$(N_{root}, I_{i,j})$

```
1: if $N_{root} == \varnothing$ then
2:     $N_{root} \leftarrow MakeNode(I_{i,j})$
3: else
4:     $Queue\ Q \leftarrow N_{root}$
5: end if
6: while $Q$ is not empty do
7:     $N \leftarrow Q.dequeue()$
8:     if $N$ is a subdomain node then
9:         $N \leftarrow MakeNode(I_{i,j})$
10:        return
11:    end if
12:    if $\exists X\ f_i(X) - f_j(X) \geq 0$ AND $f_u(X) - f_v(X) = 0$ then
13:        $Q \leftarrow N.a$
14:    end if
15:    if $\exists X\ f_i(X) - f_j(X) < 0$ AND $f_u(X) - f_v(X) = 0$ then
16:        $Q \leftarrow N.b$
17:    end if
18: end while
```

3.2.4 Function Sorting

Each subdomain node is a tuple of (S_i, L_i), where S_i represents a subdomain and L_i is the list of functions sorted for S_i. We now discuss how to generate L_i efficiently. We first sort the functions for subdomain S_0, the *topmost* subdomain that is above any other subdomain. This is done by selecting any input in S_0, computing each function with the input, and sorting the functions based on their output. We then use this sorted function list, referred to as the root list L_0, to construct the lists for other subdomains. Given a subdomain S_i, we initialize L_i as a copy of L_0 and adjust the order of functions in L_i while traversing from the root node to (S_i, L_i) as follows. Let $N = (f_i, f_j, a, l, b)$ denote the current node, which is initially the root node. If (S_i, L_i) is in N's left subtree, set N to be $N.a$. Otherwise, we switch the order of f_i and f_j in L_i, and for each pair of functions f_p and f_q in $N.l$, we switch their order in L_i one by one from the topmost ones. After switching, we set N to be $N.b$. This process is repeated until (S_i, L_i) is reached. The resulted L_i is the list of functions sorted for S_i. We give a more formal description of this sorting procedure in Algorithm 2 and an illustrative example in Figure 3.3. The correctness of this algorithm is proved as follows.

Algorithm 2 *SortSubdomain*(N, S_i, L_i)

```
1: $L_i \leftarrow L_0$
2: if $S_i == S_0$ then
3:     RETURN $L_i$
4: end if
5: if isInSubtree($N.a$,$S_i$) then
6:     $SortSubdomain(N.a, S_i, L_i)$, RETURN
7: else
8:     switchOrder($f_i$,$f_j$,$L_i$)
9:     for Each function pair $\{f_p, f_q\}$ in $N.l$ do
10:        switchOrder($f_p$,$f_q$,$L_i$)
11:    end for
12:    $SortSubdomain(N.b, S_i, L_i)$
13: end if
14: RETURN $L_i$
```

Theorem 2 *The above function switching algorithm generates L_i for S_i in the correct order.*

Proof: Let (f_i, f_j) be a pair of functions with intersection $I_{i,j}$. If S_0 and S_i are separated only by $I_{i,j}$, we can switch the order of f_i and f_j in L_0 to get L_i. The function switching process generalizes this idea. We show briefly that for any function pair (f_i, f_j), their order will switch if and only if $I_{i,j}$ falls between S_0 and S_i. The searching path from the root node to S_i partitions the I-tree into three disjoint parts: 1) *Path*, which contains all nodes on the searching path, 2) *Left*, which

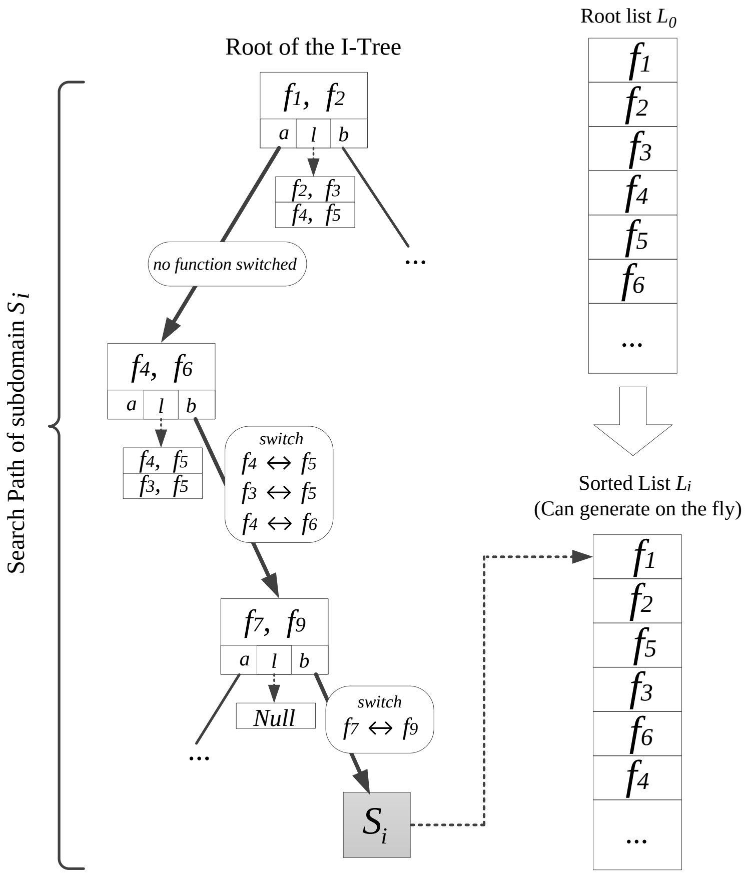

Root list L_0
Root of the I-Tree
f_1, f_2
a l b
f_2, f_3
f_4, f_5
no function switched
...
f_4, f_6
a l b
f_4, f_5
f_3, f_5
switch
$f_4 \leftrightarrow f_5$
$f_3 \leftrightarrow f_5$
$f_4 \leftrightarrow f_6$
f_7, f_9
a l b
Null
switch
$f_7 \leftrightarrow f_9$
...
S_i
Search Path of subdomain S_i
f_1 f_2 f_3 f_4 f_5 f_6 ...
Sorted List L_i
(Can generate on the fly)
f_1 f_2 f_5 f_3 f_6 f_4 ...

Figure 3.3: Sort F for S_i using the I-tree

contains all nodes in the left of the path, and 3) *Right*, which contains all nodes in the right side of the path. Let N be the intersection node representing $I_{i,j}$. We argue that N cannot be in *Right*. This is due to the fact that if the node appears in *Right*, then both S_0 and S_i must be above $I_{i,j}$, which is a contradiction. So we consider two cases:

1) N appears in *Path*. First, N must appear exactly once on the path due to the fact that intersection $I_{i,j}$ cannot appear in *above* or *below* subdomain partitioned by itself. Second, the path must visit $N.b$. Otherwise, S_i will be on the same side with S_0, i.e., both being above $I_{i,j}$, which is a contradiction. Thus the order of f_i and f_j must be switched when visiting node $N.b$.

2) N appears in *Left*, but not in *Path*. We can always find another intersection $I_{p,q}$ that falls between $I_{i,j}$ and S_i, such that a node M that represents $I_{p,q}$ is in *Path*; The traversing path must

visit $M.b$ for the same reason of case 1). We also know that f_i, f_j must appear in the list $M.l$, otherwise N should also be in $Path$ which is the same as case 1). As a result, f_i and f_j must switch their order in visiting $M.b$.$\square$

3.2.5 Database Updating

Adding a new function f_i may create new intersections with existing functions and further partition some subdomains. It will also affect existing sorted function lists. As such, we need to update the I-tree with new intersections and the root list L_0 (and invalidate other stored sorted lists, if any). The procedure for inserting a new intersection is given in Algorithm 1. To insert f_i into L_0, it computes $f_i(X_0)$, where X_0 is the input used to sort L_0 and then inserts f_i to L_0 via binary search.

To delete a function $f_i \in F$, we need to update the I-tree and the root list. Let $I_{i,j}$ denote f_i and f_j that intersect. We merge the subdomains separated by hyperplane $\{X|f_i(X) - f_j(X) = 0\}$. For every node N that represents intersection $I_{i,j}$, without loss of generality, assume subtree $N.a$ is larger than $N.b$. We first remove subtree $N.b$ and replace N by $N.a$. Then for each subdomain node M in $N.b$, we check if there exists a subdomain node in $N.a$ that has the same traversing path as M (which means that they are separated only by $I_{i,j}$). If such a subdomain node exists, we remove M. Otherwise, we insert M (and all its parents, if missing in $N.a$) into the same position of $N.a$ since these positions are not occupied. The pseudocode of this process is given in Algorithm 3.

3.2.6 Query Processing

There are two steps in processing an FQ. Let X be its function input. The first step, which is the same for all types of FQ, is to search the I-tree for the subdomain node (S_i, L_i) where $X \in S_i$. We have explained this step in Chapter 3.2.2. The second step is to search L_i to find the functions that satisfy the query condition under input X. For a top-k FQ, we simply return the first k functions in L_i. For range and kNN FQs, we retrieve such functions by performing a binary search on L_i. These algorithms are classic and we will not elaborate.

Algorithm 3 *Delete*(N)

```
1:  $N_{tmp} \leftarrow N.b$
2:  $N \leftarrow N.a$
3:  for each leaf $N_l$ in $N.tmp$ do
4:    new $Queue\ Path$
5:    $Path \leftarrow$ all nodes on path $N_{tmp} \rightarrow N_l$
6:    $N_{cur} \leftarrow N$
7:    for each node $M$ in $Path$ do
8:      if $M.Next == M.a$ then
9:        if $N_{cur}.a == \varnothing$ then
10:         $N_{cur}.a \leftarrow M.a$
11:       else if $N_{cur}.b == \varnothing$ then
12:         $N_{cur}.b \leftarrow M.b$
13:       end if
14:     end if
15:   end for
16: end for
```

3.2.7 Performance Improvement Strategies

The performance of I-tree can be improved significantly with some advanced strategies.

Sorting functions on the fly: Instead of proactively building L_i for each subdomain node (S_i, L_i), we can leave L_i empty during the tree construction but generate it *on the fly* when processing an FQ with an input $X \in S_i$. The process of generating L_i can be performed in parallel when searching for S_i. This approach significantly reduces the space needed for the I-tree, especially when only a few subdomains are visited frequently.

Sorting functions virtually: A constructed L_i can be used not only for FQs with input $X \in S_i$, but also for those whose input X falls in nearby subdomains. When a subdomain S_j is close to S_i, most functions in L_j have the same order as they appear in L_i. In the case S_j and S_i are adjacent, only a few, usually two, functions will switch their orders (see Figure 3.1 (a)). As such, instead of storing L_j, we can simply store a pointer to L_i and a *function switch table* that records the pairs of functions whose order need to be switched for L_j, thus creating a *virtually* sorted function list for L_j. The solution is attractive when S_j is not frequently visited (e.g., there are but only a few FQs with input $X \in S_j$).

Reducing I-tree size: To reduce the tree size, we can explicitly limit the number of subdomain nodes (leaves) by merging adjacent subdomains. Let S_i and S_j be two adjacent subdomains separated by intersection $I_{p,q}$. Instead of using two subdomain nodes, we can index S_i and S_j with only one node $N_{i,j}$. In the list $N_{i,j}.l$, we sort all functions except for f_p and f_q. The order of f_p and f_q is determined dynamically when $N_{i,j}$ is searched during query processing. Merging nodes on the lowest level can reduce the tree height by one and therefore reduce nearly half of the tree size. We can repeat to merge subdomain nodes level by level until the storage overhead is reached at the desired threshold.

3.3 Extensions

The techniques presented in Chapter 3.2 can be applied directly for more complex functions. This is due to the fact that, regardless of their complexity, the functions in F are sortable in each of the subdomains partitioned by their intersections. Our concern is, for some complex functions, the number of subdomains can be very large and become costly to handle. Consider the case of high-degree polynomials. The intersection of two k-variable d-degree polynomials defined on $\mathbb{R}^k$ formulates a k-dimensional surface with order up to d. A set of n functions have $O(n^2)$ intersections. In the worst case, these intersections partitions the domain space into 2^{n^2} subdomains (a subdomain may not be continuous).

To address the above problem, we propose to convert complex functions into linear functions and then apply the proposed indexing technique. Recall the number of subdomains partitioned by n k-variable linear functions is bounded by $O(n^{2k})$, which is substantially smaller than $O(2^{dn^2})$ and can be handled efficiently using I-tree. For large-scale datasets, where n is much larger than k, the improvement can be especially significant. We discuss how function conversion can be used to support FQs over 1) a single set of complex functions, and 2) multiple sets of functions which are interpreted from a same database through different function definitions.

3.3.1 Handling Complex Functions

To start with, consider a polynomial F in Equation 3.2. For each term $C_i x_i^{d_i}$, we can replace $x_i^{d_i}$ with an intermediate variable y_i and rewrite F as Equation 3.3.

$$F(x_1, x_2, \cdots, x_k) = C_1 x_1^{d_1} + C_2 x_2^{d_2} \cdots + C_k x_k^{d_k} \tag{3.2}$$

$$F^*(y_1, y_2, \cdots, y_k) = C_1 y_1 + C_2 y_2 \cdots + C_k y_k \tag{3.3}$$

This variable substitution allows us to convert a complex function into a linear once, and index only the linear functions using I-tree, which has been discussed in the previous chapter. Note that even if we only index functions in the form of Equation 3.3, a user can still query the complex functions in the original form (i.e., Equation 3.2). This is done by introducing an simple intermediate step in the query processing phase.

When processing an FQ, we evaluate each intermediate variable y_i using the original inputs, and then query the converted functions. For example, a condition $L \leq F(x_1, x_2, \cdots, x_k) \leq U$ can be rewritten as $L \leq F^*(y_1, y_2, \cdots, y_k) \leq U$, where $y_i = x_i^{d_i}$. This strategy introduces some extra costs, including: 1) function conversion, which is one time for each function definition, and 2) computing intermediate variables, which is one time for each query. There is also an extra cost resulted from the increased number of variables. Nevertheless, these extra costs are negligible comparing with the time and space saved from the reduction of the number of subdomains. The implementation of function conversion is made transparent to users.

Function conversion works for all functions that can be written in the general form:

$$F(X) = C_1 H_1(X) + C_2 H_2(X) + \cdots + C_l H_l(X) \tag{3.4}$$

where $X = \{x_1, x_2, \cdots, x_k\}$ is the set of variables, C_i is the i-th coefficient, and $H_i(X)$ is any computation on X as long as it does not involve any database attribute. We can replace $H_i(X)$ in each term with an intermediate variable y_i as showed in the above example. Note that we do not require the intermediate variables to be independent, i.e., $H_i(X)$ and $H_j(X)$ in $F(X)$ may relay on the same set of input variables to compute. This has no impact on the function conversion process since we only need to be able to compute the value of each intermediate variable y_i given X.

The above general form covers all polynomials, which is trivial to see, but also a rich family of non-polynomial functions. For example,

$$F(x_1, x_2) = A_1\sqrt{x_1^2 + x_2^2} + (A_2 + x_1x_2)^2, \tag{3.5}$$

which is non-polynomial, can be converted as

$$F^*(y_1, y_2, y_3) = A_1y_1 + A_2^2 + A_2y_2 + y_3 \tag{3.6}$$

where $y_1 = \sqrt{x_1^2 + x_2^2}$, $y_2 = 2x_1x_2$, and $y_3 = (x_1x_2)^2$. Note that all the intermediate variables depend on both x_1 and x_2, but since none of them contains database attributes, they can be handled by the function conversion technique.

When a function cannot be converted to the general form, the idea of function conversion may still work. Consider a point-of-interest (POI) table of (P, A_1, A_2), where P is a POI's ID and A_1 and A_2 are the POI's x and y coordinates, respectively. Consider the following functions defined on the dataset:

$$F(x, y) = \sqrt{(x - A_1)^2 + (y - A_2)^2} \tag{3.7}$$

by which users can issue a range FQ $F(x_0, y_0) \leq y$ to retrieve all POIs whose distance from a certain coordinate (x_0, y_0) is no greater than y miles. This function cannot be converted to the general form directly, but we can replace the function as $F^*(x, y) = (x - A_1)^2 + (y - A_2)^2$ and the query condition as $F^*(x_0, y_0) \leq d^2$. Since it can be converted into the general form, our techniques can then be applied. Nevertheless, finding the general conversion rule for arbitrary non-polynomial functions can be challenging and we leave it as a future work.

3.3.2 Handling Multiple Function Definitions

As discussed early, the same database can be interpreted as different sets of functions for application needs, through different function definitions. Here, a function definition servers as a "template" that tells the query process engine how to interpret each record as a function. For example, $\mathit{CashFlow}(d, r, n)$ and $\mathit{ROI}(d, r, n)$ are two functions defined on the same database showed in Table 1.1. A simple way to support multiple function definitions is to treat them independently by building one I-tree for each definition. This approach is inefficient, but can be addressed through function conversion.

Let $F_1(X)$ and $F_2(Y)$ be two function definitions on D. We define a new function $F_{12}(X, Y) = F_1(X)+F_2(Y)$ as the *Super Function* of $F_1(X)$ and $F_2(Y)$. $F_1(X)$ or $F_2(Y)$ can be seen as a special case of $F_{12}(X, Y)$ by setting the variables to a special value (e.g., 0). For example, a range query $L \leq F_1(X) \leq U$ can be converted to $L - C \leq F_{12}(X, 0) \leq U - C$, where $C = F_2(0)$. As such, we need to deal with only the super function. The super function, however, has $O(|X|+|Y|)$ variables. Fortunately, this problem can be alleviated by merging the terms with the same coefficient and then converting the super function into a simpler function via variable replacement. Consider two functions defined on a dataset with three numerical attributes (A_1, A_2, A_3), $F_1(x_1, x_2) = A_1x_1^{d_1} + A_2x_2$ and $F_2(x_1, x_2) = A_1x_1^{d_2} + A_3x_2$. We first generate a super function (some variables are renamed to avoid ambiguity):

$$F_{12}(x_1, x_2, x_3, x_4) \tag{3.8}$$

$$= F_1(x_1, x_2) + F_2(x_3, x_4) \tag{3.9}$$

$$= A_1x_1^{d_1} + A_2x_2 + A_1x_3^{d_2} + A_3x_4 \tag{3.10}$$

$$= A_1(x_1^{d_1} + x_2^{d_2}) + A_2x_3 + A_3x_4 \tag{3.11}$$

Since term $A_1x_1^{d_1}$ in F_1 and term $A_1x_1^{d_2}$ in F_2 share the same coefficient A_1, we merge them into one term in the super function and then convert the function into its linear version by variable replacement:

$$F_{12}^*(y_1, y_2, y_3) = A_1y_1 + A_2y_2 + A_3y_3, \tag{3.12}$$

where $y_1 = x_1^{d_1} + x_2^{d_2}$, $y_2 = x_3$, and $y_3 = x_4$. The extra step of computing y_1, y_2, and y_3 has been discussed previously (e.g., a query $L \leq F_1(a, b) \leq U$ is converted into $L \leq F_{12}^*(c, d, 0) \leq U$, where $c = a^{d_1}$ and $d = b$) and this process can also be made transparent to users.

3.4 Performance Analysis

3.4.1 I-tree Construction Cost

Let F be a set of n linear functions, each having k variables. Our technique first finds all function pairs that intersect. This can be done in $O((n+m)\log n)$ with the Plane-Sweep algorithm (Nievergelt and Preparata (1982)), where m is the number of function pairs that intersect. When the input domain is $\mathbb{R}^k$, m can be up to $n(n-1)$. But in reality, the allowed domain is usually limited and therefore m can be much smaller. For example, the input domain of top-k query is usually normalized to $[0,1]^k$ (Chang et al. (2000); Hristidis et al. (2001); Tao et al. (2007); Zou and Chen (2008)). To construct an I-tree for m intersections, we insert the intersections into the tree one by one. In the worst case, the i-th intersection will create $O(B_i^k)$ new subdomains (Schläfli (1901)). So the total cost is $\sum_{i=1}^{m} B_i^k$, which is bounded to $O(m^k)$. The time complexity of sorting the root list L_0 is $O(n\log n)$.

The storage overhead is determined by the size of I-tree and L_0, assuming other sorted lists are constructed on the fly. It may appear that this size can be $O(2^m)$ (since it could have up to m levels, but as discussed before, the total number of subdomains is bounded by $O(m^k)$ even in the worst case (i.e., each newly inserted intersection cuts as many existing subdomains as possible). Therefore, the storage overhead of the I-tree is also $O(m^k)$. The cost for storing L_0 is $O(n)$.

3.4.2 Function Conversion Cost

High-degree polynomials and non-polynomials are converted into linear functions and then indexed using I-tree. The conversion can be done by scanning each term of the function definition and replacing each complex term with an intermediate variable. The number of terms in a function definition is bounded to $O(kd)$, where k is the number of variables and d the maximal degree. In the worst case, each term needs an intermediate variable. Thus, the time cost of conversion is $O(kd)$. As for storage, we only record the replaced variables in order to make the same conversion in query processing phase. This will, in the worst case, create $O(kd)$ extra information for each function definition on D. The above cost is negligible as it is a one-time cost per function definition regardless of the size of the database. In terms of running time, this cost is responsible for less than 1% of the total indexing time in our experiments.

3.4.3 Query Processing Cost

Since tree balancing algorithm is discussed in Chapter 3.2.3, here we assume the I-tree is balanced. There are three steps in processing an FQ. The first step is computing the intermediate variables. This is needed when querying complex functions. The cost incurred in this step is also $O(kd)$. The second step is searching the I-tree for the subdomain node (S_i, L_i) where $X \in S_i$. Let $m = O(n^2)$ be the number of intersections, where n is the number of functions. The expected time complexity is $O(k \log m^k) = O(k^2 \log n)$. Here $O(\log m^k)$ is the expected tree height. $O(k)$ is the time of determining which subtree to visit, i.e., computing a k-variable linear function. Since k is usually much smaller than n (the number of functions), I-tree is much more efficient in query processing when compared to the linear searching, the cost of which is $O(kn)$, even in the rare worst case when $m = (n-1) \cdot n$. Again, m in real-world applications is usually much smaller because of domain space limitation. The last step in query processing is retrieving the query result from L_i. Since L_i is sorted, the time complexity is $O(\log n + r)$, where r is the number of the functions that satisfy the query condition.

3.4.4 Database Updating Cost

Insertion and deletion require to update the I-tree and the root list L_0. The latter is trivial and takes $O(\log n)$ (e.g., on a binary search tree), while the former needs to traverse the tree to update the affected subdomain nodes. A subdomain node is affected if it needs to be further partitioned or merged with another subdomain node. For insertion, the time complexity is $O(l \log m^k)$, where l is the number of newly created subdomains. A newly created intersection partitions all existing subdomains. This is the worst case, but is unlikely to happen. The cost of deletion is the same. We notice that the cost of inserting/deleting an object is relatively high due to the cost of tree traversing, so the proposed I-tree is more suitable for indexing the data sets that are collected and stored in batch, but may not be very efficient for dynamic databases where data items are changing frequently. Nevertheless, since different part of I-tree does not interdependent on each other, store and update I-tree in a distributed manner can mitigate its relatively higher updating cost.

3.5 Implementation and Evaluation

3.5.1 Language Extension

To integrate FQ as a query primitive into a database management system (DBMS), we extend SQL for users to define functions on a database and issue FQs. We give the syntax of the commands and their usage examples as follows. Let D be a database with numerical attributes $\{A_1, A_2, \cdots, A_m\}$. We extend SQL's `CREATE` statement for users to specify how to interpret the database as a set of functions.

```
CREATE Functions $F(x_1, x_2, \cdots, x_m)$
AS $A_1 \cdot x_1 + A_2 \cdot x_2 + \cdots + A_m \cdot x_m$
FROM $D$
ON [$x_1^{min}, x_1^{max}$], [$x_2^{min}, x_2^{max}$], $\cdots$, [$x_m^{min}, x_m^{max}$]
```

The statement specifies a function definition F, by which the system interprets each record r_i in D as a function $F_i(x_1, x_2, \cdots, x_m) = r_i.A_1 \cdot x_1 + r_i.A_2 \cdot x_2 + \cdots + r_i. \cdot x_i + \cdots + r_i.A_m \cdot x_m$. The `ON` clause denotes the domain on which the functions are defined, where x_i^{min} and x_i^{max} are the allowed range of variable x_i $(1 \leq i \leq m)$. As an example, the following statement lets the system interpret each property in Table 1.1 as a *CashFlow* function, where the allowed function input d is set from 0 to 1, r from 0 to 1, and n from 0 to 1000.

```
CREATE Functions $CashFlow(d, r, n)$
AS $Income - Expense - Price \cdot (1 - d) \cdot \frac{r(1+r)^n}{(1+r)^n - 1}$
FROM $RentalProperties$
ON $[0,1]$, $[0,1]$, $[0,1000]$
```

After the CREATE statement, users can then perform FQs over the database. A range FQ $Q_{range} = (F, X, L, U)$ consists of four parameters: F is the name of a function definition, $X = (x_1, x_2, \cdots, x_m)$ the function input, L the lower bound, and U the upper bound of the query range. We extend SQL's SELECT statement to specify a range FQ:

```
SELECT $\cdots$ FROM $D$
WHERE $L \leq F(X) \leq U$
```

As an example, the following statement retrieves the rentals which generate a cash flow in between \$1000 and \$3000 with a down payment of 15% of their prices and a 15-year mortgage at 0.5% monthly interest rate:

```
SELECT $PID$ FROM $RentalProperties$
WHERE $1000 \leq CashFlow(0.15, 0.005, 180) \leq 3000$
```

The statement for a top-k FQ $Q_{top\_k} = (F, X, K)$ has three parameters: F is the name of a function definition, $X = (x_1, x_2, \cdots, x_m)$ the function input, and K the number of records to return, is as follows.

```
SELECT TOP $K$ $\cdots$ FROM $D$
ORDER BY $F(X)$ ASC|DESC
```

As an example, the following statement retrieves the rental properties which generate the highest cash flow with input $(d = 0.15, r = 0.005, n = 180)$:

```
SELECT TOP 1 PID FROM RentalProperties
ORDER BY CashFlow(0.15, 0.005, 180) DESC
```

Finally, a kNN FQ $Q_{kNN} = (F, X, K, V)$ has an additional parameter V, a user-supplied value (query point). The following statement retrieves the 2 properties whose cash flow is closest to $2000 with input $(d = 0.15, r = 0.005, n = 180)$:

```
SELECT NEAREST 2 PID FROM RentalProperties
WHERE CashFlow(0.15, 0.005, 180) = 2000
```

3.5.2 Experimental Settings

We implement three versions of the proposed technique:

1) **StandardFQ**: The sorted function list in each subdomain node is computed and stored when building the I-tree. There is no limitation on the maximum leaf number.

2) **AdvancedFQ**: Only the root list L_0 is stored. Others are computed on the fly when an FQ with an input in the corresponding subdomain arrives. A sorted function list is then cached for future use. The cache size is set to 512MB and the list with the least access is discarded first. The maximal leaf number is also limited to be $n/2$, where n is the total number of functions.

3) **SuperFQ**: A single I-tree is used to support multiple function definitions on a same database. In contrast, one I-tree is built for each function definition in the above two schemes.

We use C++ to implement the I-tree index and query processor, and C# to implement user interfaces and query parser. Our system runs on a Windows server with an Intel Xeon 64-bit 8-core 2.93GHz CPU and 32GB RAM. FQs can be applied on a database of continuous functions, and a database where each record is originally discrete values but interpreted as a function. Accordingly, we evaluate the performance of FQs on two different types of databases.

3.5.3 Querying a Database of Functions

There is little work on querying a database that stores a set of functions. So we compare our technique with **LinearSearch**. Given an FQ, this scheme computes each function with the user-supplied input to identify those satisfying the output condition. We use synthetical data in this study. To generate a dataset, we first generate a function definition, which has input variables $X = \{x_1, x_2, ..., x_m\}$ in the form of $F(X) = \sum_{i=0}^{m} c_i \times g_i(x_i^{d_i})$. Here $g_i(\cdot)$ is some computation applied on $x_i^{d_i}$, which we randomly select from the following univariant functions: $g_i(x) = x$, $g_i(x) = \sin(x)$, $g_i(x) = \cos(x)$, $g_i(x) = 1/x$. Recall that FQ can support such complex computations through function conversion. For each term, the degree d_i of x_i is uniformly generated in $[1, d_{max}]$, where d_{max} is the max degree allowed. With the function definition in place, we then generate the coefficients for each function, where each coefficient c_i is uniformly randomly selected in a range. The ranges of the parameters and their default values are given in Table 3.1.

Table 3.1: Experiment Settings

Parameters	**Range**	**Default**
Number of functions	10^5 - 10^6	5x10^5
Coefficient (c_i)	$[-10, 10]$	N/A
Number of variables (m)	1 - 10	5
Degree of variables (d_i)	1 - 5	3
Variable domain	$[-1, 1]$ - $[-20, 20]$	$[-10, 10]$

Since each dataset is originally continuous functions, we focus on three techniques, i.e., LinearSearch, StandardFQ, and AdvancedFQ. We are mainly interested in how the performance of these techniques in terms of their actual *query processing time* is impacted by the number of functions, number of variables, and domain sizes. Regardless of its type, processing an FQ takes two main steps: 1) traversing the I-tree to the subdomain node whose subdomain contains the function input, and 2) retrieving the query results from the corresponding sorted functions. For the second step, since the functions are sorted, we can perform a binary search to locate the first function that satisfies the output condition and then sequentially retrieve the following functions until reaching a function that is not in the query result. So the time of performing the second step is largely determined by the number of functions in a query result. To help us focus on the performance of I-tree, we choose to perform only top-1 FQs. Specifically, for each simulation run, we issue 1000 top-1 FQs with a randomly selected input X. The results are plotted in Figure 3.4. In general, the query processing time of both StandardFQ and AdvancedFQ grows logarithmically w.r.t. the number of functions, the number of variables, as well as the domain size. In contrast, the cost of LinearSearch increases linearly as expected. These results demonstrate that the proposed techniques have good scalability especially when dealing with large datasets. Since the query processing time is the actual running time, which is dynamic and different for different machines, we also report the percentage of the functions accessed in query processing, which is reported in Figure 3.4-(d). Note that for LinearSearch, processing a top-1 query requires to compute all functions. As such, its performance curve is flat.

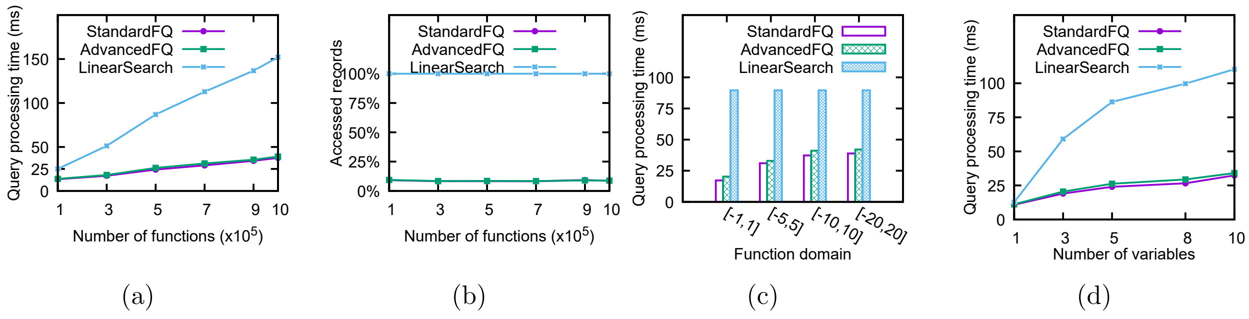


Figure 3.4: Query processing cost of Top-1 FQ

3.5.4 Querying Discrete Database as Functions

By interpreting a database of discrete values as a set of functions, FQs support analysis-based data retrieval. In this scenario, top-k FQ and range FQ are equivalent to existing *top-k query* and *scalar product query*, respectively, in terms of their functionality. For comparison, we implement two state-of-the-art techniques:

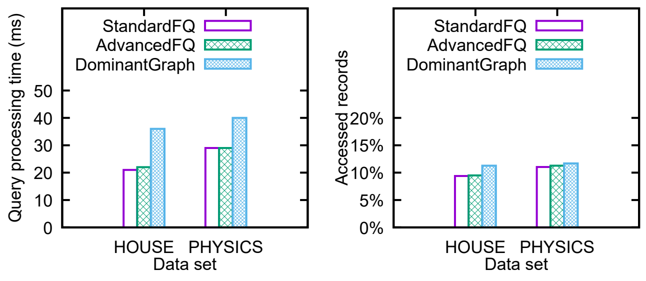
1) **DominantGraph** (Zou and Chen (2008)): This is the most efficient technique for top-k query. It allows a weight-based *Utility Function* $f(w_1, w_2, ..., w_m) = w_1 \cdot a_1 + w_2 \cdot a_2 + ... + w_m \cdot a_m$, where w_i is a user-defined weight applied on attribute a_i $(1 \leq i \leq m)$. The domain of each variable must be normalized to $[0, 1]$ and their sum equal to 1. DominantGraph facilitates efficient query processing by identifying the dominating relationships among records and then partitioning the dataset into different layers such that the records in a higher layer dominate those in a lower layer for any weight assignment. A record r_i is said to dominate another record r_j if there exists no set of weights that r_i ranks lower than r_j. Thus r_j cannot be included in a query result unless r_i is included. DominantGraph supports the output condition of top k, but not range.

2) **ScalarProduct** (Khan et al. (2014)): This technique supports scalar product query. The query receives its name "scalar product" as the function in the query must be in the form of a scalar product between a set of user-defined variables and some attributes of a dataset. The proposed technique indexes query hyperplanes for efficient query processing. It supports the inequality output condition (i.e., greater or smaller than a threshold), but not top k.

3.5.4.1 Query processing Cost

For this experiment, we use two real-world discrete datasets HOUSE and PHYSICS. HOUSE is extracted from (Ruggles et al. (2015)), including 100,000 records with 12 numerical attributes such as house value, household income, and monthly mortgage payment. PHYSICS is the KDD cup quantum physics dataset (KDD (2004)), which contains 100,000 records with 78 numerical attributes.

We first compare our techniques with DominantGraph. For DominantGraph to work, we use only weight-based function definition on the datasets and normalize the parameters. For each simulation run, we issue 1000 top-k FQs (equivalent to top-k query in this setting) and calculate their average query processing time and the percentage of records accessed during query processing. The k value (number of records to retrieve) of each query is randomly chosen from $[10, 500]$. Figure 3.5 shows the results. StandardFQ and AdvancedFQ have similar performance in query processing

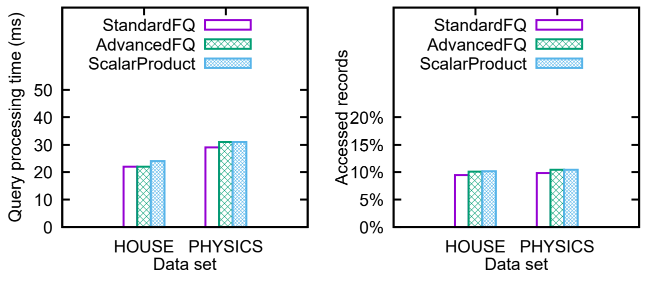


(a) (b)

Figure 3.5: Top-k FQ vs. DominantGraph

time. Recall AdvancedFQ sorts the functions on the fly while searching for the subdomain that contains a given function input. This study indicates that the cost of such on-the-fly sorting is negligible. On both datasets, our technique outperforms DominantGraph.

Then, we compare our techniques with ScalarProduct. We use randomly generated polynomials in the form of scalar product as function definitions, and issue 1000 such queries over the datasets for each experiment run. The ranges in the queries are randomly selected with a selection ratio between 1/10000 to 1/1000. Figure 3.6 shows the comparison result. Given a range query with input X, our techniques follow the I-tree to locate the subdomain that contains X and then do a binary search on the sorted function list for the functions whose output falls in between the given range. The overall cost of these two-step processing is close to that of the planar indexing in ScalarProduct. Nevertheless, our techniques are more versatile, because the latter does not support output conditions such as top-k or kNN.

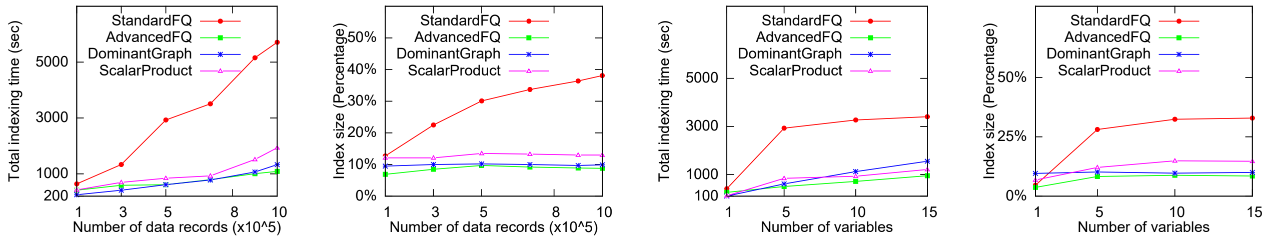


(a) (b)

Figure 3.6: Range FQ vs. ScalarProduct

3.5.4.2 Indexing Cost

We build an I-tree to support efficient processing of FQs. Similarly, DominantGraph and ScalarProduct need to build their own indexing structure. In this study, we compare the cost of building these indexes. We choose the time of building an index and the size of the index (in terms of the percentage of the original dataset size) as performance metrics. Figure 3.7-(a) and (b) show the impact of the number of data records. As the number of records increases, all techniques require more time in index building and result in larger index sizes (yet the percentage to the data size remains stable). StandardFQ is the worst performer. This is not surprising because this scheme pre-sorts functions for each of the subdomains partitioned by the functions and stores the sorted function lists for query processing. AdvancedFQ addresses this problem and achieves a performance similar to ScalarProduct. It is worth mentioning that this performance improvement is not achieved at the expense of query processing time. As discussed previously, AdvancedFQ and StandardFQ perform similarly in query processing (see 3.4). While AdvancedFQ and ScalarProduct incur similar indexing time, the former constantly outperforms the latter in terms of index size. Figure 3.7-(c) and (d) show the impact of variable number. The results are similar to the impact of record numbers. This study demonstrates the advantages of our solution: It supports more query types than other techniques combined, and it is more efficient in query processing, at similar indexing costs.



(a) (b) (c) (d)

Figure 3.7: Indexing cost

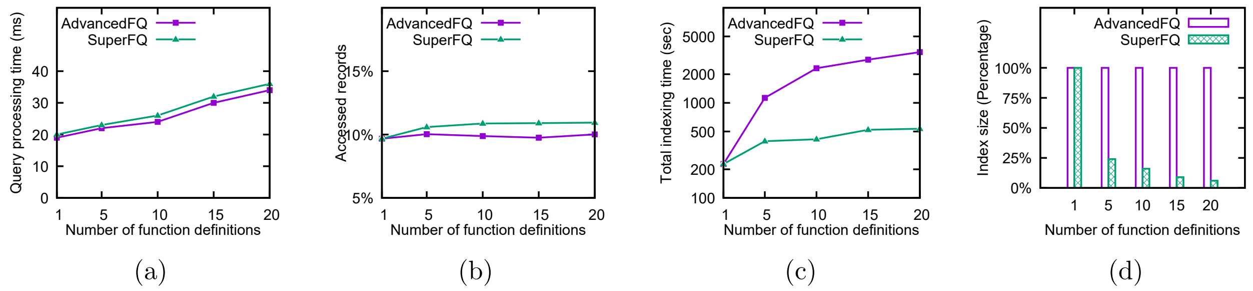


Figure 3.8: Handling multiple function definitions

3.5.4.3 Handling Multiple Function Definitions

Our solution also features supporting multiple function definitions over a same discrete database. We implement this *super function* technique, namely *SuperFQ*, and compare it with AdvancedFQ. We use synthetic datasets in this study for the sake of flexibility. We generate 50,000 data records, where each record has 10 numerical attributes and a unique ID. The value of each numerical attribute of a record is randomly selected in the range of $[-1, 1]$. Since complex functions can be converted, we focus on linear functions. We use the approach described in Chapter 3.5.3 to generate function definitions, but make $g_i(x) = x$ always. The number of functions definitions ranges from 1 to 20, so each record can be interpreted as up to 20 different multivariate linear functions.

For each simulation run, we issue 300 top-1 FQs. We choose not to perform other types of queries for the same reason explained in Chapter 3.5.3. Figure 3.8 show the average query processing time under the two techniques. AdvancedFQ incurs slightly less query process time. Since it builds one independent I-tree for each function definition, less nodes need to be traversed in locating the subdomain that contains the input for a given query. The minor gains, however, comes at a major cost – the time of building the I-trees and their aggregate size increase sharply as the number of function definitions increases.

3.6 Discussion

The need to support functions, which are often the best in represent continuous data, has been well recognized by both academia and industry. Various techniques (e.g., Thiagarajan and Madden

(2008); Guo et al. (2013); Katsis et al. (2015); Anagnostopoulos and Triantafillou (2017)) are now available to support function as a primitive datatype in a DBMS and allow users to query the data represented by a function. Commercial DBMS such as DB2 10.5 and PostgreSQL 9.4 have also recently supported expression-based indexes over numeric attributes. Our work extends this track of research by enabling users to retrieve the functions whose output under a user-supplied input satisfies a certain condition.

Our research also complements existing work on analytic queries (e.g., Vlachou et al. (2011); Zhang et al. (2014a,b); Mouratidis et al. (2015); Cheema et al. (2014); Das et al. (2007); Gao et al. (2015); He and Lo (2014); Peng and Wong (2015); Tang et al. (2017); Yang and Cai (2017)) developed for analysis-based data retrieval on discrete databases. For efficient query processing, a range of techniques have been proposed. These techniques, which are vastly different, share a common characteristic, i.e, they all index their data as discrete values. Our research demonstrates a new strategy, which is to interpret and index a database of discrete values as a set of functions. We show that a single indexing structure (i.e., the proposed I-tree) can support both top-k query and scalar product query, and do so more efficiently when compared to their corresponding the state-of-the-art techniques. In addition to top-k query and scalar product query, our techniques have the potential to support other queries all together. In the following, we discuss three representative queries, without giving detailed analysis due to limited space.

Reverse top-k query (Vlachou et al. (2011)): Given a set of records and a set of top-k queries, a reverse top-k query retrieves all top-k queries whose result contains a selected record. A query here is a weight vector that assigns a weight to each attribute. As such, we can treat the top-k queries as a set of FQs which are different in their inputs. On an I-tree, all queries whose inputs belong to the same subdomain will rank the functions in exactly the same order and do not need to be evaluated separately. Therefore, these queries can processed efficiently as follows: For each query, we find the subdomain that contains its input, and for each of the subdomains that contain a query's input, we locate its sorted function list. If the top k functions in the list contains a selected record, then we return all queries whose inputs fall in the subdomain.

Maximum rank query (Mouratidis et al. (2015)): Such a query computes the highest possible rank that a record can achieve for any linear scoring function. We can slightly modify the first-cut algorithm (FCA) proposed in (Mouratidis et al. (2015)) to process the maximum rank query with the support of I-tree. FCA is also based on examining function intersections. Recall that when two functions intersect, their ranks switch. As such, the maximal rank of a function f_i can only be achieved when f_i switches order with another function f_j, whose rank was higher than that of f_i. On our I-tree, the rank-switching takes place when a search path leaves an intersection node of f_i and f_j. Thus we can examine all such nodes involving f_i, and traverse all paths from the root node to these intersection nodes. The highest rank that f_i can achieve is equal to the highest rank that f_i achieves on any of these paths.

Global immutable region (Zhang et al. (2014a)): Given a top-k query, its global immutable region (GIR) consists of all inputs under which the query result remains unchanged. On the I-tree, GIR is the union of all subdomains in which the result of a given query point q remains the same. To find these subdomains, we first locate the subdomain that contains the query using the I-tree, and retrieve the query result (top-k records) R. Then we can construct the GIR by finding its boundaries. A function intersection is the boundary of a GIR if and only if moving q across the boundary will cause the query result to change. Thus, only intersections involving functions in R could be a GIR boundary. It is fairly easy to locate such intersections with the I-tree, since it stores all intersections.

3.7 Summary of the Chapter

The notion of *Function Query* (FQ) is a powerful extension to existing database query languages. It can be applied on a database that is originally a set of functions, e.g., representing some data that is continuous in nature. While this is obvious, FQ can also be applied, which we believe is less apparent but inspiring, on a database of discrete values. By interpreting each record as a function, FQ supports analysis-based information retrieval like existing top-k and scalar-product queries. However, unlike these existing queries, FQ supports more complex functions and a variety of output conditions.

We have addressed the challenges of enabling efficient FQ execution. Our key observation is, a set of functions can be sorted based on their outputs in the input subdomains partitioned by the intersections of these functions. This observation alone, however, is not sufficient to develop a good solution. Finding the exact boundaries and sorting the functions in each subdomain can be computation-intensive. The problem is even more complicate when having to deal with a very large number of subdomains, which happens when the functions involve non-linear terms or are non-polynomial. To circumvent these problems, we proposed a novel data structure called *Intersection-tree* (I-tree). I-tree indexes the subdomains created by function intersections and allows one to sort the functions for each subdomain, without having to computing subdomain boundaries. With I-tree in place, we proposed to convert complex functions into multivariate linear polynomial functions through variable replacement. We show that this strategy works for all polynomial functions and non-polynomial functions that conform certain form. Moreover, it becomes possible to handle multiple functions definitions on the same database with a single I-tree. While I-tree is mainly developed for FQs, we show that it can also be used to support the execution of some other well-known analytic queries, including reverse top-k query, maximum rank query and global immutable region.

In addition to algorithms and data structures, we have integrated FQs into a database system as a query primitive. We evaluated the proposed techniques through prototyping and experiments over synthetic data and real-world data, and our techniques exhibit excellent performance in our extensive evaluation.

CHAPTER 4. AUTHENTICATION OF FUNCTION QUERIES

4.1 Motivation

The proliferation of digital technologies has created data explosion in every segment of our society. To many data owners, managing their data is a big challenge, both technically and financially. This is especially true for certain big data applications where the data is so large and complex that cannot be handled by standard database management tools.

Processing function queries over large datasets is particularly challenging given the complexity of the queries. Instead of managing the data and processing queries by their own, an alternative approach is to outsource these duties to a third-party cloud services such as Amazon and IBM. By releasing data owners from day-to-day responsibility of software and hardware management, outsourcing has the potential to save them significant operation cost. Data owners, however, are facing with difficulties in fully trusting such services. Ultimately, the data is in the hand of a third party that is beyond their own administrative domain. Today's computing and networking systems are inherently insecure and vulnerable to attacks. The third party may also intentionally deliver incomplete results or even manipulate the data for its own benefits.

The above problem has motivated significant research effort on *query authentication*(e.g., Devanbu et al. (2003); Pang et al. (2005)), i.e., enabling users to verify the query results they receive are indeed correct. A query result is deemed correct if it is both *sound* and *complete*. The former requires that every data item in the query result appears in the original database, whereas the latter, every data item in the original database that satisfies the query condition is included in the query result. Existing authentication techniques, which we discussed in Chapter 2, are developed for simple queries over discrete data. They cannot be applied directly for FQ authentication. In their case, the raw data is compared directly against a query condition. The data owner can sort and digitally sign the raw data before releasing to the service provider. In contrast, an FQ retrieves the functions whose outputs under a given function input satisfy certain condition (e.g., within a certain range). When processing an FQ, the server needs to compute the functions based on user-supplied arguments. The computation results, not the raw data, are then checked against the query condition. The data owner cannot sort and sign the computation results because the input parameters to the functions are supplied by query issuers and not known when the database is outsourced to the service provider. It may first appear that one can compute the continuous functions with a series of discrete parameter values. The computation results can then be sorted and digitally signed with existing authentication techniques. This approach can provide only approximate query results, where the accuracy is determined by the interval used in parameter discretization. A large interval provides very coarse results, while a fine interval will generate an overwhelming amount of data for the data owner to sign, which will in turn dramatically increase the cost of outsourcing.

In this chapter, we present our techniques for authenticating FQs. Again, our research leverages the fact that the intersections of a set of functions partition the input domain into a number of subdomains, and in each of these subdomains, the functions can be sorted. We consider various kinds of math functions, starting from the simplest univariate linear functions, to multivariate linear functions, and finally multivariate functions with higher degree. The performance of the proposed techniques is evaluated though theoretical analysis, simulation and empirical study.

4.2 Problem Formulation

4.2.1 System Model

The outsourcing model consists of three parties (Figure 4.1), data owner, third-party server, and data user. Let D be a database. The data owner uploads D along with a *function definition* to the server, which manages the data and answers all user queries. The function definition serves as a template for the server to interpret each record in D as a function. Since we are mainly concerned of authenticating the queries over the functions, we will simply refer to the outsourced data as a set of functions and denote this set as $F = \{f_1, f_, \cdots, f_n\}$, where f_i is the function represented by record r_i in D under a given function definition.

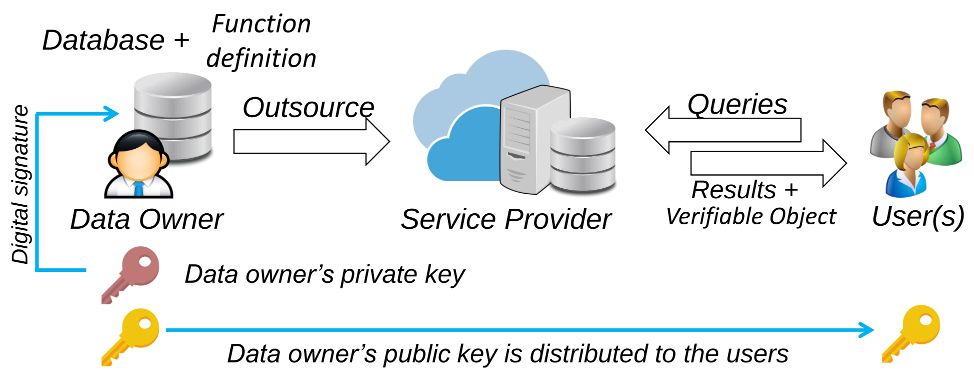


Figure 4.1: The three-party outsourcing model

In this research, we consider the polynomial functions where each term has at most one variable. Such functions are generic enough for many applications, including all those mentioned in the introduction. Let k be the maximum number of variables and d the highest degree of each term. We have the following general form for each function:

$$\begin{aligned} f(x_1, x_2, ..., x_k) = \quad & c_0 + \\ & c_{11}x_1 + c_{12}x_1^2 + \cdots + c_{1d}x_1^d + \\ & c_{21}x_2 + c_{22}x_2^2 + \cdots + c_{2d}x_2^d + \\ & \cdots \\ & c_{k1}x_k + c_{k2}x_k^2 + \cdots + c_{kd}x_k^d \end{aligned} \tag{4.1}$$

Here $X = (x_1, x_2, ..., x_k)$ is a set of variables and c_{ij} the coefficient for term x_i^j, where $1 \le i \le k$ and $1 \le j \le d$.

4.2.2 Adversary Model and Security Goal

To query the functions in F, a user provides an input $X = (x_1, x_2, ..., x_k)$ and a query condition. We consider the three types of FQs introduced in the previous chapter, including range, top-k, and kNN. In response to an FQ, the server returns a set of functions (and their corresponding data records) whose outputs under the given input satisfy the query condition. Users are concerned if the query results they receive are correct. The server may return wrong results on purpose (e.g., to save computation and communication costs) or unknowingly (e.g., compromised by hackers). Let $q = (X, l, u)$ be a range FQ submitted by a user, $R(q)$ the query result received by the user, $f_i(\cdot)$ the function corresponding to record r_i. We say $R(q)$ is correct if the following two requirements are satisfied:

- *Soundness*: $\forall r_i \in R(q)$ appears in the original database, and $l \leq f_i(X) \leq u$;
- *Completeness*: $\forall r_i$ in the original database D, if $l \leq f_i(X) \leq u$, we have $r_i \in R(q)$.

For top-k and kNN FQs, soundness and completeness of their results can be defined similarly. Our research is to develop an approach that allows users to verify if the query results they receive are indeed correct.

4.3 Proposed Solution

We start from the simplest case, univariate linear function, where variable number $k = 1$ and degree $d = 1$. Let F be a set of n univariate linear functions, each in the form of $f_i(x) = a_i x + b_i$. As pointed out in the overview of Intersection-tree, the functions in F are sortable in each subdomain created by their intersections. In this case, each intersection of two functions is a single point on the x-axle and hence each intersection is an interval. For example, in Figure 3.1-(a), we have $f_1(\cdot) \leq f_3(\cdot) \leq f_2(\cdot)$ for any input in $[x_1, x_2)$.

This observation allows us to create an authentication structure for verifying FQ results. That is, the data owner can digitally sign the sorted functions in each subdomain and create a corresponding signature chain for the subdomain. To authenticated an FQ, the server needs to locate the subdomain that contains the user-specified function input, and then constructs a VO using the corresponding signature chain created for that subdomain. Unfortunately, the potentially huge number of subdomains indicates the data owner will have to create $O(2^n)$ signature chains, each contains n data records. This process can be very expensive especially when the number of functions is large. To address this challenge, we present a highly efficient authentication structure namely *Signature Mesh.*

4.3.1 Signature Mesh Construction

Given a set of n univariate linear functions F to be outsourced, the data owner computes and sorts the set of intersection points of F, say $I = \{x_1, x_2, \cdots, x_m\}$, where $x_1 < x_2 < \cdots < x_m$. For each interval $[x_i, x_{i+1})$, the data owner sorts the list of functions according to their outputs. Let $\{f_1(\cdot), f_2(\cdot), \cdots, f_n(\cdot)\}$ be the sorted list of functions for interval $[x_i, x_{i+1})$. Assuming non-decreasing order, we have $f_1(\cdot) \leq f_2(\cdot) \leq \cdots \leq f_n(\cdot)$ for any input x in $[x_i, x_{i+1})$. On this sorted list, the data owner builds a digital chain as follows. For each pair of contiguous functions $f_j(\cdot)$ and $f_{j+1}(\cdot)$ in the sorted list, it creates a signature using the digest of their corresponding records:

$$Sig_{[x_i,x_{i+1})}(r_j|r_{j+1}) = Sig(H(H(r_j)|H(r_{j+1})|x_i|x_{i+1}))) \tag{4.2}$$

Here r_j and r_{j+1} are the corresponding records of function $f_j(\cdot)$ and $f_{j+1}(\cdot)$. $H(\cdot)$ can be a collision resistant one-way hash function (e.g., SHA1 (Burrows (1995))) or other digest functions such as the one proposed in (Pang et al. (2005)). Note that in addition to the digests of r_j and r_{j+1}, the two intersection points x_i and x_{i+1} are included in the signature. This is to allow a user to verify if the signature indeed belongs to the signature chain built for the interval $[x_i, x_{i+1})$. By examining this signature, a user can be assured that there exists no function $f \in F$ whose output is in between the outputs of $f_j(x)$ and $f_{j+1}(x)$ for any x in $[x_i, x_{i+1})$. In other words, $f_j(\cdot)$ and $f_{j+1}(\cdot)$ are contiguous in the original sorted list.

In the above approach, the data owner creates a signature chain for each interval and each chain has $n-1$ signatures. The total number of signatures need to be created for the entire domain can actually be significantly reduced. Given two univariate linear functions, they can intersect at most one time (unless they have the same coefficients, which we will discuss later in this chapter). This means that two functions can remain contiguous in a number of consecutive intervals. In this case, only one signature needs to be created for this pair of functions to cover the whole range. For example, if $f_j(\cdot)$ is immediate before $f_{j+1}(\cdot)$ from intersection point x_p to intersection point x_q (where they swap their order), then we can create one signature $Sig_{(x_p,x_q)}(r_j|r_{j+1})$ for the two functions from x_p to x_q, instead of creating $q-p$ signatures, $Sig_{[x_p,x_{p+1})}(r_j|r_{j+1})$, ..., $Sig_{[x_{q-1},x_q)}(r_j|r_{j+1})$, one for each interval within the whole range from x_p to x_q. Algorithm 4 describes the process of constructing the signatures.

Algorithm 4 ConstructMesh($F = \{f_1, f_2, ...f_n\}$)

```
1:  Compute all intersection points of $F$
2:  $I \leftarrow \{x_1, x_2, ...x_m\}$
3:  Sort $F$ in $(-\infty, x_1)$
4:  Create signature chain for $F$ in $(-\infty, x_1)$
5:  For i = 0 to m, do:
6:    If $f_a$ and $f_b$ intersect on $x_i \in I$
7:      Swap $f_i$ and $f_j$ in $F$
8:    For j = 1 to n, do:
9:      If $Sig(r_j|r_{j+1})$ exists in previous interval, then
10:       Extend the range of $Sig(r_j|r_{j+1})$ to cover $[x_i, x_{i+1})$
11:     Else create $Sig(r_j|r_{j+1})$ for $[x_i, x_{i+1})$
```

Figure 4.2 illustrates the signatures created for a set of six functions. We refer to the whole set of signatures created by the data owner as a *Signature Mesh*. This mesh is given to the service provider together with the original database to be outsourced.

4.3.2 Query Result Verification

We first show how to verify correctness of range FQ results using the above scheme. Upon receiving a range query $q = (X, l, u)$, the server retrieves all functions whose computation results with the user-defined input x is within the specified range l and u. We assume some efficient techniques are used for the server to find the functions that satisfy the query conditions and will not concern ourselves with their implementation details. Our focus is on the verification of the query result.

Let $R(q) = \{r_a, r_{a+1}, \cdots, r_{b-1}, r_b\}$ be the sorted query result, r_{a-1} the record immediately before r_a and r_{b+1} the record immediately after r_b. Let $f_i(\cdot)$ be the function in record r_i, where $a-1 \leq i \leq b+1$. We have $f_{a-1}(\cdot) < l \leq f_a(\cdot) < \cdots < f_b(\cdot) < f_{b+1}(\cdot)$ for the given input x.

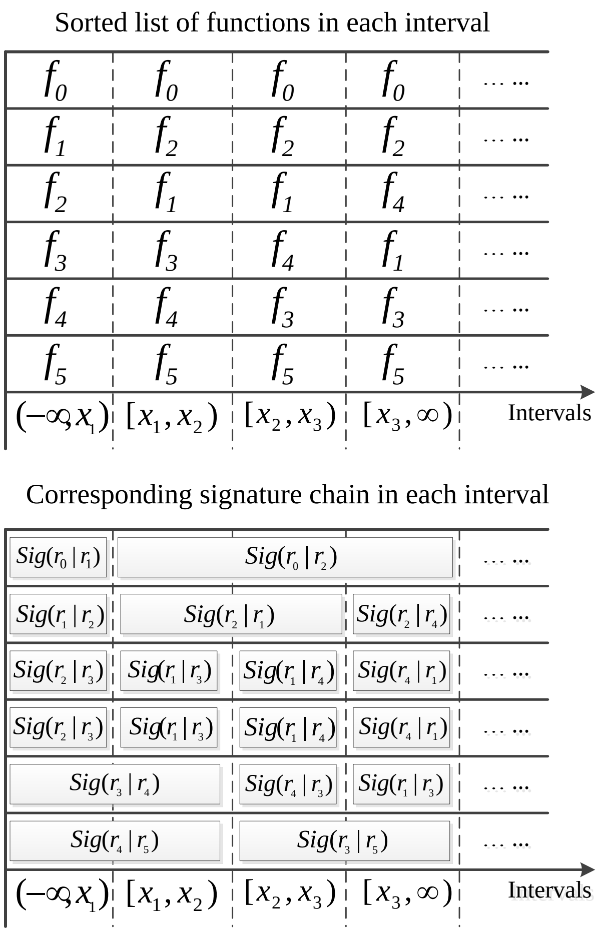

Sorted list of functions in each interval
f_0 f_0 f_0 f_0 … ...
f_1 f_2 f_2 f_2 … ...
f_2 f_1 f_1 f_4 … ...
f_3 f_3 f_4 f_1 … ...
f_4 f_4 f_3 f_3 … ...
f_5 f_5 f_5 f_5 … ...
$(-\infty, x_1)$ $[x_1, x_2)$ $[x_2, x_3)$ $[x_3, \infty)$
Intervals
Corresponding signature chain in each interval
$Sig(r_0 | r_1)$ $Sig(r_0 | r_2)$ … ...
$Sig(r_1 | r_2)$ $Sig(r_2 | r_1)$ $Sig(r_2 | r_4)$ … ...
$Sig(r_2 | r_3)$ $Sig(r_1 | r_3)$ $Sig(r_1 | r_4)$ $Sig(r_4 | r_1)$ … ...
$Sig(r_2 | r_3)$ $Sig(r_1 | r_3)$ $Sig(r_1 | r_4)$ $Sig(r_4 | r_1)$ … ...
$Sig(r_3 | r_4)$ $Sig(r_4 | r_3)$ $Sig(r_1 | r_3)$ … ...
$Sig(r_4 | r_5)$ $Sig(r_3 | r_5)$ … ...
$(-\infty, x_1)$ $[x_1, x_2)$ $[x_2, x_3)$ $[x_3, \infty)$
Intervals

Figure 4.2: Signature mesh created for $F = \{f_0, f_1, f_2, f_3, f_4, f_5\}$.

The server generates the corresponding verification object $VO(q)$ as follows. For each record r_i $(a-1 \leq i \leq b)$ in $\{r_{a-1}\} \cup R(q)$, the server finds $Sig_{[x_p,x_q)}(r_i|r_{i+1})$ in the signature mesh, where the interval $[x_p, x_q)$ must contain x. This signature and the corresponding interval $[x_p, x_q)$ are added to $VO(q)$. Moreover, the two boundary records r_{a-1} and r_{b+1} are included in $VO(q)$. Algorithm 5 describes the process of creating the verifiable object for a query. An example of verification object is showed in Figure 4.3.

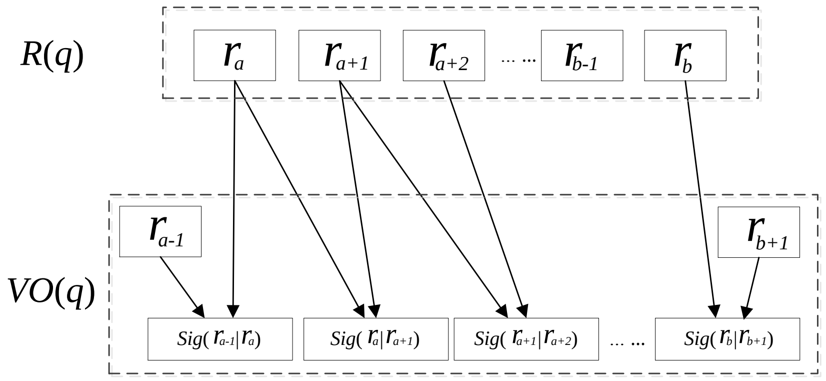


Figure 4.3: Verification object for $R(q) = \{r_a, r_{a+1}, \cdots, r_{b-1}, r_b\}$.

Algorithm 5 ConstructVO($q =< x, l, u >$, $R(q) = \{r_a, r_{a+1}, ...r_b\}$)

```
1: $VO(q) \leftarrow \{r_{a-1}, r_{b+1}\}$
2: For $i = a-1$ to $b$, do:
3:   Locate the interval $[x_i, x_{i+1})$ that contains $x$.
4:   Find $Sig_{[x_p,x_q)}(r_i|r_{i+1})$ such that $[x_i, x_{i+1}) \subseteq [x_p, x_q)$
5:   Add $Sig_{[x_p,x_q)}(r_i|r_{i+1})$ into $VO(q)$
6: Return $VO(q)$
```

The server sends both $R(q)$ and $VO(q)$ to the user, which verifies the query result as follows. For each pair of consecutive records r_i and r_{i+1}, where $a-1 \leq i \leq b$, the user finds the signature $Sig_{[x_p,x_q)}(r_i|r_{i+1})$ and the corresponding interval $[x_p, x_q)$ in $VO(q)$. If x does not belong to the interval, the user rejects $R(q)$. Otherwise, it computes the following digest, using the data owner's public key:

$$Sig^{-1}(Sig_{[x_p,x_q)}(r_i|r_{i+1})) = H(H(r_i)|H(r_{i+1})|x_p|x_q) \tag{4.3}$$

This digest is compared with the corresponding digest included in $VO(q)$. If the two digests do not match, the user rejects $R(q)$. Otherwise, it means r_i and r_{i+1} are in the original order and there is no record in between which satisfies the query condition, and the user proceeds to check the correctness of boundary records by verifying if the following boundary conditions hold:

$$f_{a-1}(x) < l \leq f_a(x) \leq f_b(x) \leq u < f_{b+1}(x) \tag{4.4}$$

One special case is when the query result contains the very first and/or last record in the sorted list of functions. In this case, there is no boundary function for a user to verify. This problem can be circumvented by inserting two dummy functions, $f_0(\cdot) = -\infty$ and $f_{n+1}(\cdot) = +\infty$, to the sorted list and creating corresponding signatures to add them to the signature chain. The two functions serve as the virtual boundaries for any query result that contains the first and/or the last record. A more formal description of the above verification process is given in Algorithm 6. The dummy functions also provide us a way to verify a top-k FQ, which retrieves the functions whose outputs are among the K smallest with user specified input. For a top-k FQ, $VO(q)$ includes the K records and the dummy function $f_0(\cdot) = -\infty$. Obviously, if their signature indicates that the $K+1$ records are contiguous, the result must be correct.

The process of authenticating top-k and kNN FQs is similar. Specifically, for a top-k FQ $q = (x, K)$, $VO(q)$ contains $Sig_{[x_p,x_q)}(r_0|r_1)$, $Sig_{[x_p,x_q)}(r_1|r_2)$, ..., $Sig_{[x_p,x_q)}(r_K|r_{K+1})$, where r_0 is the dummy record corresponding to $f_0(\cdot) = -\infty$, and r_1, r_2, ..., and r_K are the top-k records in interval $[x_p, x_q)$. It is easy to see r_0, r_1, r_2, ..., and r_K must be contiguous in the sorted list and this can be verified by examining the signature chain. Similarly, for a kNN FQ $q = (x, K, y_0)$, $VO(q)$ contain the signatures of the K contiguous records, along with two extra boundary records, whose function outputs are among the K nearest neighbour of y_0 in a interval $[x_p, x_q)$ that contains x. Note that only verifying if the records are contiguous in $R(q)$ is not sufficient for a kNN FQ, since

Algorithm 6 Verify($q = (x, l, u)$, $R(q)$, $VO(q)$)

```
1: For each $Sig_{[x_p,x_q)}(r_j|r_{j+1})$ in $VO(q)$, do
2:   If $x$ is not in $[x_p, x_q)$, then
       Reject $R(q)$
3:   Else if
     $Sig^{-1}(Sig_{[x_p,x_q)}(r_j|r_{j+1})) \neq H(H(r_j)|H(r_{j+1})|x_p|x_q)$,
       then Reject R(q).
4: If $l \leq f_a(x)$ AND $f_b(x) \leq u$ AND $f_{a-1}(x) < l$
   AND $u < f_{b+1}$, then
     Accept $R(q)$.
5: Else, Reject $R(q)$.
```

the server may return a list of contiguous records on the signature chain that are not the K nearest neighbours to the query point. To solve this problem, the user first checks if there exists a record r_a in $R(q)$ such that the distance between $f_a(X)$ and y_0 is the smaller than the distance between r_{a-1} and y_0 and the distance between r_{a+1} and y_0. If this is true, r_a must be the neighbour nearest to y_0. The user then checks if the function outputs of the two boundary records have the largest distance to y_0 comparing with all the other records in the query result. This condition guarantees that there exists no other record whose function output is closer to y_0 than any record in the query result.

4.3.3 Discussion

It is possible a same function appears multiple times in a database (due to duplicate records) but with different IDs. To avoid duplicated functions from creating infinity intersection points, we merge the records with the same function into one super record that contains multiple IDs but one common set of coefficients. When the output of the function satisfies the query condition, then all these IDs/records are returned to the user.

The functions in the database may have different domains, too. For example, the data owner may specify that a function $f_i(\cdot)$ is valid only for domain $[x_p, x_q)$. When the function input specified by the client is not within its domain, $f_i(\cdot)$ should not be included in the query result. In our solution, we still consider the domain of the set of functions F as $(-\infty, +\infty)$, but treat the domain boundaries of each function as intersection points too. In the above example, we treat x_p and x_q as two intersection points. The augmented set of intersection points is used to partition the domain of F into intervals as before. If a function is not defined in an interval $[x_i, x_{i+1})$, we simply remove it from the sorted list for this interval.

4.4 Extension

We now extend the technique for univariate linear functions to support query authentication over more complex polynomials.

4.4.1 Multivariate Linear Function

The intersection of two univariate linear functions creates a point that divides the x-axis into two halves. Given two two-variable linear functions, say $f_a(x, y)$ and $f_b(x, y)$, defined on $\mathbb{R}^2$, their intersection creates a straight line in the 2D plane. Here $\mathbb{R}$ denotes the real number set. Given any point (x_i, y_i) on this line, we have $f_a(x_i, y_i) = f_b(x_i, y_i)$. In general, two k-variable linear functions intersect each other on a hyperplane in the k-dimension domain space. Consider a set of n k-variable linear functions F. Let $I = \{h_1, h_2, \cdots, h_m\}$ denote the set of intersection hyperplanes of the functions in F. Each hyperplane h_i partitions the domain space into two disjoint , denoted by $h_i > 0$ (the half space above the hyperplane) and $h_i < 0$ (the half space below the hyperplane). Therefore, the m intersection hyperplanes in I can partition the k-dimension domain space of the functions in F into $O(m^2)$ subspaces in the worst case. It is clear that the functions in F can be sorted in each subspace. Thus, for each subspace, the data owner can build a signature chain on the sorted functions for query result authentication.

The process of constructing the signature mesh is similar to that of the univariate linear case. First, the data owner computes the set of intersection hyperplanes I of F. The second step is to identify all the subspaces partitioned by I. To facilitate this process, we use a space partitioning tree similar to kd-tree as showed in Figure 4.4. Each node in the tree corresponds to a hyperplane in I. The left and right subtrees represent the subspaces above and below the hyperplane respectively.

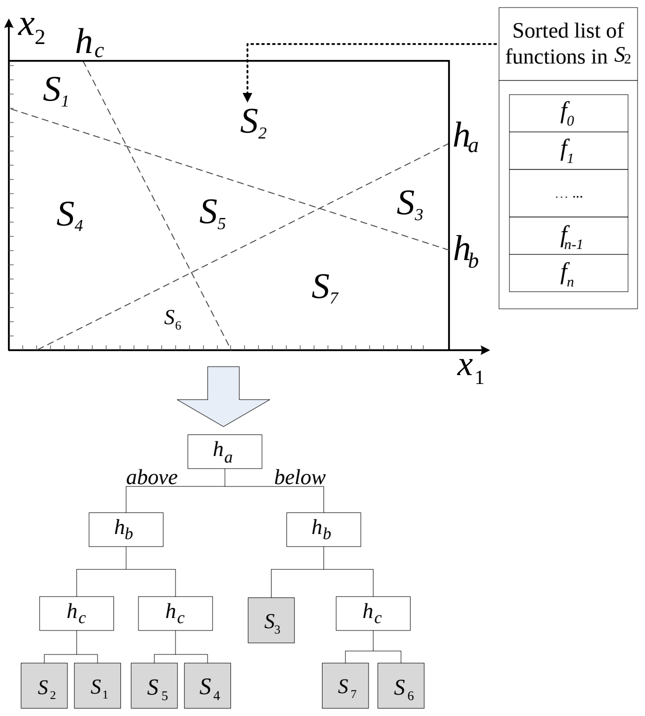


Figure 4.4: Subspace partitioning and corresponding space partitioning tree.

Thus, each leaf of the tree represents a subspace partitioned by I. Let $\{f_1(\cdot), f_2(\cdot), \cdots, f_n(\cdot)\}$ be the sorted list of the functions in F in a subspace S_i. Let r_j be the record containing $f_j(\cdot)$. For each pair of functions $f_j(\cdot)$ and $f_{j+1}(\cdot)$ which are contiguous in the sorted list, we create a signature using the digest of their records:

$$Sig_{<S_i>}(r_j|r_{j+1}) = Sig(H(H(r_j)|H(r_{j+1})|B_i)) \tag{4.5}$$

where $B_i = \{h_a \geq 0, h_{a+1} \geq 0, \cdots, h_b \geq 0\}$ is the set of k-dimension half-spaces that defines the boundary of a subspace S_i. Note that a point is in S_i if and only if it is within each half-space in B_i. For instance, in Figure 4.4, the boundary of subspace S_5 consists of three half-spaces (or half-planes in the $2D$ case), $h_a \leq 0$, $h_b \geq 0$, and $h_c \leq 0$.

It is possible that the input values specified by the client locate exactly on an intersect hyperplane that separates two (or more) adjacent subspaces S_i and S_{i+1}. In this case, either the signature chain created for S_i or that of S_{i+1} can be used to verify the correctness of the query result. This is due to the fact that the output of these functions will be the same for the given input, and thus their relative position in the sorted function list will not affect the verification process and the correctness of the query result. Therefore, we can set the boundary half-spaces of any subspaces to be a closed half-space, without affecting the verification process. In verification, the client can be assured that the signatures returned by the server is originated from the signature chain that contains its input arguments $X = (x_1, x_2, \cdots, x_k)$ by checking X against all the half-spaces in B. If any of the half-spaces does not contain X, the query result is rejected.

Usually, only a small number of the functions in F will change their order in the sorted list when crossing an intersection hyperplane. Thus, similar to the univariate linear case, if the order of two functions f_j and f_{j+1} remains unchanged in a number of consecutive subspaces, the data owner can create only one signature for this pair of functions covering all these subspaces, instead of one signature for each subspace. In the following, we show that the l subspaces where a signature are valid is always a set of consecutive subspaces which can be represented by one larger subspace, and thus can share one signature. Consider two multivariate linear functions $f_j(\cdot)$ and $f_{j+1}(\cdot)$ in the sorted list of functions in S_i. It is easy to see that their corresponding signature $Sig_{<S_i>}(r_j|r_{j+1})$ will remain valid until one of the following three events occurs:

1. $f_j(\cdot)$ and $f_{j+1}(\cdot)$ exchange their order in the sorted list.
2. $f_{j-1}(\cdot)$ and $f_j(\cdot)$ exchange their order in the sorted list.
3. $f_{j+1}(\cdot)$ and $f_{j+2}(\cdot)$ exchange their order in the sorted list.

Let $h_{i,j}$ denote the hyperplane on which $f_j(\cdot)$ and $f_{j+1}(\cdot)$ exchange their order. No matter how the domain space is partitioned, $Sig_{<S_i>}(r_j|r_{j+1})$ will be valid in the subspace defined by three half-spaces: $h_{j,j+1} \geq 0$, $h_{j-1,j} \geq 0$, $h_{j+1,j+2} \geq 0$, which together form a convex hall. An example of signature mesh for a set of two-variable linear functions in a selected range is illustrated in Figure 4.5.

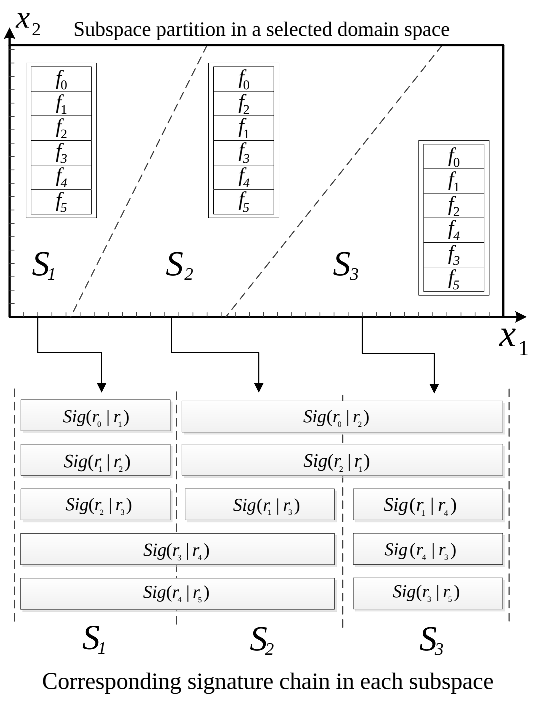


Figure 4.5: Signature mesh for a set of two-variable linear functions $F = \{f_1, f_2, f_3, f_4\}$ in a selected range.

To generate a verifiable object $VO(q)$ for $q = (X, l, u)$, the server first finds the subspace that contains the query input $X = (x_1, x_2, \cdots, x_k)$, then returns the corresponding signatures of each record in the query result, plus the two boundary records. The set of boundary half-spaces signed within each signature is also included in $VO(q)$ for the client to compute the digest and verify the signature. The verification process is basically the same as the univariate case: The client checks if the returned signatures one by one is contiguous and if its input is within the domain subspace of each signature, and finally checks if the two boundary records are valid according to (4).

4.4.2 Multivariate High Degree Function

We now consider a database where each record is treated as a multivariate function with degree up to $d > 1$, i.e., polynomials that take the general form showed in (1). The functions in the database may have different degrees, but those with degree lower than d can be considered a special case of d-degree polynomial where the coefficients of missing terms are zero.

The intersection of two polynomials with degree greater than 1 may take different forms depending on specific functions. As an example, Figure 3.1 in Chapter 3 illustrates three possible intersection forms of high degree polynomials. Nevertheless, regardless of the value of k and d, this rule remains the same: The intersection of two functions $f_i(\cdot)$ and $f_j(\cdot)$ partitions the domain space into a number of subspaces, and in each subspace, the two functions are sortable. That is, for any subspace S_i, either $f_i(X) \geq f_j(X)$ holds for $\forall X \in S_i$, or $f_j(X) \geq f_i(X)$ holds for $\forall X \in S_i$. As such, we can apply the same strategy: 1) Partition the domain space based on the intersections of the functions, 2) Sort the functions in each subspace, and 3) Construct a signature chain for each sorted function list.

The general form of the intersection of two k-variable d-degree functions defined on $\mathbb{R}^k$ can be seen as a surface with order up to d in the k-dimension domain space. Let F be a set of n k-variable d-degree polynomial functions. We define $I = \{s_1, s_2, \cdots s_m\}$ as the set of intersection surfaces of F. Again, we use these surfaces to partition the k-dimension domain space into a number of subspaces and sort F in each subspace. The process of computing I and indexing all the subspaces is the same as the previous case. Let $\{f_1(\cdot), f_2(\cdot), \cdots, f_n(\cdot)\}$ be the sorted list of functions corresponding to subspace S_i in which $f_j(\cdot)$ denotes the j-th function in the list. For each pair of two consecutive functions $f_j(\cdot)$ and $f_{j+1}(\cdot)$ in the list, we generate a signature:

$$Sig_{<S_i>}(r_j|r_{j+1}) = Sig(H(H(r_j)|H(r_{j+1})|B_i)) \tag{4.6}$$

Here, $B_i = \{s_a \geq 0, s_{a+1} \geq 0, \cdots, s_b \geq 0\}$ is the boundary of the subspaces S_i, the subspace surrounded by a set of surfaces $\{s_a, s_{a+1}, \cdots, s_{b+1}\}$ with order up to d. For example, the boundary of S_2 in Figure 3.1-(b) is $\{s_a \geq 0, s_b \leq 0\}$.

In the linear cases, we showed that only one signature needs to be created for l consecutive subspaces as long as the signature is valid in all these subspaces. However, in the high degree case, a signature might be valid in several disconnect subspaces $\{S_a, S_{a+1}, \cdots, S_b\}$ that cannot be represented by one large simple connected subspace. Take Figure 3.1-(b) for example. The signature $Sig(r_1|r_2)$ is valid in two disconnect subspaces h_a and h_c. This is due to the fact that the intersection of two d-degree functions may partition the domain spaces into more than two subspaces. In order to reuse a signature in multiple disconnected subspaces, we modify the way of calculating signature to:

$$Sig(r_j|r_{j+1}) = Sig(H(H(r_j)|H(r_{j+1})|B_p|B_{p+1}|\cdots|B_q)) \tag{4.7}$$

where B_p,B_{p+1}, B_q represent the boundaries of a set of disconnected subspaces $\{S_p, S_{p+1}, \cdots, S_q\}$ that satisfies the following condition: for any S_i where $p \leq i \leq q$, $f_j(\cdot)$ is the direct predecessor of $f_{j+1}(\cdot)$ in the sorted list of functions corresponding to S_i. In other words, $\{S_p, S_{p+1}, \cdots, S_q\}$ is the set of all subspaces where $Sig(r_j|r_{j+1})$ is valid. Therefore, the order of two functions remains unchanged in several subspaces, still at most one signature is needed.

Once the signature mesh is established, the query processing and verification process is basically the same as for the other two types of polynomials. The only noticeable difference is how the client verifies that a signature is valid for its input. For each signature in $VO(q)$ received for query $q = (X, l, u)$, the client needs to check each of the boundaries in $\{B_p, B_{p+1}, \cdots, B_q\}$ signed with the signature. Unless at least one of the boundaries, say B_i, contains X, the signature is invalid and the query result should be rejected.

Note that checking the boundary condition (4) requires the client to perform the computation of up to four k-variable d-degree polynomial functions $f_a(X)$, $f_b(X)$, $f_{a-1}(X)$, and $f_{b+1}(X)$ for a range query. The cost of this computation is $O(kd)$ for range query and slightly larger for top-k and kNN queries. It is possible to further reduce the computation cost on the client side, for example, by outsourcing the computation of these polynomials to the server using verifiable computation techniques (e.g., Benabbas et al. (2011) and Parno et al. (2012))

4.5 Security and Performance Analysis

4.5.1 Security Analysis

We prove that the proposed technique can achieve our security goal for range queries as follows. We will consider only the case of range FQ. The analysis for top-k and kNN queries are similar and we will not discuss them separately for space constraint. Let $R(q) = \{r_a, r_{a+1}, \cdots, r_b\}$ be the query result for $q = (X, l, u)$, r_{a-1} and r_{b+1} the two boundary records. We first discuss the two cases when $R(q)$ is not complete:

Case 1: At lease one boundary record is forged. There are two ways for the adversary to do so. One is to create a fake record r'_{a-1} containing a function $f'_{a-1}(\cdot)$, where $f'_{a-1}(X) < l$. The adversary then inserts the fake record somewhere between r_a and r_b, and return $R(q) = \{r_i, \cdots, r_b\}$ with forged r'_{a-1} and r_{b+1}. Here r_i can be any record between r_a and r_b, and all the records from r_a to r_{i-1} are removed from $R(q)$. In order for the client to accept the manipulated query result, the adversary must forge a signature $Sig(r_{a-1}|r_i)$ using the digest of the fake boundary record. This is computationally infeasible without knowing the private key of the data owner. The other way to forge a boundary record is to report a record r_i in $R(q)$, where $a \leq i \leq b$, as a boundary record. The client will detect this flaw when checking the value of $f_i(X)$ during the verification process. It will discover that $l \leq f_i(X) \leq u$ and know that this is not a boundary record.

Case 2: Two records are not contiguous in the sorted list of the original database but is contiguous in $R(q)$. This happens when the adversary removes some record from $R(q)$. Suppose some record r_i ($a \leq i \leq b$) does not appear in $R(q)$. To avoid being detected, the adversary needs to replace two original contiguous signatures, $Sig(r_{i-1}|r_i)$ and $Sig(r_i|r_{i+1})$ with a forged signature, $Sig(r_{i-1}|r_{i+1})$. Again this is computationally infeasible without knowing the data owner's private key.

There are also two cases when $R(q)$ is not sound:

Case 1: Some record r_i in $R(q)$ is forged. This will be detected when a client verifies $Sig(r_i|r_{i+1})$ or $Sig(r_i|r_{i+1})$, the two signatures created with r_i and its immediate neighbours. The mechanisms used in signature creation basically make it computationally infeasible for one to create a false record without being detected.

Case 2: Some record r_i in $R(q)$ exists in the original database, but does not satisfy the query condition. The adversary can include such records into the query result by either reporting fake boundary records or inserting some records between two records that are contiguous in the original query result. Both of them are proved infeasible in the completeness cases.

4.5.2 Overhead Analysis

We now analyze the overhead introduced by the proposed technique on the data owner, the server, and the client side, respectively.

4.5.2.1 Data Owner Overhead

There are two costs for the data owner, computation cost incurred in constructing the signature mesh and communication cost in sending the mesh to the server. The key factor is the size of the signature mesh. In the next, we first analyze the total number of signatures that need to be created in the worst case.

Suppose there are n functions and each function has up to k variables with degree up to d. The total number of intersections of these functions is bounded by $O(n^2)$. Since these intersections are of order up to d, they can partition the k-dimension domain space of the functions into $O(n^{2d})$ subspaces. Recall that our techniques do not create a signature for each distinct subspace. Instead, only one signature $Sig(r_j|r_{j+1})$ is created for all subspaces where the two functions are adjacent and do not change their order in the sorted list. As such, for each pair of functions $f_i(\cdot)$ and $f_j(\cdot)$, at most two signatures will be created: $Sig(r_i|r_j)$ and $Sig(r_j|r_i)$. Even in the worst case where every two functions intersect with each other at certain point in the domain space, we will create at most $2 * C_2^{n+2}$ signatures. The size of the final signature mesh can be written as $O(C_2^{n+2}) = O((n+2)!/2n!)$, which is bounded by $O(n^2)$ and much smaller than $O(n^{2d})$. This result indicates that the upper bound of the overhead is a polynomial of the number of functions, regardless of the degree of functions and the number of variables. Note that this is the worst case, where the domain space is partitioned to the maximal possible number of subspaces by a set of functions, which is rare in real applications. The expected performance of the proposed technique in real application could be better than the worst case due to the limited function domain. This is supported by the simulation results and empirical study.

To efficiently find all intersections of a set of n polynomial functions, methods such as the Plane Sweep algorithm (Nievergelt and Preparata (1982)) can be used, which takes $O((n+m)\log n)$ where m is the actual number of intersections. After finding intersections, the data owner needs to partition the domain space into subspaces using these intersection surfaces. Constructing the aforementioned space partitioning tree requires $O(m^2)$. Assuming the computation cost of calculating a digital signature and computing a digest function is constant, we have the worst case computation cost of constructing the signature mesh being bounded by $O((n+m)\log n+m^2)$. This is a one-time cost for a data owner to prepare a database to be outsourced.

Since each data record consists of at most $kd+1$ coefficients(it may have some other non-coefficient attributes such as an id), the communication cost of sending the database to the server is $O(kdn)$. This is a fixed cost when outsourcing the database, with or without any authentication technique. Assuming that each signature takes a constant storage size (e.g., 512 bits), the total communication overhead between the data owner and the service provider (i.e., the cost of sending the signature mesh) is also subject to $O(n^2)$ in the worst case, which is rare. The actual cost is determined by the number of intersections of functions in the database.

4.5.2.2 Server Overhead

For the server, there is a one-time cost in receiving the original database and corresponding signature mesh from the data owner. After that, the main cost is constructing verification objects and sending them to clients. There are different ways to find the signatures corresponding to the subspace that containing the client-specified input value. Assuming binary search is used in traversing the space partitioning tree, we have the average cost of constructing $VO(q)$ as $O(l \log m^2)$ where m is the number of intersections and l the number of records in $R(q)$ that needs to be returned to the client.

4.5.2.3 Client Overhead

Each client needs to retrieve the public key from the data owner. This one-time cost is minimal. There two main overheads, the communication cost of receiving $R(q)$ and $VO(q)$ from the server, and the computation cost in the verification process. Apparently, the size of the $VO(q)$ grows linear w.r.t. the size of $R(q)$, since each signature is of constant length. In verifying $VO(q)$, the client needs to match the digest received from the server and the digest computed by himself using the records in $R(q)$, and also check the validity of signatures for each pair of contiguous records included in $R(q)$. In the worst case, a signature could be valid in $O(d)$ disconnected subspaces, and the client need to go through all these subspaces to decide if the signature is valid for the query. Let l be the number of records the client received, the process of verifying signatures will then cost $O(ld)$ time. Additionally, the client also needs to check the boundary conditions by computing up to four polynomials with k-variables and d-degrees. This can also be done in $O(kd)$ and the actual cost is negligible in practice. Therefore, the total computation overhead on the client can been seen as linear to the size of query result.

4.6 Performance Evaluation

We study the performance of the proposed techniques through security and overhead analysis, simulation and empirical evaluation on real world data. Since there is no existing work on function query authentication, the purpose of our simulation and empirical evaluation is to demonstrate the feasibility of the proposed signature mesh.

4.6.1 Security Analysis

We prove that the proposed technique can achieve our security goal for range queries as follows. We will consider only the case of range FQ. The analysis for top-k and kNN queries are similar and we will not discuss them separately for space constraint. Let $R(q) = \{r_a, r_{a+1}, \cdots, r_b\}$ be the query result for $q = (X, l, u)$, r_{a-1} and r_{b+1} the two boundary records. We first discuss the two cases when $R(q)$ is not complete:

Case 1: At lease one boundary record is forged. There are two ways for the adversary to do so. One is to create a fake record r'_{a-1} containing a function $f'_{a-1}(\cdot)$, where $f'_{a-1}(X) < l$. The adversary then inserts the fake record somewhere between r_a and r_b, and return $R(q) = \{r_i, \cdots, r_b\}$ with forged r'_{a-1} and r_{b+1}. Here r_i can be any record between r_a and r_b, and all the records from r_a to r_{i-1} are removed from $R(q)$. In order for the user to accept the manipulated query result, the adversary must forge a signature $Sig(r_{a-1}|r_i)$ using the digest of the fake boundary record. This is computationally infeasible without knowing the private key of the data owner. The other way to forge a boundary record is to report a record r_i in $R(q)$, where $a \leq i \leq b$, as a boundary record. The user will detect this flaw when checking the value of $f_i(X)$ during the verification process. It will discover that $l \leq f_i(X) \leq u$ and know that this is not a boundary record.

Case 2: Two records are not contiguous in the sorted list of the original database but is contiguous in $R(q)$. This happens when the adversary removes some record from $R(q)$. Suppose some record r_i ($a \leq i \leq b$) does not appear in $R(q)$. To avoid being detected, the adversary needs to replace two original contiguous signatures, $Sig(r_{i-1}|r_i)$ and $Sig(r_i|r_{i+1})$ with a forged signature,

$Sig(r_{i-1}|r_{i+1})$. Again this is computationally infeasible without knowing the data owner's private key.

There are also two cases when $R(q)$ is not sound:

Case 1: Some record r_i in $R(q)$ is forged. This will be detected when a user verifies $Sig(r_i|r_{i+1})$ or $Sig(r_i|r_{i+1})$, the two signatures created with r_i and its immediate neighbours. The mechanisms used in signature creation basically make it computationally infeasible for one to create a false record without being detected.

Case 2: Some record r_i in $R(q)$ exists in the original database, but does not satisfy the query condition. The adversary can include such records into the query result by either reporting fake boundary records or inserting some records between two records that are contiguous in the original query result. Both of them are proved infeasible in the completeness cases.

4.6.2 Overhead Analysis

We now analyze the overhead introduced by the proposed technique on the data owner, the server, and the user side, respectively.

4.6.2.1 Data Owner Overhead

There are two costs for the data owner, computation cost incurred in constructing the signature mesh and communication cost in sending the mesh to the server. The key factor is the size of the signature mesh. In the next, we first analyze the total number of signatures that need to be created in the worst case.

Suppose there are n functions and each function has up to k variables with degree up to d. The total number of intersections of these functions is bounded by $O(n^2)$. Since these intersections are of order up to d, they can partition the k-dimension domain space of the functions into $O(n^{2d})$ subspaces. Recall that our techniques do not create a signature for each distinct subspace. Instead, only one signature $Sig(r_j|r_{j+1})$ is created for all subspaces where the two functions are adjacent and do not change their order in the sorted list. As such, for each pair of functions $f_i(\cdot)$ and

$f_j(\cdot)$, at most two signatures will be created: $Sig(r_i|r_j)$ and $Sig(r_j|r_i)$. Even in the worst case where every two functions intersect with each other at certain point in the domain space, we will create at most $2 * C_2^{n+2}$ signatures. The size of the final signature mesh can be written as $O(C_2^{n+2}) = O((n+2)!/2n!)$, which is bounded by $O(n^2)$ and much smaller than $O(n^{2d})$. This result indicates that the upper bound of the overhead is a polynomial of the number of functions, regardless of the degree of functions and the number of variables. Note that this is the worst case, where the domain space is partitioned to the maximal possible number of subspaces by a set of functions, which is rare in real applications. The expected performance of the proposed technique in real application could be better than the worst case due to the limited function domain. This is supported by the simulation results and empirical study in the following sections.

To efficiently find all intersections of a set of n polynomial functions, methods such as the Plane Sweep algorithm Nievergelt and Preparata (1982) can be used, which takes $O((n+m)\log n)$ where m is the actual number of intersections. After finding intersections, the data owner needs to partition the domain space into subspaces using these intersection surfaces. Constructing the aforementioned space partitioning tree requires $O(m^2)$. Assuming the computation cost of calculating a digital signature and computing a digest function is constant, we have the worst case computation cost of constructing the signature mesh being bounded by $O((n+m)\log n + m^2)$. This is a one-time cost for a data owner to prepare a database to be outsourced.

Since each data record consists of at most $kd+1$ coefficients(it may have some other non-coefficient attributes such as an id), the communication cost of sending the database to the server is $O(kdn)$. This is a fixed cost when outsourcing the database, with or without any authentication technique. Assuming that each signature takes a constant storage size (e.g., 512 bits), the total communication overhead between the data owner and the service provider (i.e., the cost of sending the signature mesh) is also subject to $O(n^2)$ in the worst case, which is rare. The actual cost is determined by the number of intersections of functions in the database.

4.6.2.2 Server Overhead

For the server, there is a one-time cost in receiving the original database and corresponding signature mesh from the data owner. After that, the main cost is constructing verification objects and sending them to users. There are different ways to find the signatures corresponding to the subspace that containing the user-specified input value. Assuming binary search is used in traversing the space partitioning tree, we have the average cost of constructing $VO(q)$ as $O(l \log m^2)$ where m is the number of intersections and l the number of records in $R(q)$ that needs to be returned to the user.

4.6.2.3 User Overhead

Each user needs to retrieve the public key from the data owner. This one-time cost is minimal. There two main overheads, the communication cost of receiving $R(q)$ and $VO(q)$ from the server, and the computation cost in the verification process. Apparently, the size of the $VO(q)$ grows linear w.r.t. the size of $R(q)$, since each signature is of constant length. In verifying $VO(q)$, the client needs to match the digest received from the server and the digest computed by himself using the records in $R(q)$, and also check the validity of signatures for each pair of contiguous records included in $R(q)$. In the worst case, a signature could be valid in $O(d)$ disconnected subspaces, and the client need to go through all these subspaces to decide if the signature is valid for the query. Let l be the number of records the client received, the process of verifying signatures will then cost $O(ld)$ time. Additionally, the client also needs to check the boundary conditions by computing up to four polynomials with k-variables and d-degrees. This can also be done in $O(kd)$ and the actual cost is negligible in practice. Therefore, the total computation overhead on the client can been seen as linear to the size of query result.

4.6.3 Simulation

We have implemented a detailed simulator that allows us to evaluate the performance of the proposed technique in various aspects. Our simulator incorporates a data generator that can be configured to generate various kinds of polynomial function randomly. The input to this generator includes the number of functions n, the maximum number of variables k, and the maximum number of degree d. When generating the functions, we assign a unique ID and using random real numbers as its coefficients. The domain of each function is set to $[-X, X]^k$ where X is a random integer no greater than 10. Such a domain is large enough for all the applications mentioned in introduction. Given the values of k and d, the actual numbers of variables in each multivariate function and degree in each high degree polynomial function is generated to follow a normal distribution with standard deviation of $k/2$ and $d/2$, respectively, and which are rounded to the nearest integer. We use SHA-1 for digest function and RSA (256 bits) for digital signature. More detailed settings of our simulation are given in table 4.1. We are interested in three performance metrics, including the one-time data owner overhead, server overhead per query, and client overhead per query. Our simulation platform is a Windows server with Intel Xeon 64-bit 8-core CPU running on 2.93GHz and 32GB RAM. In the following subsections, we report and explain the performance results in the following subsections.

Table 4.1: Experiment Settings

Parameter	**Range**	**Default**
Number of data records	10,000 - 100,000	50,000
Average number of variables	1 - 5	3
Average degree of variables	1 - 5	3
Size of query result	0 - 500	250

4.6.3.1 Data Owner Overhead

We first study the effect of the number of functions on the size of signature mesh. We adjust the number of functions from 10,000 to 100,000 while using the default value for the other parameters. The actual size of the constructed signature mesh is showed in Figure 4.6, which reflects the one-time communication overhead between the data owner and the server. The size in general grows linearly with the number of functions in our experiments. This can be explained by the fact that in a limited domain range, it is not likely that every pair of functions will intersect, thus the number of signatures needed and the number of subspaces partitioned increases much slower than in the worst case.

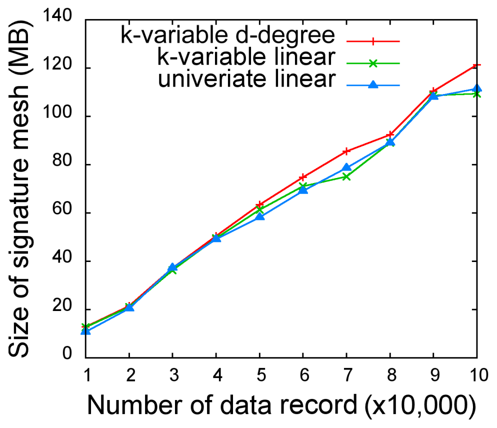
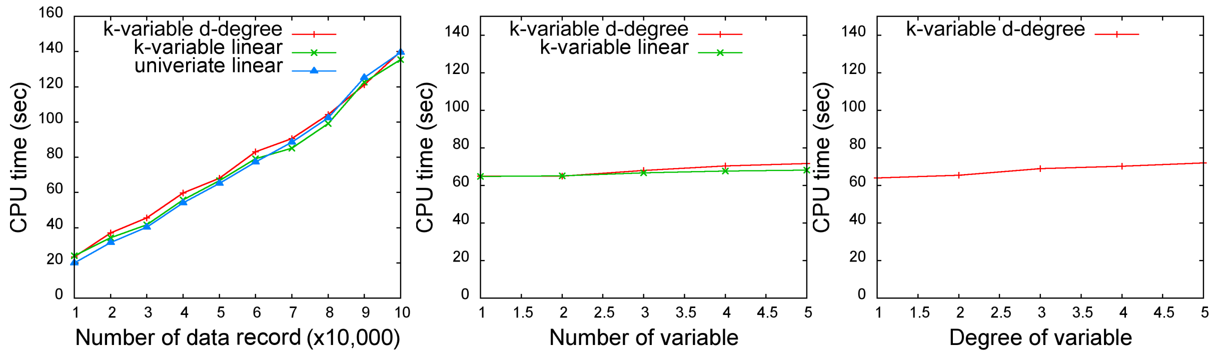


Figure 4.6: Communication overhead of signature mesh

The one-time computation cost of constructing the signature mesh is evaluated with respect to the number of data records, the average number of variables, and the average degree of each variable. The simulation result, in terms of CPU time (of the data owner), is showed in Figure 4.7. The construction time is not sensitive to the number of variables for multivariate functions, or the number of degrees. The number of functions has relatively stronger effect on the construction time for all the three types of polynomials in general. The result is not surprising, as we have known through the analysis. Nevertheless, since the construction and transmission of the signature mesh occur only once when the database is outsourced, it is just a one-time cost and will not adversely impact the performance of the proposed technique in a long term.

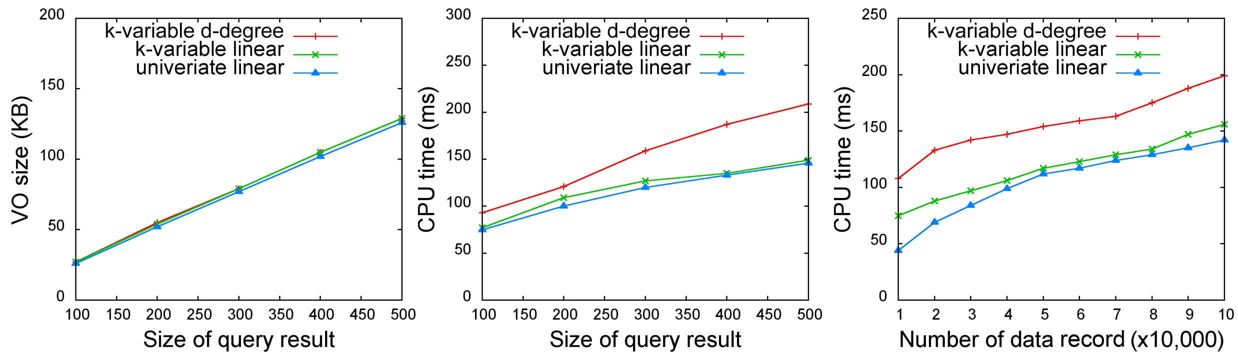
4.6.3.2 Server Overhead

Once the signature mesh is constructed and database outsourced, the major overhead of the authentication process is the communication and computation cost of generating and sending the verification object. We adjust the number of records returned to the user (size of $R(q)$) from 0 to



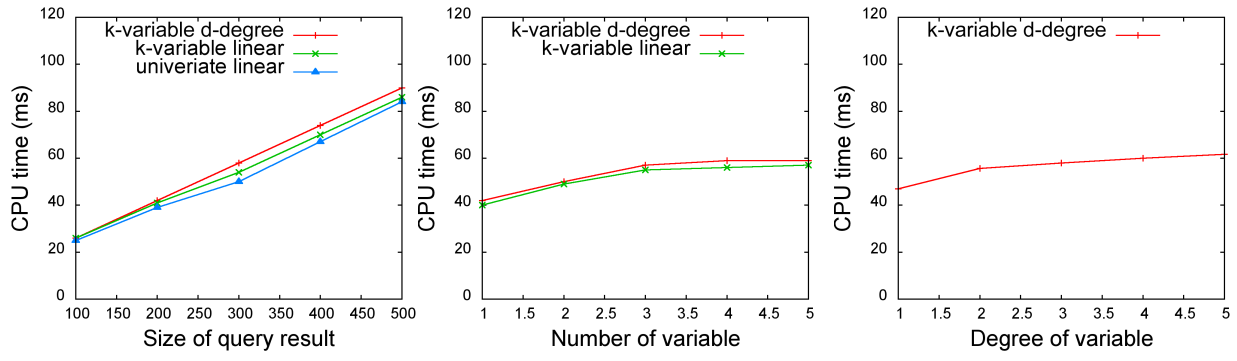
(a) (b) (c)

Figure 4.7: Computation cost of constructing signature mesh



(a) (b) (c)

Figure 4.8: Communication and computation cost of constructing verification object



(a) (b) (c)

Figure 4.9: Computation cost of the verification process

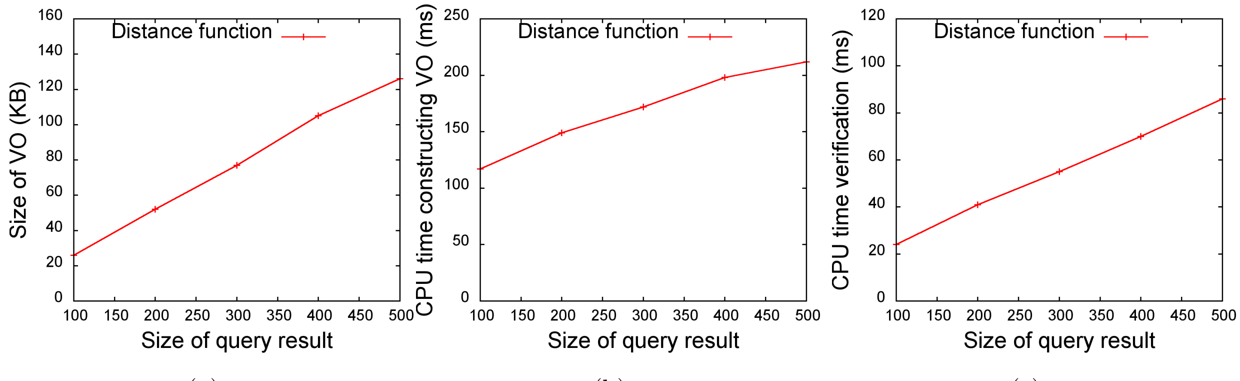


Figure 4.10: Performance on Gowalla dataset

1000 and report the corresponding size of $VO(q)$. Note that even when 0 records satisfies the query condition, a VO still need to be return to the use as proof. The simulation result in Figure 4.8 (a) illustrates that the size of $VO(q)$ increases linearly to the size of $R(q)$ as expected. Note that the actual size of $VO(q)$ in real applications also depends on what digest function/size of digital signature is used.

The computation cost of generating $VO(q)$ is determined by the size of query result $R(q)$ as well as the size of the signature mesh, which is mainly impacted by the total number of data records as showed in the above simulations. We adjust these two parameters same as before. Figure 4.8-(b) and (c) shows the average time cost of constructing $VO(q)$ for 100 function queries with random inputs/query conditions. The figure confirms that the time of constructing $VO(q)$ grows linearly to the size of $R(q)$, but is less sensitive to the number of data records. This is due to the fact that the space partitioning tree we use for subspace indexing prevents the server from having to perform a linear search through all the subspaces.

4.6.3.3 Client Overhead

We now evaluate the computation cost incurred during the verification process in terms of the following three factors: the size of query result $R(q)$, the average number of variables, and the average degree of variables. The results are showed in Figure 4.9. Our analysis indicates that the number of variables and degree has little impact on the verification time, since our technique needs to verify only the boundary records. This is confirmed in this study. In general, the cost is linear to the size of $R(q)$ since the main cost of verification process comes from verifying the signature of each two contiguous records. Nevertheless, the absolute cost of verification is very small as showed in the figures since computing simple polynomial functions and verifying digital signatures can all be performed efficiently(especially on multi-core CPU platforms, since the data dependency is very low in these tasks).

4.6.4 Empirical Study

We use the Gowalla moving object dataset (Cho et al. (2011)) to evaluate performance of our technique in real world applications. The dataset contains 196,591 locations where each location represents a Point of Interest (POI) that a mobile client has checked-in. To demonstrate the capability of the proposed technique, we treat each record r_i as a distance function $d_i(x, y) = (x_i - x)^2 + (y_i - y)^2$ where (x_i, y_i) is the coordinate of r_i. The function computes the (square of) distance between the POI and a user-specified location (x, y). The input domain is set to $x \in [53, 54]$ and $y \in [-3, -2]$ which formulates a rectangle area that covers all the POIs. These functions are then outsourced to a server along with the authentication structure.

We issue function queries over the distance functions using randomly selected query point to find the POIs among the K nearest neighbour of the query point, which is a realistic example of function query authentication. We adjust the size of query result and study its impact on computation cost on server and client side, and the communication overhead in terms of VO size. For each test point the average cost of 100 queries is reported and the results is plotted in Figure 4.10. In general, the authentication overhead grows linearly with respect to the number of records returned to users as expected. In other words, the cost per record is basically a constant since each pair of contiguous records in the query result is accompanied by a digital signature.

4.7 Summary of the Chapter

We consider the problem of outsourcing a database where each record is interpreted as a mathematical function. A third party manages this database and answers function queries over the functions. A function query retrieves the set of functions whose outputs with user-supplied function input satisfy certain query condition (e.g., being the top-k). We are interested in providing a mechanism that allows users to verify if the query results they receive are sound and complete. The challenge of enabling such verification comes from the fact that the input to the functions is known only when a query is issued, so the data owner cannot pre-compute the function outputs and then apply existing techniques to build a MH-tree or a signature chain on top of these outputs for authentication.

Two functions have the same output when they intersect. So for any input within a domain where they do not intersect, the output of one function is always no less than that of the other function. In light of this, we develop a three-step general solution for efficient verification of function query results: 1) Partition the input domain into a number of subdomains which are defined by the intersections of the functions; 2) Sort the functions based on their outputs in each of the subdomains; 3) Create a signature chain for each sorted function list. We show that this general solution works for various types of function queries over different kinds of functions, including univariate linear function, multivariate linear function, and multivariate high degree function. We prove that without knowing the data owner's private key, it is computationally infeasible for an adversary to forge a query result without being detected. Our extensive performance evaluation shows the proposed techniques are practical and can be used in real-world applications.

CHAPTER 5. IMPROVEMENT QUERIES

5.1 Motivation

As an analytic query, top-k query is often used in applications like e-commerce for users to find objects (e.g., products) that best match their preference. Here a user's preference is computed by a utility function that gives a "score" for each object, and a top-k query retrieves the k objects with the highest 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. We refer to such an adjustment as an *improvement strategy.* Our research is interested in finding the improvement strategies for objects of interest under some cost constraints. 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 (Viner (1932); Binger et al. (1998)). 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 (Vlachou et al. (2011)), maximal rank query (Mouratidis et al. (2015)), and reverse k-ranks query (Zhang et al. (2014b)) 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 chapter.

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. In function queries, each object is interpreted as a function and a top-k query is treated 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. As such, 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.

5.2 Problem Formulation

5.2.1 Definitions

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

Cameras

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

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

ID	*resolution (Megapixel)*	*storage (GB)*	*price ($)*
p'_1	15	4	200
p_2	12	4	340
$\cdots$	$\cdots$	$\cdots$	$\cdots$

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*	$k = 1$
$\cdots$	$\cdots$	$\cdots$

Figure 5.1: Example of improvement strategy for cameras

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 (Viner (1932); Binger et al. (1998); Anderson (2009)) 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 Chapter 5.3 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.

5.2.2 Basic Idea

In the proposed function query, each object is interpreted as a function and each top-k query is treated as a function input. This is different from existing top-k queries where queries are considered as utility functions and objects as their input. We show that by interpreting objects as functions, the cost of processing IQs can be significantly reduced. To make it easy to follow,w e use the most common *linear* utility functions (e.g., Chaudhuri and Gravano (1999); Chang et al. (2000); Hristidis et al. (2001); Zou and Chen (2008)) as an example.

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:

$$f_i(q) = \textstyle\sum_{j=1}^{d} q^{(j)} p_i^{(j)} \tag{5.1}$$

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.

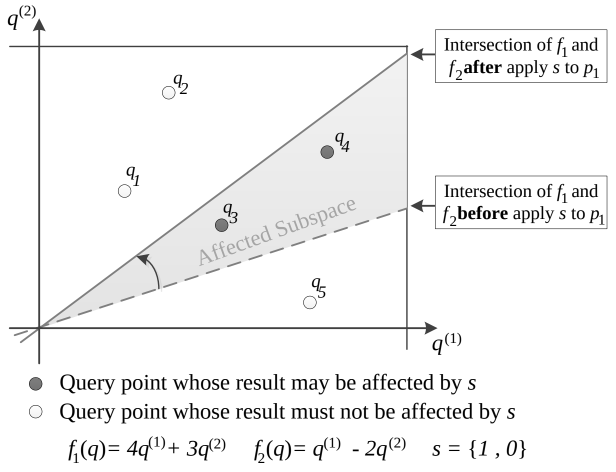
The intersection of two functions f_i and f_j creates a *hyperplane* in the d-dimensional domain space. The intersection partitions the domain into two *subdomains*, namely *above* and *below*. For any input q falls in the above subdomain, we have $f_i(q) \geq f_j(q)$, and for any input q in the below 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 5.2, 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 and we refer readers to De Berg et al. (2000); Preparata and Shamos (2012) for proof details. These facts suggest an efficient way to


$q^{(2)}$
$q^{(1)}$
Intersection of f_1 and f_2 **after** apply s to p_1
Intersection of f_1 and f_2 **before** apply s to p_1
q_1
q_2
q_3
q_4
q_5
Affected Subspace
● Query point whose result may be affected by s
○ Query point whose result must not be affected by s
$f_1(q) = 4q^{(1)} + 3q^{(2)}$ $f_2(q) = q^{(1)} - 2q^{(2)}$ $s = \{1, 0\}$

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 5.2: An improvement strategy affects subdomain boundaries and query results

evaluate a given improvement strategy s. First, we apply s to p_i and 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.

5.3 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 (Guttman (1984)) or X-tree (Berchtold et al. (2001)). 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 Chapter 5.4.

5.3.1 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) (Vlachou et al. (2011)) 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:

$$\sum_{j=1}^{d} q^{(j)}(p_i^{(j)} - p_l^{(j)}) = 0 \tag{5.2}$$

Equation 5.3 represents the new intersection hyperplane after some improvement strategy s is applied to p_i.

$$\sum_{j=1}^{d} q^{(j)}(p_i^{(j)} + s_j - p_l^{(j)}) = 0 \tag{5.3}$$

The area bounded between the old and new intersection hyperplanes represented by Equation 5.2 and 5.3 formulates a subspace (e.g., the shadow area showed in Figure 5.2) 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.

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 (Nievergelt and Preparata (1982)). We then partition the function domain into subdomains gradually, by considering function intersections one at a time.

Algorithm 7 *FindSubdomains*(I, Q)

```
$d \leftarrow newSubdomain()$
$Subdomains.add(d)$
**for all** $q \in Q$ **do**
    $q.subdomain \leftarrow d$
**end for**
**for all** $I_i \in I$ **do**
    **for all** Subdomain $d \in Subdomains$ such that $d$ overlaps $I_i$ **do**
        $d_{above} \leftarrow newSubdomain()$
        $d_{above}.boundaries.add(I_i, above)$
        $d_{below} \leftarrow newSubdomain()$
        $d_{below}.boundaries.add(I_i, below)$
        **for all** $q$ falls in $d$ **do**
            **if** $q$ falls above $I_i$ **then**
                $q.subdomain \leftarrow d_{above}$
            **else**
                $q.subdomain \leftarrow d_{below}$
            **end if**
        **end for**
        **if** $d_{above}$ contains query **then**
            $Subdomains.add(d_{above})$
        **end if**
        **if** $d_{below}$ contains query **then**
            $Subdomains.add(d_{below})$
        **end if**
    **end for**
**end for**
Return $Subdomains$
```

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 intersection. 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 7.

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 5.2 and 5.3. It is determined by two boundary conditions:

$$\textstyle\sum_{j=1}^{d} q^{(j)}(p_i^{(j)} - p_l^{(j)}) \geq 0 \tag{5.4}$$

$$\textstyle\sum_{j=1}^{d} q^{(j)}(p_i^{(j)} + s_j - p_l^{(j)}) < 0 \tag{5.5}$$

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 8.

Algorithm 8 *EfficientStrategyEvaluation*(p_i, s)

```
1: $H(p_i + s) \leftarrow |TP(p_i)|$
2: for all $f_l \in D$ intersects $f_i$ and $f_l \neq f_i$ 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 $q_j \notin TP(p_i)$ and $q_j \in TP(p_i + s)$ then
11:        $H(p_i + v) + +$;
12:      else if $q_j \in TP(p_i)$ and $q_j \notin TP(p_i + s)$ then
13:        $H(p_i + v) - -$;
14:      end if
15:    end for
16:  end for
17: end for
18: Return $H(p_i + s)$
```

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.

5.3.2 Improvement Strategy Searching

5.3.2.1 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:

$$f'_i(q_j) < f_{j,k}(q_j) \tag{5.6}$$

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:

$$\text{minimize} \quad Cost_{p_i}(s) \tag{5.7}$$

$$\text{subject to} \quad \sum_{j=1}^{m} x_j \geq \tau \tag{5.8}$$

$$f'_i(q_j) < f_{j,k}(q_j) + (1 - x_j)C \qquad \forall j \in [1, m] \tag{5.9}$$

$$x_j \in \{0, 1\} \qquad \forall j \in [1, m] \tag{5.10}$$

where C denotes a very large number that exceeds the highest score of all objects. Constraint 5.8 guarantees that the improved object hits at least τ queries, while Constraint 5.9 ensures that Equation 5.6 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 (Wolsey and Nemhauser (2014)), 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:

$$u_i(p) = w_{i1} * p^{(1)} + w_{i2} * p^{(1)} + ... + w_{im} * p^{(m)} \tag{5.11}$$

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:

$$Cost_{p_0}(s) = s_1 + s_2 + ... + s_m \tag{5.12}$$

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 9) 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 9 $MinCostIQ(p_i, \tau, Cost_{p_i})$

```
1:  $p'_i \leftarrow p_i$
2:  while $H(p'_i) < \tau$ do
3:    $S \leftarrow \emptyset$
4:    for each query $q_j \in Q$ and $\notin TP(p'_i)$ do
5:      $s_j \leftarrow \arg\min Cost_{p'_i}(s)$ such that $q_j \in TP(p'_i + s)$
6:      Compute $H(p'_i + s_j)$
7:      $S.add(s_j)$
8:    end for
9:    Find $s \in S$ with minimal $Cost_{p'_i}(s)/H(p'_i + s)$
10:   if $H(p'_i + s) \leq \tau$ then
11:     $p'_i = p'_i + s$
12:   else
13:     Return $s \in S$ with minimal $Cost_{p'_i}(s)$ and $H(p'_i + s) \geq \tau$
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:

$$\text{minimize } \; Cost_{p_i}(s) \tag{5.13}$$

$$\text{subject to } \; f'_i(q_j) < f_{j,k}(q_j) \tag{5.14}$$

which can be efficiently solved using standard math tools like the ones discussed in (Khachiyan (1980)). Note that the cost of s_j is equivalent to the minimal distance between any intersection hyperplane of f_i and the query point q, where the distance is measured by the cost function.

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 9 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 (Khachiyan (1980)) to solve the above optimization problem. However, due to the intractability of the problem, this algorithm is only feasible for very small datasets.

5.3.2.2 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.

$$\text{maximize } H(p_i + s) \tag{5.15}$$

$$\text{subject to } Cost_{p_i}(s) \leq \beta \tag{5.16}$$

$$f_i'(q_j) < f_{j,k}(q_j) + (1 - x_j)C \qquad \forall j \in [1, m] \tag{5.17}$$

$$x_j \in \{0, 1\} \qquad \forall j \in [1, m] \tag{5.18}$$

The target function computes the hit number of the improved object, while Constraint 5.16 corresponds to the limited budget. The meaning of Constraint 5.17 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 $MaxHit\ (p_i, \beta, Cost_{p_i})$ 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 9 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 10 $\mathit{MaxHitIQ}(p_i, \beta, Cost_{p_i})$

```
1:  $p'_i \leftarrow p_i$
2:  $s* \leftarrow 0$
3:  while $Cost_(p_i)(s*) < \beta$ do
4:    $S \leftarrow \emptyset$
5:    for each query $q_j \in Q$ and $\notin TP(p'_i)$ do
6:      $s_j \leftarrow \arg\min Cost_{p'_i}(s)$ such that $q_j \in TP(p'_i + s)$
7:      Compute $H(p'_i + s_j)$; $S.add(s_j)$
8:    end for
9:    Find $s \in S$ with minimal $Costp'_i(s)/H(p'_i + s)$
10:   if $Cost_(p_i)(s*) + Cost_(p_i)(s) \leq \beta$ then
11:     $s* += s$
12:   else
13:     for each $s \in S$, sorted by cost do
14:       if $Cost_(p_i)(s*) + Cost_(p_i)(s) \leq \beta$ 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 10. For convenience, we will refer to Algorithms 9 and 10 together as the **Efficient-IQ** algorithm. Similarly, we also provide the exhaustive search option in our implementation.

5.3.3 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 7 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 7. 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 7 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 7 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.

5.4 Extensions

5.4.1 Improving Multiple Target Objects

So far we have considered improving a single object. We now show how to 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 Chapter 5.3.2 to handle the combinatorial improvement strategy searching problems. To search for the combinatorial minimal cost improvement strategy, we can modify Algorithm 9 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 10 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.

5.4.2 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 5.1), where w_1 and w_2 are user-specified weights.

$$u(Car \quad c) = \sqrt{w_1 * c.Price} + w_2 \frac{c.Capacity}{c.MPG} \tag{5.19}$$

Table 5.1: Example of a dataset with complex untility 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	$\sqrt{8000w_1} + w_2\frac{2}{35}$

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.

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 Chapter 5.3 for efficient processing of IQs. Consider an example of top-k queries with polynomial utility function, applied on a 4-dimensional dataset D:

$$u(p) = w_1(p^{(1)})^3 + w_2(p^{(2)} * p^{(3)}) + w_3(p^{(4)})^2 \tag{5.20}$$

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

$$u^*(p) = w_1 p^{(5)} + w_2 p^{(6)} + w_3 p^{(7)} \tag{5.21}$$

[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 (Schläfli (1901)). While for some high-degree functions, the number can be $O(2^n)$.

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:

$$u(p) = \sqrt{(w_1 - p^{(1)})^2 + (w_2 - p^{(2)})^2} \tag{5.22}$$

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

$$u^*(p) = (w_1 - p^{(1)})^2 + (w_2 - p^{(2)})^2 \tag{5.23}$$

$$u^*(p) = (w_1^2 + w_2^2) - 2w_1 p^{(1)} - 2w_2 p^{(2)} \tag{5.24}$$

$$+ p^{(3)} + p^{(4)} \tag{5.25}$$

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 remains the same.

5.4.3 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 5.1), some users may express their preference as a different utility function:

$$v(Car \quad c) = \frac{c.MPG}{w_1 * c.Price} + w_2(c.Capacity)^2 \tag{5.26}$$

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 5.19 and 5.26 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 5.19 and 5.26 by adding them up:

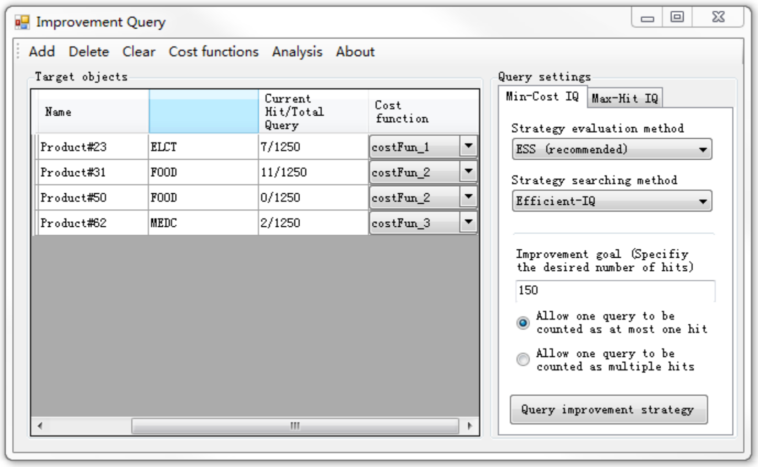
$$\begin{aligned} G(Car \quad c) =& u(Car \quad c) + v(Car \quad c) && (5.27)\\ =& \sqrt{w_1 * c.Price} + w_2 \frac{c.Capacity}{c.MPG} && (5.28)\\ +& \frac{c.MPG}{w_3 * c.Price} + w_4 (c.Capacity)^2 && (5.29) \end{aligned}$$

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 5.19 as utility function, it must set w_3, w_4 to 0. While for queries with function 5.26, 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.

5.5 Implementation and Evaluation

5.5.1 System Implementation

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 5.3. 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



Name		Current Hit/Total Query	Cost function
Product#23	ELCT	7/1250	costFun_1
Product#31	FOOD	11/1250	costFun_2
Product#50	FOOD	0/1250	costFun_2
Product#62	MEDC	2/1250	costFun_3

Strategy evaluation method: ESS (recommended)

Strategy searching method: Efficient-IQ

Improvement goal (Specifiy the desired number of hits): 150

[✓ Allow one query to be counted as at most one hit]

Allow one query to be counted as multiple hits

Figure 5.3: Graphic User Interface for Improvement Query

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.

5.5.2 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 (Borzsony et al. (2001)). 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 (FuelEconomy.gov (2016)) contains 37051 vehicle models with attributes including year, weight, horse power, mileage per gallon (MPG), and annual cost. HOUSE is extracted from (Ruggles et al. (2015)), 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 (Vlachou et al. (2011)). 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 5.2.

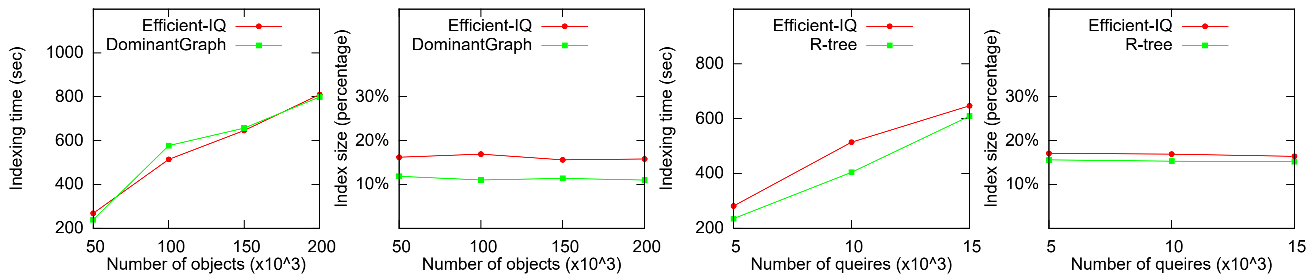
Table 5.2: Experiment Settings

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

5.5.3 Experiment Results

5.5.3.1 Data Indexing

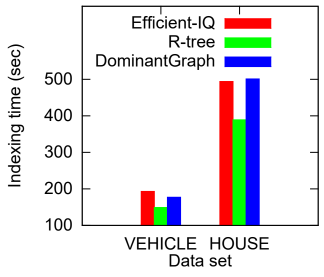
We first evaluate the indexing cost of the proposed techniques, which involve the cost of building an R-tree over the query points and grouping them by subdomains. To better understand the scale of this cost, we compare indexing structure size (showed as percentage to the original dataset) and the total 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**) (Zou and Chen (2008)) for the objects, which is the state-of-the-art indexing technique for top-k query with linear utility functions.



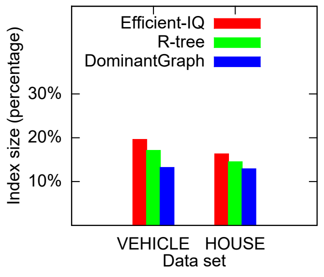
(a) Indexing time (b) Index size (c) Indexing time (d) Index size

Figure 5.4: Index Scalability

We adjust the number of objects and report the corresponding indexing time and size of the proposed technique and DominantGraph (Figure 5.4-(a) and (b)). 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. We also evaluated the indexing cost of the proposed technique on the two real-world datasets and the results are plotted in Figure 5.5.



(a) Indexing time



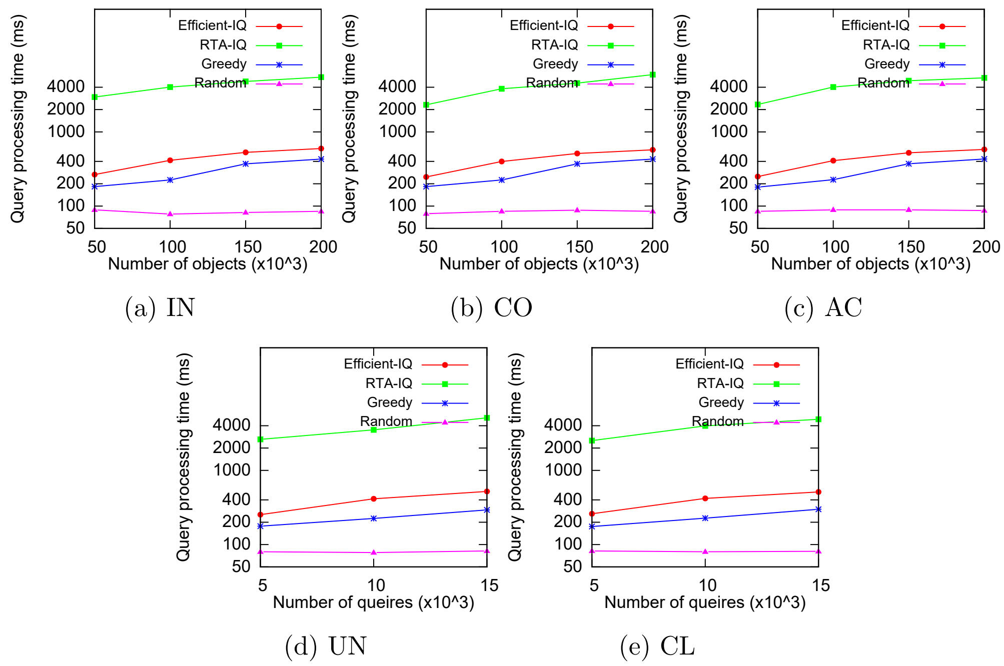
(b) Index size

Figure 5.5: Indexing cost on real world datasets

We then adjust the number of queries and compare the proposed technique with building only an R-tree (Figure 5.4-(c) and (d)). 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.

5.5.3.2 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



(a) IN (b) CO (c) AC

(d) UN (e) CL

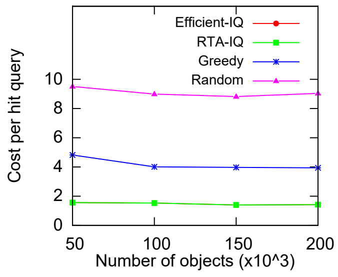
Figure 5.6: Query processing time on different sythentic datasets

cost function for all objects:

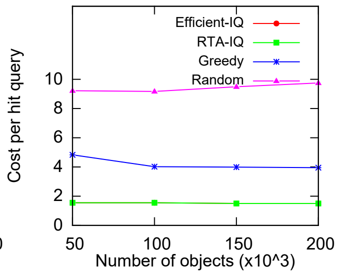
$$Cost(s) = \sqrt{\sum_{i=1}^{d} s_i^2} \tag{5.30}$$

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 5.6-5.8. For 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 5.2. For each real-world dataset, we use a randomly generate query set that is one third of its size.

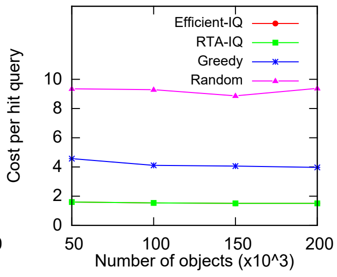
It is not surprising that Random is the fastest scheme in processing IQs, but it also yields the worst improvement strategy quality. The simple greedy algorithm has better strategy quality than Random, but is still very poor when compared with the proposed techniques. The Efficient-IQ achieves both good 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



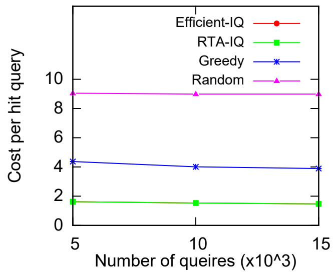
(a) IN



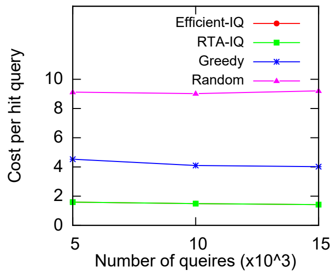
(b) CO



(c) AC



(d) UN

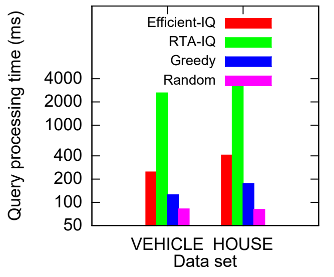


(e) CL

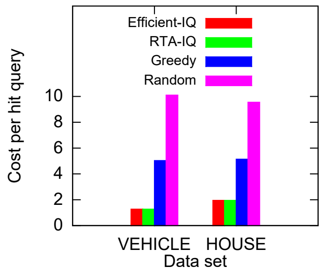
Figure 5.7: Cost per hit query on different sythentic datasets

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 5.9)



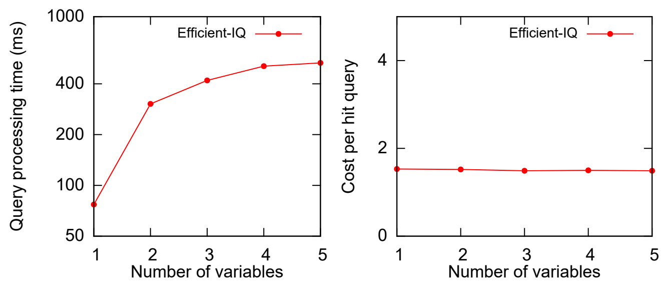
(a) Query processing time



(b) Cost per hit query

Figure 5.8: Query processing cost on the real-world datasets

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.



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

Figure 5.9: Scalability to the number of variables in functions

5.6 Summary of the Chapter

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 chapter, 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 geometrical 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.

CHAPTER 6. CONCLUDING REMARKS

The notion of *Function Query* (FQ) is a powerful extension to existing database query languages. It can be applied on a database that is originally a set of functions, e.g., representing some data that is continuous in nature. While this is obvious, FQ can also be applied, which we believe is less apparent but inspiring, on a database of discrete values. By interpreting each record as a function, FQ supports analysis-based information retrieval like existing top-k and scalar-product queries. However, unlike these existing queries, FQ supports more complex functions and a variety of output conditions.

We have addressed the challenges of enabling efficient FQ execution. Our key observation is, a set of functions can be sorted based on their outputs in the input subdomains partitioned by the intersections of these functions. This observation alone, however, is not sufficient to develop a good solution. Finding the exact boundaries and sorting the functions in each subdomain can be computation-intensive. The problem is even more complicate when having to deal with a very large number of subdomains, which happens when the functions involve non-linear terms or are non-polynomial. To circumvent these problems, we proposed a novel data structure called *Intersection-tree* (I-tree). I-tree indexes the subdomains created by function intersections and allows one to sort the functions for each subdomain, without having to computing subdomain boundaries. With I-tree in place, we proposed to convert complex functions into multivariate linear polynomial functions through variable replacement. We show that this strategy works for all polynomial functions and non-polynomial functions that conform certain form. Moreover, it becomes possible to handle multiple function definitions on the same database with a single I-tree. While I-tree is mainly developed for FQs, we show that it can also be used to support the execution of some other well-known analytic queries, including reverse top-k query, maximum rank query and global immutable region. In our research, we have integrated FQs into a database system as a query primitive. We evaluated the proposed techniques through prototyping and experiments over synthetic data and real-world data, and our techniques exhibit excellent performance in our extensive evaluation.

In addition to FQ execution, we have consider the problem of outsourcing a database where each record is interpreted as a mathematical function. A third party manages this database and answers function queries over the functions. A function query retrieves the set of functions whose outputs with user-supplied function input satisfy certain query condition (e.g., being the top k). We are interested in providing a mechanism that allows users to verify if the query results they receive are sound and complete. The challenge of enabling such verification comes from the fact that the input to the functions is known only when a query is issued, so the data owner cannot pre-compute the function outputs and then apply existing techniques to build a MH-tree or a signature chain on top of these outputs for authentication.

Two functions have the same output when they intersect. So for any input within a domain where they do not intersect, the output of one function is always no less than that of the other function. In light of this, we develop a three-step general solution for efficient verification of function query results: 1) Partition the input domain into a number of subdomains which are defined by the intersections of the functions; 2) Sort the functions based on their outputs in each of the subdomains; 3) Create a signature chain for each sorted function list. We show that this general solution works for various types of function queries over different kinds of functions, including univariate linear function, multivariate linear function, and multivariate high degree function. We prove that without knowing the data owner's private key, it is computationally infeasible for an adversary to forge a query result without being detected. Our extensive performance evaluation shows the proposed techniques are practical and can be used in real-world applications.

BIBLIOGRAPHY

Anagnostopoulos, C. and Triantafillou, P. (2017). Efficient scalable accurate regression queries in in-dbms analytics. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 559–570. IEEE.

Anderson, J. (2009). Determining manufacturing costs. *CEP*, pages 27–31.

Balke, W.-T. and Kießling, W. (2000). Optimizing multi-feature queries for image databases. *VLDB,(Sep 2000)*, pages 10–14.

Benabbas, S., Gennaro, R., and Vahlis, Y. (2011). Verifiable delegation of computation over large datasets. In *Advances in Cryptology – CRYPTO'11*, pages 111–131. Springer.

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

Binger, B. R. et al. (1998). *Microeconomics with calculus.* Number 338.501 B613 1998. Addison-Wesley.

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

Brent, R. P. (2013). *Algorithms for minimization without derivatives.* Courier Corporation.

Burrows, J. H. (1995). Secure hash standard. Technical report, DTIC Document.

Chang, Y.-C., Bergman, L., Castelli, V., Li, C.-S., Lo, M.-L., and Smith, J. R. (2000). The onion technique: indexing for linear optimization queries. In *SIGMOD'2000*, pages 391–402.

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

Chauduri, S. and Gravano, L. (1999). Evaluating Top-k Selection Queries. In *Proc. of VLDB'99*, pages 397–410.

Cheema, M. A., Shen, Z., Lin, X., and Zhang, W. (2014). A unified framework for efficiently processing ranking related queries. In *EDBT*, pages 427–438.

Chen, Q., Hu, H., and Xu, J. (2013). Authenticating top-k queries in location-based services with confidentiality. *Proceedings of VLDB'13*, 7(1).

Cheng, W., Pang, H., and Tan, K.-L. (2006). Authenticating multi-dimensional query results in data publishing. In *Data and Applications Security XX*, pages 60–73. Springer.

Cho, E., Myers, S. A., and Leskovec, J. (2011). Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1082–1090. ACM.

Choi, S., Lim, H.-S., and Bertino, E. (2012). Authenticated top-k aggregation in distributed and outsourced databases. In *Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Confernece on Social Computing (SocialCom)*, pages 779–788. IEEE.

Das, G., Gunopulos, D., Koudas, N., and Sarkas, N. (2007). Ad-hoc top-k query answering for data streams. In *VLDB'2007*, pages 183–194. VLDB Endowment.

Das, G., Gunopulos, D., Koudas, N., and Tsirogiannis, D. (2006). Answering top-k queries using views. In *VLDB'2006*, pages 451–462.

De Berg, M., Van Kreveld, M., Overmars, M., and Schwarzkopf, O. C. (2000). *Computational geometry*. Springer.

Devanbu, P., Gertz, M., Martel, C., and Stubblebine, S. (2003). Authentic data publication over the internet. *Journal of Computer Security*, 11(3):291–314.

Fagin, R., Lotem, A., and Naor, M. (2003). Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656.

FuelEconomy.gov (2016). Fueleconomy.gov vehicle data. http://www.fueleconomy.gov/feg/ws/index.shtml. Updated: Tuesday April 12 2016.

Gao, Y., Liu, Q., Chen, G., Zheng, B., and Zhou, L. (2015). Answering why-not questions on reverse top-k queries. *Proceedings of the VLDB Endowment*, 8(7):738–749.

Guo, T., Papaioannou, T. G., and Aberer, K. (2013). Model-view sensor data management in the cloud. In *Big Data, 2013 IEEE International Conference on*, pages 282–290. IEEE.

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

He, Z. and Lo, E. (2014). Answering why-not questions on top-k queries. *IEEE Transactions on Knowledge and Data Engineering*, 26(6):1300–1315.

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

Hu, H., Xu, J., Chen, Q., and Yang, Z. (2012). Authenticating location-based services without compromising location privacy. In *Proceedings of SIGMOD'12*, pages 301–312. ACM.

Katsis, Y., Freund, Y., and Papakonstantinou, Y. (2015). Combining databases and signal processing in plato. In *CIDR*.

KDD (2004). Kdd cup'04 test data for the quantum physics task. http://osmot.cs.cornell.edu/kddcup/datasets.html.

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

Khan, A., Yanki, P., Dimcheva, B., and Kossmann, D. (2014). Towards indexing functions: answering scalar product queries. In *SIGMOD'2014*, pages 241–252.

Li, F., Hadjieleftheriou, M., Kollios, G., and Reyzin, L. (2006). Dynamic authenticated index structures for outsourced databases. In *Proceedings of SIGMOD'06*, pages 121–132. ACM.

Merkle, R. C. (1990). A certified digital signature. In *Advances in Cryptology – CRYPTO'89 Proceedings*, pages 218–238. Springer.

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

Nepal, S. and Ramakrishna, M. (1999). Query processing issues in image (multimedia) databases. In *ICDE'1999*, pages 22–29. IEEE.

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

Pang, H., Jain, A., Ramamritham, K., and Tan, K.-L. (2005). Verifying completeness of relational query results in data publishing. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM.

Pang, H., Zhang, J., and Mouratidis, K. (2009). Scalable verification for outsourced dynamic databases. *Proceedings of VLDB'09*, 2(1):802–813.

Parno, B., Raykova, M., and Vaikuntanathan, V. (2012). How to delegate and verify in public: Verifiable computation from attribute-based encryption. In *Theory of Cryptography*, pages 422–439. Springer.

Peng, P. and Wong, R. C.-W. (2015). k-hit query: Top-k query with probabilistic utility function. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 577–592. ACM.

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

Ruggles, S., Genadek, K., Goeken, R., Grover, J., and Sobek, M. (2015). *Integrated public use microdata series: Version 6.0 [Machine-readable database]*. University of Minnesota.

Schläfli, L. (1901). *Theorie der vielfachen Kontinuität*, volume 38. Zürcher & Furrer.

Sistla, A. P., Wolfson, O., Chamberlain, S., and Dao, S. (1997). Modeling and Querying Moving Objects. In *ICDE'97*, pages 422–432.

Stout, Q. F. and Warren, B. L. (1986). Tree rebalancing in optimal time and space. *Communications of the ACM*, 29(9):902–908.

Tang, B., Mouratidis, K., and Yiu, M. L. (2017). Determining the impact regions of competing options in preference space. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 805–820. ACM.

Tang, Y., Liu, L., Wang, T., Hu, X., Sailer, R., and Pietzuch, P. (2014). Outsourcing multi-version key-value stores with verifiable data freshness. In *Proceedings of ICDE'14*, pages 1214–1217. IEEE.

Tang, Y., Wang, T., Hu, X., Jang, J., Liu, L., and Pietzuch, P. (2013). Authentication of freshness for outsourced multi-version key-value stores.

Tao, Y., Hristidis, V., Papadias, D., and Papakonstantinou, Y. (2007). Branch-and-bound processing of ranked queries. *Information Systems*, 32(3):424–445.

Thiagarajan, A. and Madden, S. (2008). Querying continuous functions in a database system. In *SIGMOD'2008*, pages 791–804.

Viner, J. (1932). *Cost curves and supply curves*. Springer.

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

Vlachou, A., Doulkeridis, C., Kotidis, Y., and Norvag, K. (2011). Monochromatic and bichromatic reverse top-k queries. *TKDE*, 23(8):1215–1229.

Winder, R. (1966). Partitions of n-space by hyperplanes. *SIAM Journal on Applied Mathematics*, 14(4):811–818.

Wolsey, L. A. and Nemhauser, G. L. (2014). *Integer and combinatorial optimization*. John Wiley & Sons.

Wu, D., Choi, B., Xu, J., and Jensen, C. S. (2015). Authentication of moving top-k spatial keyword queries. *Knowledge and Data Engineering, IEEE Transactions on*, 27(4):922–935.

Yang, G. and Cai, Y. (2017). Querying improvement strategies. In *EDBT*, pages 294–305.

Yang, Y., Papadopoulos, S., Papadias, D., and Kollios, G. (2008). Spatial outsourcing for location-based services. In *Proceedings of ICDE08*, pages 1082–1091. IEEE.

Yang, Y., Papadopoulos, S., Papadias, D., and Kollios, G. (2009). Authenticated indexing for outsourced spatial databases. *The VLDB Journal*, 18(3):631–648.

Zhang, J., Mouratidis, K., and Pang, H. (2014a). Global immutable region computation. In *SIGMOD'14*, pages 1151–1162.

Zhang, Z., Hwang, S.-w., Chang, K. C.-C., Wang, M., Lang, C. A., and Chang, Y.-c. (2006). Boolean+ ranking: querying a database by k-constrained optimization. In *SIGMOD'2006*, pages 359–370. ACM.

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

Zou, L. and Chen, L. (2008). Dominant graph: An efficient indexing structure to answer top-k queries. In *ICDE'2008*, pages 536–545.