A General Framework for Modeling and Processing Optimization Queries

Michael Gibas
Ohio State University
Department of Computer Science and Engineering
Columbus, OH, USA 43210
gibas@cse.ohio-state.edu

Ning Zheng
Ohio State University
Department of Industrial and Systems Engineering
Columbus, OH, USA 43210
zheng.481@osu.edu

Hakan Ferhatosmanoglu
Ohio State University
Department of Computer Science and Engineering
Columbus, OH, USA 43210
hakan@cse.ohio-state.edu

ABSTRACT

An optimization query asks for one or more data objects that maximize or minimize some function over the data set. We propose a general class of queries, model-based optimization queries, in which a generic model is used to define a wide variety of queries involving an optimization objective function and/or a set of constraints on the attributes. This model can be used to define optimization of linear and nonlinear expressions over object attributes as well as many existing query types studied in database research literature. A significant and important subset of this general model relevant to real-world applications include queries where the optimization function and constraints are convex. We cast such queries as members of the convex optimization (CP) model and provide a unified query processing framework for CP queries that I/O **optimally** accesses data and space partitioning index structures without changing the underlying structures. We perform experiments to show the generality of the technique and where possible, compare to techniques developed for specialized optimization queries. We find that we achieve nearly identical performance to the limited optimization query types with optimal solutions, while providing generic modeling and processing for a much broader class of queries, and while effectively handling problem constraints.

1. INTRODUCTION

Motivation and Goal Optimization queries form an important part of real-world database system utilization, especially for information retrieval and data analysis. Exploration of image, scientific, and business data usually requires iterative analyses that involve sequences of queries with varying parameters. Besides traditional queries, similarity-based analyses and complex mathematical and statistical operators are used to query such data [18]. Many of these data exploration queries can be cast as optimization queries where a linear or non-linear function over object attributes is minimized or maximized. Additionally, the object attributes queried, function parametric values, and data constraints can vary on a per-query basis.

There has been a rich set of literature on processing *specific* types of queries, such as the query processing algorithms for Nearest Neighbors (NN) [22] and its variants [14], top-k queries [5, 24], etc. These queries can all be considered to be a type of optimization query with different objective functions. However, they either only deal with specific function forms or require strict properties for the query functions.

Because scientific data exploration and business decision support systems rely heavily on function optimization tasks, and because such tasks encompass such disparate query types with varying parameters, these application domains can benefit from a general model-based optimization framework. Such a framework should meet the following requirements: (i) have the expressive power to support existing query types and the power to formulate new query types, (ii) take advantage of query constraints to proatively prune search space, and (iii) maintain efficient performance when the scoring criteria or optimization function changes.

Despite an extensive list of techniques designed specifically for different types of optimization queries, a unified query technique for a general optimization model has not been addressed in the literature. Furthermore, application of specialized techniques requires that the user know of, possess, and apply the appropriate tools for the specific query type. We propose a generic optimization model to define a wide variety of query types that includes simple aggregates, range and similarity queries, and complex user-defined functional analysis. Since the queries we are considering do not have a specific objective function but only have a "model" for the objective (and constraints) with parameters that vary for different users/queries/iterations, we refer to them as *model-based optimization queries*. In conjunction with a technique to model optimization query types, we also propose a query processing technique that optimally accesses data and space partitioning structures to answer the query. By applying this single optimization query model with I/O-optimal query processing, we can provide the user with a flexible and powerful method to define and execute arbitrary optimization queries efficiently.

Example Optimization Query Applications There are numerous examples of optimization queries in scientific application domains. Consider a data repository of

High Energy Physics (HEP) experimental data involving billions of events each with several hundred attributes [23, 13]. Each tuple corresponds to an event/experiment, and each attribute corresponds to an observed or simulated value for an experimental parameter. Physicists identify the significant events and study them further. For example, they are interested in events that maximize a simple non-linear function such as the momentum $(m \times v)$ and energy $(\frac{1}{2} \times m \times v^2)$ of the moving particles. A variety of query forms, including simple aggregation over the number of particles, complex scientific queries such as optimizing a function over some physical or kinematical quantity, and selecting sequences of events correlated in certain attributes, are executed over the data.

Another example of non-linear function optimization is in the environmental monitoring domain. Environmental scientists are interested in regions with high coastal vulnerability index (CVI), i.e., relative vulnerability of the coast to physical changes due to sea-level rise. The index is calculated as the square root of the weighted geometric mean of six variables: geomorphology, coastal slope, relative sea-level rise rate, shoreline erosion/accretion rate, mean tide range, and mean wave height [9]. To identify and delineate coastal regions with high erosion vulnerability, a maximization query over this non-linear function needs to be executed, potentially with additional constraints over several other attributes.

Another common real world database problem deals with matching queries. These queries involve finding the best match(es) in the database to an object that meet a set of constraints. One example would be the problem of matching patients for possible inclusion in appropriate clinical trials. Clinical trial administrators want to prioritize potential participants based on a combination of the likelihood of patient participation and some measure of the desirability of that patient for that study. Patients that do not meet certain hard inclusion and exclusion criteria constraints (e.g., the patient is between 35 and 55 years of age and is female) for the study should not be considered. This problem involves solving complex query types, such as linear optimization (LP) or nearest neighbor (NN) in the context of constrained dataspaces.

Systems incorporating user relevance feedback have frequently changing model parameters. The weighted nearest neighbor (WNN) query type forms an example, where each query defines its own weights for the similarity function over the dimensions. Weighted Euclidean distance is commonly used as a similarity measure in content-based image retrieval [21]. Multi-dimensional color or texture features are automatically extracted from the images, and the relative importance of each dimension is initially unknown. The weights are initially assigned using some statistical property of the feature vectors, such as the inverse standard deviation for each dimension of the feature vector. Then the weights are further improved by incorporating user's relevance feedback. Since queries are iteratively executed using the same model but with varying parameters, a common and efficient framework for processing these queries is needed.

Because optimization queries encompass such disparate query types with varying parameters, a general model-based querying framework that has the expressive power to support solution of each of the query types, that can take advantage of query constraints to prune search space, and that still maintains efficient performance when the scoring criteria or optimization function changes, will improve the effectiveness of scientific data exploration and business decision support systems.

Contributions The primary contributions of this work are listed as follows.

- We propose a general framework to execute convex *model-based* optimization queries (using linear and non-linear functions) and for which additional constraints over the attributes can be specified without compromising query accuracy or performance.
- We define query processing algorithms for convex model-based optimization queries utilizing data/space partitioning index structures without changing the index structure properties and the ability to efficiently address the query types for which the structures were designed.
- We prove the I/O optimality for these types of queries for data and space partitioning techniques when the objective function is convex and the feasible region defined by the constraints is convex (these queries are defined as CP queries in this paper).
- We introduce a generic model and query processing framework that optimally addresses existing optimization query types studied in the research literature as well as new types of queries with important applications.

2. RELATED WORK

There are few published techniques that cover some instances of model-based optimization queries. One is the "Onion technique" [4], which deals with an *unconstrained* and *linear* query model. An example linear model query selects the best school by ranking the schools according to some linear function of the attributes of each school. The solution followed in the Onion technique is based on constructing convex hulls over the data set. It is infeasible for high dimensional data because the computation of convex hulls is exponential with respect to the number of dimensions and is extremely slow. It is also not applicable to *constrained* linear queries because convex hulls need to be recomputed for each query with a different set of constraints. The computation of convex hulls over the whole data set is expensive and in the case of high-dimensional data, infeasible in design time.

The Prefer technique [12] is used for *unconstrained* and *linear* top-k query types. The access structure is a sorted list of records for an arbitrary linear function. Query processing is performed for some new linear preference function by scanning the sorted list until a record is reached such that no record below it could match the new query. This avoids the costly build time associated with the Onion technique, but does not provide a guarantee on minimum I/O, and still does not incorporate constraints.

Boolean + Ranking [26] offers a technique to query a database to optimize boolean constrained ranking functions. However, it is limited to boolean rather than general convex constraints, addresses only single dimension access structures (i.e. can not optimize on multiple dimensions simultaneously), and therefore largely uses heuristics to determine a cost effective search strategy.

While the Onion and Prefer techniques organize data to efficiently answer a specific type of query (LP), our technique organizes data retrieval to answer more general queries. In contrast with the other techniques listed, our proposed solution is proven to be I/O-optimal for both hierarchical and space partitioned access structures with convex partition boundaries, covers a broader spectrum of optimization queries, and requires no additional storage space.

3. QUERY MODEL OVERVIEW

3.1 Background Definitions

Let $Re(a_1, a_2, \ldots, a_n)$ be a relation with attributes $a_1, a_2, \ldots, a_n$. Without loss of generality, assume all attributes have the same domain. Denote by A a subset of attributes of Re. Let $g_i(\vec{\alpha}, A), 1 \leq i \leq u$, $h_j(\vec{\beta}, A), 1 \leq j \leq v$ and $F(\vec{\theta}, A)$ be certain functions over attributes in A, where u and v are positive integers and $\vec{\theta} = (\theta_1, \theta_2, \ldots, \theta_m)$ is a vector of parameters for function F, and $\vec{\alpha}$ and $\vec{\beta}$ are vector parameters for the constraint functions.

Definition 1 (Model-Based Optimization Query). *Given a relation $Re(a_1, a_2, \ldots, a_n)$, a model-based optimization query Q is given by (i) an objective function $F(\vec{\theta}, A)$, with optimization objective o (min or max), (ii) a set of constraints (possibly empty) specified by inequality constraints $g_i(\vec{\alpha}, A) \leq 0, 1 \leq i \leq u$ (if not empty) and equality constraints $h_j(\vec{\beta}, A) = 0, 1 \leq j \leq v$ (if not empty), and (iii) user/query adjustable objective parameters $\vec{\theta}$, constraint parameters $\vec{\alpha}$ and $\vec{\beta}$, and answer set size integer k.*

Figure 1 shows a sample objective function with both inequality and equality constraints. The objective function finds the nearest neighbor to point a (1,2). The constraints limit the feasible region to the line passing through $x = 5$, where $3 \leq y \leq 6$. The point b represents the best point within the constraints for the objective function, while the point d represents the worst point within the constraints for the objective function. Point c could be the actual point in the database that meets the constraints and minimizes the objective function.

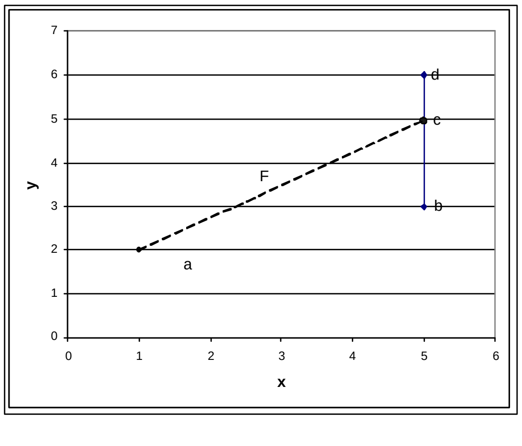

Figure 1: Sample Optimization Objective Function and Constraints $F(\vec{\theta}, A) : (x-1)^2 + (y-2)^2$, $o(min)$, $g_1(\vec{\alpha}, A) : y - 6 \leq 0$, $g_2(\vec{\alpha}, A) : -y + 3 \leq 0$, $h_1(\vec{\beta}, A) : x - 5 = 0$, $k = 1$

The answer of a model-based optimization query is a set of tuples with maximum cardinality k satisfying all the constraints such that there exists no other tuple with smaller (if minimization) or larger (if maximization) objective function value F satisfying all constraints as well. This definition is very general, and almost any type of query can be considered as a special case of model-based optimization query. For instance, NN queries over an attribute set A can be considered as model-based optimization queries with $F(\vec{\theta}, A)$ as the distance function (e.g., Euclidean) and the optimization objective is minimization. Similarly, top-k queries, weighted NN queries and linear/non-linear optimization queries can all be considered as specific model-based optimization queries. Without loss of generality, we will consider minimization as the objective optimization throughout the rest of this paper.

Definition 2 (Convex Optimization (CP) Query). *A model-based optimization query Q is a Convex Optimization (CP) query if (i) $F(\vec{\theta}, A)$ is convex, (ii) $g_i(\vec{\alpha}, A), 1 \leq i \leq u$ (if not empty) are convex, and (iii) $h_j(\vec{\beta}, A), 1 \leq j \leq v$ (if not empty) are linear or affine.*

Notice that the definition of a CP query does not have any assumptions on the *specific* form of the objective function. The only assumptions are that the queries can be formulated into convex functions and the feasible regions defined by the constraints of the queries are convex (e.g., polyhedron regions). Therefore, users can ask any form of queries with any coefficients as long as these assumptions hold. The conditions (i), (ii) and (iii) form a well known type of problem in Operations Research literature - Convex Optimization problems. A function is convex if it is second differentiable and its Hessian matrix is positive definite. More intuitively, if one travels in a straight line from inside a convex region to outside the region, it is not possible to re-enter the convex region.

3.2 Common Query Types Cast into Optimization Query Model

Although appearing to be restricted in functions F, g_i and h_j, the set of CP problems is a superset of all least square problems, linear programming problems (LP) and convex quadratic programming problems (QP) [10]. Therefore, they cover a wide variety of linear and non-linear queries, including NN queries, top-k queries, linear optimization queries, and angular similarity queries. The formulation of some common query types as CP optimization queries are listed and discussed below.

Euclidean Weighted Nearest Neighbor Queries - The Weighted Nearest Neighbor query asking NN for a point $(a_1^0, a_2^0, \ldots, a_n^0)$ can be considered as an optimization query asking for a point $(a_1, a_2, \ldots, a_n)$ such that an objective function is minimized. For Euclidean WNN the objective function is

$$\mathrm{WNN}(a_1, a_2, \ldots, a_n) = \sqrt{w_1(a_1 - a_1^0)^2 + w_2(a_2 - a_2^0)^2 + \ldots + w_m(a_n - a_n^0)^2},$$

where $w_1, w_2, \ldots, w_n > 0$ are the weights and can be different for each query. Traditional Nearest Neighbor queries are the subset of weighted nearest neighbor queries where the weights are all equal to 1.

Linear Optimization Queries - The objective function of a Linear Optimization query is

$$\mathrm{L}(a_1, a_2, \ldots, a_n) = c_1a_1 + c_2a_2 + \ldots + c_na_n$$

where $(c_1, c_2, \ldots, c_n) \in \mathbb{R}^n$ are coefficients and can be different for different queries, and the constraints form a polyhedron. Since linear functions are convex and polyhedrons are convex, linear optimization queries are also special cases of CP queries with a parametric function form.

Top-k Queries - The objective functions of top-k queries are score (or ranking) functions over the set of attributes. The common score functions are sum and average, but could be any arbitrary function over the attributes. Clearly, sum and average are special cases of linear optimization queries. If the ranking function in question is convex, the top-k query is a CP query.

Range Queries - A range query asks for all data $(a_1, a_2, \ldots, a_n)$ s.t. $l_i \leq a_i \leq u_i$, for some (or all) dimensions i, where $[l_i, u_i]^n$ is the range for dimension i. Constructing an objective function as $f(a_1, a_2, \ldots, a_n) = C$, where C is any constant, and constraints

$$g_i = l_i - a_i <= 0, g_{n+i} = a_i - u_i <= 0, i = 1, 2, \cdots, n$$

Any points that are within the constraints will get the objective value C. Points that are not within the constraints are pruned by those constraints. Those points that remain all have an objective value of C and because they all have the same maximum(minimum) objective value, all form the solution set of the range query. Therefore, range queries can be formed as special cases of CP queries with constant objective functions applying the range limits as constraints. Also note that our model can not only deal with the 'traditional' hyper-rectangular range queries as described above, but also 'irregular' range queries such as $l \leq a_1 + a_2 \leq u$ and $l \leq 3a_1 - 2a_2 \leq u$

3.3 Optimization Query Applications

Any problem that can be rewritten as a convex optimization problem can be cast to our model, and solved using our framework. In cases where the optimization problem or objective function parameters are not known in advance, this generic framework can be used to solve the problem by efficiently accessing available access structures. We provide a short list of potential optimization query applications that could be relevant during scientific data exploration or business decision support where the problem being optimized is continuously evolving.

Weighted Constrained Nearest Neighbors - Weighted nearest neighbor queries give the ability to assign different weights for different attribute distances for nearest neighbor queries. Weighted Constrained Nearest Neighbors (WCNN) find the closest weighted distance object that exists within some constrained space. This type of query would be applicable in situations where attribute distances vary in importance and some constraints to the result set are known. A potential application is clinical trial patient matching where an administrator is looking to find the best candidate patients to fill a clinical trial. Some hard exclusion criteria is known and some attributes are more important than others with respect to estimating participation probability and suitability.

kNN with Adaptive Scoring - In some situations we may have an objective function that changes based on the characteristics of a population set we are trying to fill. In such a case, we will change the nearest neighbor scoring weights dependent on the current population set. As an example, again consider the patient clinical trial matching application where we want to maintain a certain statistical distribution over the trial participants. In a case where we want half of the participants to be male and half female, we can adjust weights of the objective optimization function to increase the likelihood that future trial candidates will match the currently underrepresented gender.

Queries over Changing Attributes - The attributes involved in optimization queries can vary based on the iteration of the query. For example, a series of optimization queries may search a stock database for maximum stock gains over different time intervals. The attributes involved in each query will be different.

Weights, constraints, functional attributes, and optimization functions themselves can all change on a per-query basis. A database system that can effectively handle the potential variations in optimization queries will benefit data exploration tasks. In the examples listed above, each query or each member of a set of queries can be rewritten as an optimization query in our model. This demonstrates the power and flexibility that the user has to define data exploration queries and the examples represent a small subset of the query types that are possible.

3.4 Approach Overview

We propose the use of Convex Optimization (CP) in order to traverse access structures to find optimal data objects. The goal in CP is optimize (minimize or maximize) some convex function over data attributes. The solution can optionally be subject to some convex criteria over the data attributes. Additionally, the data attributes may be subject to lower and upper bounds.

All of the discussed applications can be stated as an objective function with a objective of maximization or minimization. We can solve a continuous CP problem over a convex partition of space by optimizing the function of interest within the constraints of that space. Most access structures are built over convex partitions. Particularly common partitions are Minimum Bounding Rectangles (MBRs). Rectangular partitions can be represented by lower and upper bounds over data attributes and can be directly addressed by the CP problem bounds. Other convex partitions can be addressed with data attribute constraints (such as $x+y < 5$). If the problem under analysis has constraints in addition to the constraints imposed by the partition boundaries, they can also be added to the CP problem as constraints.

Given the function, partition constraints, and problem constraints, we can use CP to find the optimal answer for the function within the intersection of the problem constraints and partition constraints. If no solution exists (problem constraints and partition constraints do not intersect), the partition is found to be infeasible for the problem. The partition and its children can be eliminated from further consideration. Given a function, problem constraints, and a set of partitions, the partition the yields the best CP result with respect to the function under analysis is the one that contains some space that optimizes the problem under analysis better than the other partitions. These partition functional values can be used to order partitions according to how promising they are with respect to optimizing the function within problem constraints.

With our technique we endeavor to minimize the I/O operations and access structure search computations required to perform arbitrary model-based optimization queries. The access structure partitions to search and evaluate during query processing are determined by solving CP problems that incorporate the objective function, model constraints, and partition constraints. The solution of these CP problems allows us to prune partitions and search paths that can not contain an optimal answer during query processing.

The CP literature discusses the efficiency of CP problem solution. In [19], it is stated that a CP problem can be very efficiently solved with polynomial-time algorithms. The readers can refer to [20] for a detailed review on interior-point polynomial algorithms for CP problems. Current implementations of CP algorithms can solve problems with hundreds of variables and constraints in very short times on a personal computer [15]. CP problems can be so efficiently solved that Boyd and Vandenberghe stated "if you formulate a practical problem as a convex optimization problem, then you have solved the original problem." ([3] page 8). We are in effect replacing the bound computations from existing access structure distance optimization algorithms with a CP problem that covers the objective function and constraints. Although such computations can be more complex, we found very little time difference in the functions we examined.

We focus on query processing techniques by presenting a technique for general CP objective functions, which can be applied to existing space or data partitioning-based indices without losing the capabilities of those indexing techniques to answer other types of queries. The basic idea of our technique is to divide the query processing into two phases: First, solve the CP problem without considering the database, (i.e. in continuous space). This "relaxed" optimization problem can be solved efficiently in the main memory using many possible algorithms in the convex optimization literature. It can be solved in polynomial time in the number of dimensions in the objective functions and the number of constraints ([20]). Second, search through the index by pruning the impossible partitions and access the most promising pages. Details for our technique for different index types are provided in the following section.

4. QUERY PROCESSING FRAMEWORK

By solving CP problems associated with the problem constraints, partition constraints, and objective function, we find the best possible objective function value achievable by a point in the partition. Thus, there exists a lower bound for each partition in the database that can be used to prune the data space before retrieving pages from disk. The bounds can be defined for any type of convex partition (such as minimum bounding regions) used for indexing the data.

Figure 2 is a functional block diagram of the optimization query processing system that shows interactions between components. Query processing proceeds first by the user providing an objective function and constraints to the system. A lightweight query compiler validates the user input, converts the function and constraints into appropriate format for the query processor, and executes the appropriate query processing algorithm using the specified access structure. The query processor invokes a CP solver to find bounds for partitions that could potentially contain answers to the query. By solving CP subproblems using index partition boundaries as additional constraints, we can ensure that we access pages in the order of how promising they are. The user gets back those data object(s) in the database that answer the query within the constraints.

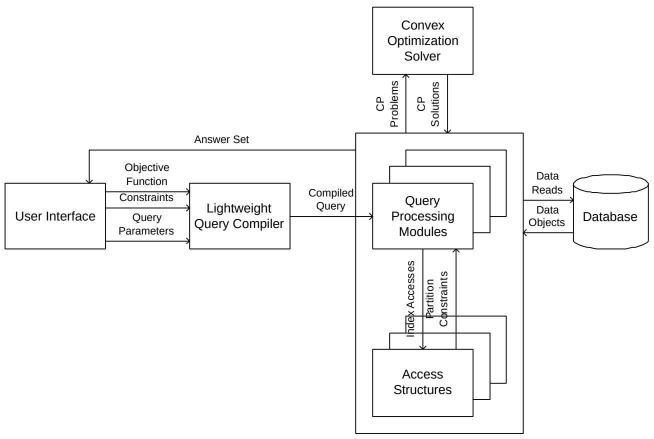

Figure 2: Optimization Query System Functional Block Diagram

The generic query processing framework for a CP query is described below. This framework can be applied to indexing structures in which the partitions on which they are built are convex because the partition boundaries themselves are used to form the new CP problems.

A popular taxonomy of access structures divides the structures into the categories of hierarchical and non-hierarchical structures. Hierarchical structures are usually partitioned based on data objects while non-hierarchical structures are usually based on partitioning space. These two families have important differences that affect how queries are processed over them. Therefore we provide algorithms for hierarchical structures in Subsection 4.1 and non-hierarchical structures in 4.2.

The overall idea is to traverse the access structure and solve the problem in sequential order of how good partitions and search paths *could* be. We terminate when we come to a partition that could not contain an optimal answer to the query. The implementations described address a minimization objective function and prune based on comparison to minimum values. Solutions to maximization problems can be performed by using maximums (i.e. "worst" or upper bounds) in place of minimums.

The number of results that are returned by the query processing is dependent on an input k. If the user desires only the single best result, $k = 1$. It can however, be an arbitrary value in order to return the k best data objects. For range queries, if all objects that meet the range constraints are desired, the value of k should be set to n, the number of data points. Only the data points within the constrained region will be returned, and only the points that are within the lowest-level partitions that intersect the constrained region will be examined during query processing. Therefore, the query processing will be at least as I/O-efficient as standard range query processing over a given access structure.

Now, we describe the implementations of this framework based on hierarchical and non-hierarchical indexing techniques. In Section 5 we show that both of these algorithms are in fact *optimal*, i.e., no unnecessary MBRs (or partitions) given an access structure are accessed during the query processing.

4.1 Query Processing over Hierarchical Access Structures

Algorithm 1 shows the query processing over hierarchical access structures. This algorithm is applicable to any hierarchical access structure where the partitions at each tree level are convex. A partial list of structures covered by this algorithm includes the R-tree, R*-tree, R+-tree, BSP-tree, Quad-tree, Oct-tree, k-d-B-tree, B-tree, B+-tree, LSD-tree, Buddy-tree, P-tree, SKD-tree, GBD-tree, Extended k-D-Tree, X-tree, SS-tree, and SR-tree [7, 2].

The algorithm is analogous to existing optimal NN processing [11], except that partitions are traversed in order of promise with respect to an arbitrary convex function, rather than distance. Additionally, our algorithm also prunes out any partitions and search paths that do not fall within problem constraints.

We solve a new CP problem for each partition that we traverse using the objective function, original problem constraints, and the constraints imposed by the partition boundaries. The algorithm takes as input the convex optimization function of interest OF, the set of convex constraints C, and a desired result set size k. In lines 1-3, we initialize the structures that we maintain throughout a query. We keep a vector of the k best points found so far as well as a vector of the objective function values for these points. We keep a worklist of promising partitions along with the minimum objective value for the partition. We initialize the worklist to contain the tree root partition.

We then process entries from the worklist until the worklist is empty. In line 5, we remove the first promising partition. If this partition is a leaf partition, then we access the points in the partition and compute their objective function values. If a point is not within the original problem constraints, then it will not have a feasible solution and will be discarded from further consideration. Points with objective function values lower than the objective function value of the k^{th} best point so far will cause the best point and best point objective function value vectors to be updated accordingly.

If the partition under analysis is not a leaf, we find its child partitions. For each of these child partitions, we solve the CP problem corresponding to the objective function, original problem constraints, and partition constraints (line 11). This yields the best possible objective function value achievable within the intersection of the original problem constraints and partition constraints. If the minimum objective function value is less than the k^{th} best objective function value (i.e. it is possible for a point within the intersection of the problem constraints and partition constraints to beat the k^{th} best point), then the partition is inserted into the worklist according to its minimum objective value. If there is no feasible solution for the partition within problem constraints, the partition will be dropped from consideration.

After a partition is processed, we may have updated our k^{th} best objective function value. We prune any partitions in the worklist that can not beat this value.

Query Processing Example As an example of query processing, consider Figure 3 with an optimization objective of minimizing the distance to the query point within the constrained region (i.e. constrained 1-NN query). We initialize our worklist with the root and start by processing it. Since it is not a leaf, we examine its child partitions A and B. We solve for the minimum objective function value for A and do not obtain a feasible solution because A and the constrained region do not intersect. We do not process A further. We find the minimum objective function for partition B and get the distance from the query point to the point where the constrained region and the partition B intersect, point c. We insert partition B into our worklist and since it is the only partition in the worklist, we process it next. Since B is not a leaf, we examine its child partitions, $B.1, B.2$, and $B.3$. We find minimum objective function values for each and obtain the distances from the query point to point d, e, and f respectively. Since point e is closest, we insert partitions into the worklist in the order $B.2, B.1$, and $B.3$.

We process $B.2$ and since it is a leaf, we compute objective function values for the points in $B.2$. Point $B.2.a$ is the closer of the two. This point is designated as the best point thus far, and the best point objective function is updated. After processing $B.2$, we know we can do no worse than the distance from the query point to $B.2.a$. Since partition $B.3$ can not do better than this, we prune it from the worklist. We examine points in $B.1$. Point $B.1.b$ is not a feasible solution (it is not in the constrained region) so it is dropped. Point $B.1.a$ gets a value which is better than the current best point. We update the best point to be $B.1.a$. Since the worklist is now empty, we have completed the query and

```
CPQueryHierarchical (Objective Function $OF$,
Constraints $C$,$k$)

notes:
$b[i]$ - the $i^{th}$ best point found so far
$F(b[i])$ - the objective function value of the $i^{th}$ best point
$L$ - worklist of (partition, minObjVal) pairs

1:   $b[1]...b[k] \leftarrow$null.
2:   $F(b[1])...F(b[k]) \leftarrow +\infty$
3:   $L \leftarrow (root)$
4:   while $L \neq \emptyset$
5:       remove first partition $P$ from $L$
6:       if $P$ is a leaf
7:           access and compute functions for points in
             $P$ within constraints $C$
8:           update vectors $b$ and $F$ if better points
             discovered
9:       else
10:          for each child $P_{child}$ of $P$
11:              $minObjVal$ = CPSolve($OF$,$P_{child}$,$C$)
12:              if $minObjVal < F(b[k])$
13:                  insert $P_{child}$ into $L$ according to
                     $minObjVal$
14:          end for
15:      end if
16:      prune partitions from $L$ with $minObjVal >$
         $F(b[k])$
17:  end while
18:  return vector $b$
```

Algorithm 1: I/O Optimal Query Processing for Hierarchical Index Structures.

return the best point.

The optimal point for this optimization query this query is $B.1.a$. We examine only points in partitions that could contain points as good as the best solution.

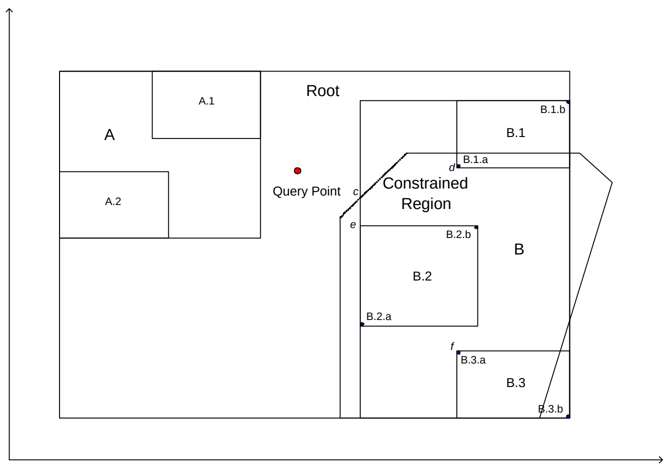

Figure 3: Example of Hierarchical Query Processing

4.2 Query Processing over Non-hierarchical Access Structures

In this section, we show that the general framework for CP queries can also be implemented on top of non-hierarchical structures. Algorithm 2 can be applied to non-hierarchical access structures where the partitions are convex. A partial list of structures where the algorithm can be applied is Linear Hashing, Grid-File, EXCELL, VA-File, VA+-File, and Pyramid Technique [7, 2, 25, 6].

We use a two stage approach detailed in Algorithm 2 to determine which data objects optimize the query within the constraints. This technique is similar to the technique identified in [25] for finding NN for a non-hierarchical structure, but we use the objective function rather than a distance and also consider original problem constraints to find lower bounds. We take the objective function, OF, original problem constraints C, and result set size k as inputs. We initialize the structures to be used during query processing (lines 1-3). In stage 1, we process each populated partition vin the structure by solving the CP problem that corresponds to the original objective function and constraints combined with the additional constraints imposed by the partition boundaries (line 5). Those partitions that do not intersect the constraints will be bypassed. For those that do have a feasible solution, the lower bound of the CP problem is recorded. At the end of stage 1, we place unique unpruned partitions in increasing sorted order based on their lower bound. In stage 2, we process the unpruned partitions by accessing objects contained by the first partition and computing actual objective values. We maintain a sorted list of the top-k objects we have found so far, and continue accessing points contained by subsequent partitions until a partition's lower bound is greater than the k^{th} best value. By accessing partitions in order of how good they could be, according to the CP problem, we access only those points in cell representations that could contain points as good as the actual solution.

```
CPQueryNonHierarchical (Objective Function $OF$,
Constraints $C$,$k$)

notes:
$b[i]$ - the $i^{th}$ best point found so far
$F(b[i])$ - the objective function value of the $i^{th}$ best point
$L$ - worklist of (partition, minObjVal) pairs

Initialize:
1:   $b[1]...b[k] \leftarrow$null.
2:   $F(b[1])...F(b[k]) \leftarrow +\infty$
3:   $L \leftarrow \emptyset$
Stage 1:
4:   for each populated partition $P$
5:       $minObjVal$ = CPSolve($OF$, $P$, $C$)
6:       if $minObjVal < +\infty$
7:          insert ($P$, $minObjVal$) into $L$ according to
            $minObjVal$
Stage 2:
8:  for $i = 1$ to $|L|$
9:       if $minObjVal$ of partition $L[i] > F(b[k])$
10:      break
11:      Access objects that partition $L[i]$ contains
12:      for each object $o$
13:         if $OF(o) < F(b[k])$
14:            insert $o$ into $b$
15:            update $F(b)$
```

Algorithm 2: I/O Optimal Query Processing over Non-hierarchical Structures.

4.3 Non-Covered Access Structures

A review of access structure surveys yielded few structures that are not covered by the algorithms. These include structures that either do not guarantee spatial locality (i.e. space filling curves which are used for approximate ordering). They include structures built over values computed for a specific function (such as distance in M-trees. Since attribute information is lost, they can not be applied to general functions over the original attributes). They include structures that contain non-convex regions (BANG-File, hB-tree, BV-tree) [7, 2]. Note that M-trees would still work to answer queries for the functions they are built over and the structures built with non-convex leaves could still be optimally traversed down to the level of non-convexity.

5. I/O OPTIMALITY

In this section, we will show that the proposed query processing technique achieves optimal I/O for each implementation in Section 4. We denote the objective function contour which goes through the actual optimal data point in the database as the optimal contour. This contour also goes through all other points in continuous space that yield the same objective function value as the optimal point. The k^{th} optimal contour would pass through all points in continuous space that yield the same objective function value as the k^{th} best point in the database. In the following arguments, we use the optimal contour, but could replace them with the k^{th} optimal contour. Following the traditional definition in [1], we define the optimality as follows.

DEFINITION 3 (OPTIMALITY). *An algorithm for optimization queries is optimal iff it retrieves only the pages that intersect the intersection of the constrained region R and the optimal contour.*

For hierarchical tree based query processing, the proof is given in the following.

LEMMA 1. *The proposed query processing algorithm is I/O optimal for CP queries under convex hierarchical structures.*

Proof. It is easy to see that any partition intersecting the intersection of optimal contour and feasible region R will be accessed since they are not pruned in any phase of the algorithm. We need to prove only these partitions are accessed, i.e., other partitions will be pruned without accessing the pages.

Figure 4 shows different cases of the various position relations between a partition, R and the optimal contour under 2 dimensions. $\boldsymbol{a}$ is an optimal data point in the data base. We will use $minObjVal(Partition, R)$ to denote the minimum objective function value that is computed for the partition and constraints R.
A: intersects neither R nor the optimal contour.
B: intersects R but not the optimal contour.
C: intersects the optimal contour but not R.
D: intersects the optimal contour and R, but not the intersection of optimal contour and R.
E: intersects the intersection of the optimal contour and R.

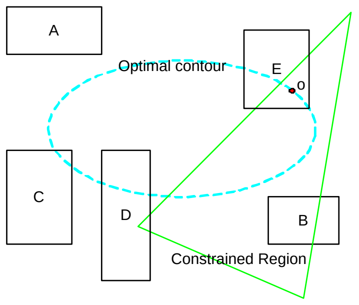

Figure 4: Cases of MBRs with respect to R and optimal objective function contour

A and C will be pruned since they are infeasible regions, when the CP for these partitions are solved, they will be eliminated from consideration since they do not intersect the constrained region and $minObjVal(A, R) = minObjVal(C, R) = +\infty$. B is pruned because $minObjVal(B, R) > F(\boldsymbol{a})$ and E will be accessed earlier than B in the algorithm because $minObjVal(E, R) < minObjVal(B, R)$. We shall show that a page in case D is pruned by the algorithm. Let CP_0 be the data partition that contains an optimal data point, CP_1 be the partition that contains $CP_0, \ldots$, and CP_k be the root partition that contains $CP_0, CP_1, , CP_{k-1}$. Because the objective function is convex, we have

$$F(\boldsymbol{a}) \geq minObjVal(CP_0, R) \geq minObjVal(CP_1, R) \geq \ldots \geq minObjVal(CP_k, R)$$

and

$$minObjVal(D, R) > F(\boldsymbol{a}) \geq minObjVal(CP_0, R) \geq minObjVal(CP_1, R) \geq \ldots \geq minObjVal(CP_k, R)$$

During the search process of our algorithm, CP_k is replaced by CP_{k-1}, and CP_{k-1} by CP_{k-2}, and so on, until CP_0 is accessed. If D will be accessed at some point during the search, then D must be the first in the MBR list at some time. This only can occur after CP_0 has been accessed because $minObjVal(D, R)$ is larger than constrained function value of any partition containing the optimal data point. If CP_0 is accessed earlier than D, however, the algorithm prunes all partitions which have a constrained function value larger than $F(\boldsymbol{a})$, including D. This contradicts with the assumption that D will be accessed.

The I/O-optimality relates to accessing data from leaf partitions. However, the algorithm is also optimal in terms of the access structure traversal, that is, we will never retrieve the objects within an MBR (leaf or non-leaf), unless that MBR could contain an optimal answer. The proof applies to any partition, whether it contains data or other partitions.

Similarly, we can prove the I/O optimality of proposed algorithm over non-hierarchical structures.

LEMMA 2. *The proposed query processing algorithm is I/O optimal for CP queries over non-hierarchical convex structures.*

Proof. Figure 5 shows different possible cases of the partition position. Partitions A, B, C, D, and E are the same as defined for Figure 4.

Without loss of generality, assume that only these five partitions contain some data points, and partition E contains an optimal data point, $\boldsymbol{a}$, for a query $F(x, y)$. Then an optimal algorithm retrieves only partition E.

Partitions C and A will not be retrieved because they are infeasible and our algorithm will never retrieve infeasible partitions. Partition B will not be retrieved because partition E will be retrieved before partition B (because $minObjVal(E, R) < minObjVal(B, R)$), and the best data point found in E will prune partition B. Thus, we only need to show that partition D will not be retrieved.

Because partition D does not intersect the intersection of the optimal contour and R but E does, $minObjVal(D, R) > minObjVal(E, R)$. Therefore, partition E must be retrieved before partition D since the algorithm always retrieves the most promising partition first, and the data point a is found at that time. Suppose partition D is also retrieved later. This means partition D cannot be pruned by data point a, i.e., $F(\boldsymbol{a}) \geq minObjVal(D, R)$, which also means the virtual solution corresponding to $(minObjVal(D, R))$ is contained by the optimal contour (because of the convexity of function F). Therefore, the virtual solution corresponding to $(minObjVal(D, R)) \in D \cap R \cap$optimal contour. Hence, we can find at least one virtual point x^* (without considering the database) in partition D such that x^* is both in R and the optimal contour. This contradicts to the assumption that partition D does not intersect the intersection of R and the optimal contour.

6. PERFORMANCE EVALUATION

The goals of the experiments are to show i) that the proposed framework is general and can be used to process a variety of queries that optimize some objective ii) that the generality of the framework does not impose significant cpu burden over those queries that already have an optimal so-

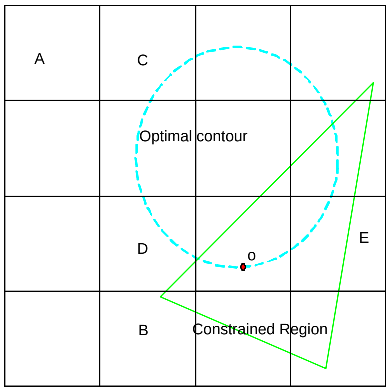

Figure 5: Cases of partitions with respect to R and optimal objective function contour

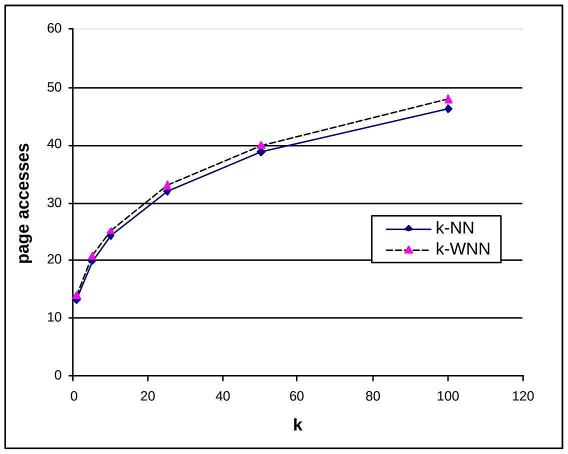

Figure 6: Accesses, Random weighted kNN vs kNN, R*-tree, 8-D Histogram, 100k pts

lution, and iii) that non-relevant data subspace can be effectively pruned during query processing.

We performed experiments for a variety of optimization query types including kNN, Weighted kNN, Weighted Constrained kNN, and Constrained Weighted Linear Optimization. Where possible, we compare cpu times against a non-general optimization technique. We index and perform queries over four real data sets. One of these is Color Histogram data, a 64-dimensional color image histogram dataset of 100,000 data points. The vectors represent color histograms computed from a commercial CD-ROM. The second is Satellite Image Texture (Landsat) data, which consists of 100,000 60-dimensional vectors representing texture feature vectors of Landsat images [17]. These datasets are widely used for performance evaluation of index structures and similarity search algorithms [16, 8]. We used a clinical trials patient data set and a stock price data set to perform real world optimization queries.

Weighted kNN Queries Figure 6 shows the performance of our query processing over R*-trees for the first 8-dimensions of the histogram data for both kNN and k weighted NN (k-WNN) queries. The index is built over the first 8 dimensions of the histogram data and the queries are generated to find the kNN or k-WNN to a random point in the data space over the same 8 dimensions. We execute 100 queries for each trial, and randomly assign weights between 0 and 1 for the k-WNN queries. We vary the value of k between 1 and 100. We assume the index structure is in memory and we measure the number of additional page accesses required to access points that could be kNN according to the query processing. Because the weights are 1 for the kNN queries, our algorithm matches the I/O-optimal access offered by traditional R-tree nearest neighbor branch and bound searching.

The figure shows that we achieve nearly the same I/O performance for weighted kNN queries using our query processing algorithm over the same index that is appropriate for traditional non-weighted kNN queries. For unconstrained, unweighted nearest neighbor queries, the optimal contour is a hyper-sphere around the query point with a radius equal to the distance of the nearest point in the data set. If instead, the queries are weighted nearest neighbor queries, the optimal contour is a hyper-ellipse. Intuitively, as the weights of a weighted nearest neighbor query become more skewed, the hyper-volume enclosed by the optimal contour increases, and the opportunity to intersect with access structure partitions increases. Results for the weighted queries track very closely to the non-weighted queries. This indicates that these weighted functions do not cause the optimal contour to cross significantly more access structure partitions than their non-weighted counterparts. For a weighted nearest neighbor query, we could generate an index based on attribute values transformed based on the weights and yield an optimal contour that was hyper-spherical. We could then traverse the access structure using traditional nearest neighbor techniques. However, this would require generating an index for every potential set of query weights, which is not practical for applications where weights are not typically known prior to the query. For similar applications where query weights can vary, and these weights are not known in advance, we would be better served to process the query over a single reasonable access structure in an I/O-optimal way than by building specific access structures for a specific set of weights.

Figure 7 shows the average cpu times required per query to traverse the access structure using convex problem solutions for the weighted kNN queries compared to the standard nearest neighbor traversal used for unweighted kNN queries.

The figure shows that the generic CP-driven solution takes slightly longer to perform than a specialized solution for nearest neighbor queries alone. Part of this difference is due to the slightly more complicated objective function (weighted versus unweighted Euclidean distance), part is due to the additional partitions that the optimal contour crosses, while the remaining part is due to incorporating data set constraints (unused in this example) into computing minimum partition function values.

Hierarchical data partitioning access structures are not effective for high-dimensional data sets. As data set dimensionality increases, the partitions overlap with each other more. As partitions overlap more, the probability that a point's objective function value can prune other partitions decreases. Therefore, the same characteristics that make high-dimensional R-tree type structures ineffective for traditional queries also make them ineffective for optimization

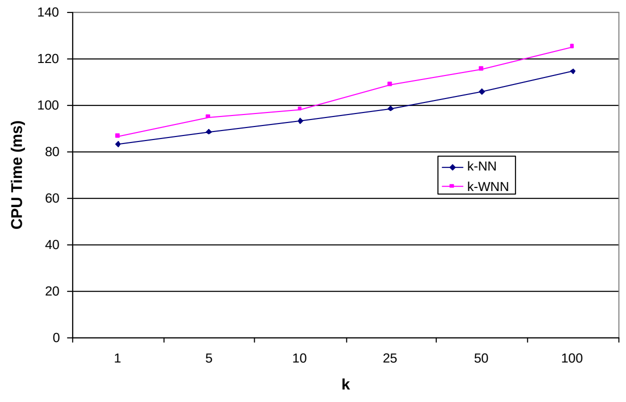

Figure 7: CPU Time, Random weighted kNN vs kNN, R*-tree, 8-D Histogram, 100k pts

queries. For this reason, we perform similar experiments for high-dimensional data using VA-files.

Figure 8 shows page access results for kNN and k-WNN queries over 60-dimensional, 5-bit per attribute, VA-file built over the landsat dataset. The experiments assume the VA-file is in memory and measure the number of objects that need to be accessed to answer the query.

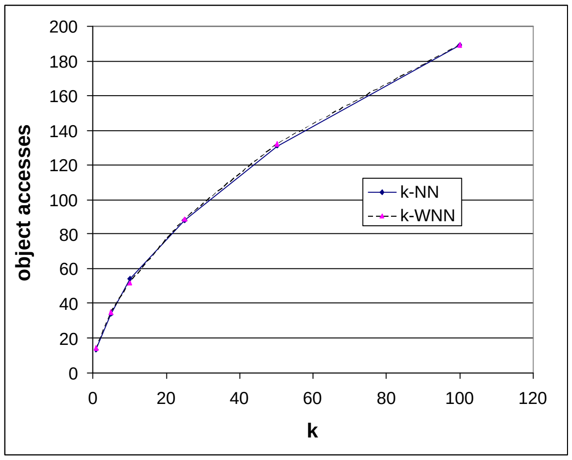

Figure 8: Accesses, Random weighted kNN vs kNN, VA-File, 60-D Sat, 100k pts

Similar to the R*-tree case, the results show that the performance of the algorithm is not significantly affected by re-weighting the objective function. Sequential scan would result in examining 100,000 objects. If the dataspace were more uniformly populated, we would expect to see object accesses much closer to k. However, much of the data in this data set is clustered, and some vector representations are heavily populated. If a best answer comes from one of these vector representations, we need to look at each object assigned the same vector representation.

It should be noted that while the framework results in optimal access of a given structure, it can not overcome the inherent weaknesses of that structure. For example, at high dimensionality, hierarchical data structures are ineffective, and the framework can not overcome this. A consequence of VA-File access structures is that, without some other data reorganization, a computation needs to occur for every approximated object. As distance computations need to be performed for each vector in traditional VA-file nearest neighbor algorithms, following the VA-file processing model so too must a CP problem be solved for each vector for general optimization queries. Times for this experiment reflect this fact. It takes longer to perform each query but the ratio of the general optimization query times to the traditional kNN CPU times is similar to the R*-tree case, about 1.006 (i.e. very little computational burden is added to allow query generality).

Constrained Weighted Nearest Neighbor Queries We also performed experiments over new query types, specifically the Weighted Constrained Nearest Neighbor (WCNN) queries discussed earlier. We performed such queries by finding the point within a constrained area that is a randomly weighted nearest neighbor to a randomly selected point in the dataspace (not necessarily in the constrained region). The constrained region of each query is a hypercube with one corner at the origin and extending the indicated distance in each queried dimension. Results are the average ratio of page accesses required to find the nearest neighbor over 100 queries. Figure 9 shows results for both 10,000 and 100,000 points of the first 8 dimensions of the color histogram data set using the 8-D R*-tree as the underlying access structure.

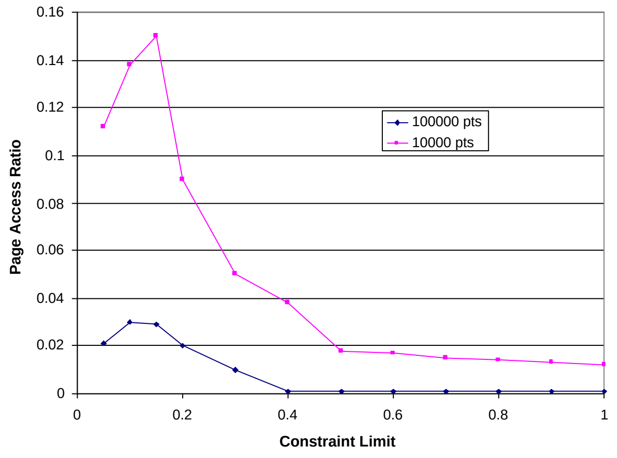

Figure 9: Accesses for Weighted Constrained Nearest Neighbor queries vs. constraints, Histogram

Constrained Weighted Linear Optimization Queries We also explore Constrained Weighted-Linear Optimization queries. Figure 10 shows the number of page accesses required to answer randomly weighted LP minimization queries for the histogram data set using the 8-D R*-Tree access structure. We vary k and we vary the constrained area region fixed at the origin as a 8-D hypercube with the indicated value. We generated 100 queries for each data point in the form of a summation of weighted attributes over the 8 dimensions. Weights are uniformly randomly generated and are between the values of -10 and 10.

The graph shows interesting results. The number of page accesses is not directly related to the constraint size. The

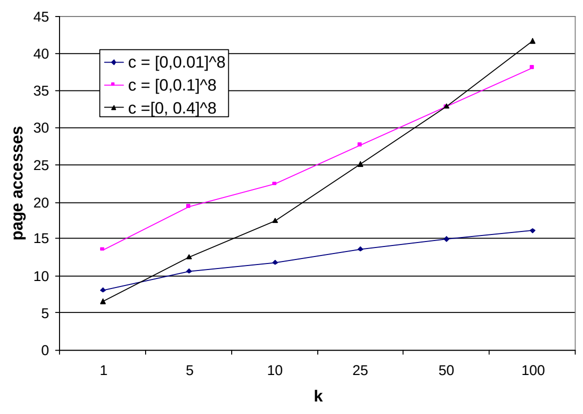

Figure 10: Page Accesses, Random Weighted Constrained k-LP Queries, R*-Tree, 8-D Color Histogram, 100k pts

rate of increase in page accesses to accommodate different values of k does vary with the constraint size. When weights can be negative, the corner corresponding to the minimum constraint value for the positively weighted attributes, and the maximum constrained value for the negatively weighted attributes will yield the optimal result within the constrained region (e.g. corner [0,1,0] would be an optimal minimization answer for a constrained unit cube and a linear function with a weight vector [+,-,+]). For this data set, many points are clustered around the origin, leading to denser packing of access structure partitions near the origin. Because of the sparser density of partitions around the point that optimizes the query, we access fewer partitions when this particular problem is less constrained. However, the radius of the k^{th}-optimal contour increases more in these less dense regions than more clustered regions, leading to a faster increase in the number of page accesses required when k increases.

Note that the Onion technique is not feasible for this dataset dimensionality, and it, as well as the other competing techniques are not appropriate for convex constrained areas. For comparative purposes, the sequential scan of this data set is 834 pages.

We should also note what happens when there are less than k optimal answers in the data set. This means that there are less than k objects in our constrained region. For our query processing, we only need to examine those points that are in partitions that intersect the constrained region. A technique that did not evaluate constraints prior to or during query processing will need to examine every point.

Queries with Real Constraints The previous example showed results with synthetically generated constraints. We also wanted to explore that constrained optimization problems for real applications. We performed Weighted Constrained kNN experiments to emulate a patient matching data exploration task described in Section 3.3. To test this, we performed a number of queries over a set of 17000 real patient records. We established constraints on age and gender and queried to find the 10 weighted nearest neighbors in the data set according to 7 measured blood analytes. Using the VA-file structure with 5 bits assigned per attribute and 100 randomly selected, randomly weighted queries, we achieved

Max. Stock Value	Page Accesses
5	3.5
10	4.2
25	3.4
50	2.9
100	3.2
200	3.9

Table 1: Page Accesses, k=1, Objective Function = max(Random Attribute/Attribute 1), R*-Tree, 10-D Stock Data, Constrained Maximum Stock Price

an average of 11.5 vector accesses to find the $k = 10$ results. This means that on average the number of vectors within the intersection of the constrained space and 10^{th}-optimal contour is 11.5. It corresponds to a vector access ratio of 0.0007.

Arbitrary Function Attributes Besides varying weights and constraints, the function itself and the attributes involved in a query to be optimized may differ for each optimization query. We took at data set of 6500 stock prices over 10 different times and performed an optimization query for the maximum stock price gain (new stock price/original stock price). The attribute to be used as the new stock price was randomly selected and we performed 100 of these queries for different constrained regions over a single 10-d R*-tree. Table 1 shows the average number of page accesses required to find the stock that optimized the function.

An alternative method to our technique to perform such a query would be to compute the function for all objects that met constraints and find the maximum. Since we do not know the function beforehand, we would have to compute it at run-time for constraint matching data objects. We instead traverse an access structure that covers any of our potential functions. We achieve relatively good results even though the dimensionality of the access structure is starting to get high (10 dimensions) and the data is highly correlated (since one day's stock price is highly dependent another day, the index space is not uniformly utilized). Results are not influenced much by the constraints in our case for this example, but they would be for the alternative approach. For comparative purposes, sequential scan of the data set takes 64 pages.

Arbitrary Functions over Different Access Structures In order to show that the framework can be used to find optimal data points for arbitrary convex functions, we generated a set of 100 random functions. Unlike nearest neighbor and linear optimization type queries, the continuous optimal solution is not intuitively obvious by examining the function. Functions contain both quadratic and linear terms and were generated over 5 dimensions in the form $\sum_{i=1}^{5}(random() * x_i^2 - random() * x_i)$, where x_i is the data attribute value for the i^{th} dimension. A sample function, where coefficients are rounded to two decimal places, is to minimize $0.91x_1^2 + 0.49x_2^2 + 0.4x_3^2 + 0.55x_4^2 + 0.76x_5^2 - 0.09x_1 - 0.49x_2 - 0.28x_3 - 0.91x_4 - 0.51x_5$.

We explored the results of our technique over 5 index structures. These include 3 hierarchical structures, including the R-tree, the R*-tree, and the VAM-split bulk loaded R*-tree. We explored 2 non-hierarchical structures, the grid file and the VA-file. For each of these index types, we modified code to include an optimization function that uses CP to traverse the index structure. We applied Algorithm 1 for hierarchical structures and Algorithm 2 for the non-hierarchical structures. We added a new set of 50000 uniform random data points within the 5-dimension hypercube and built each of the index structures over this data set.

We ran the set of random functions over each structure and measured the number of partitions that we retrieve data from. For the hierarchical structures these are leaf partitions and for the non-hierarchical structures they are grid cells. For each structure we try to keep the number of leaf partitions and grid cells relatively close to each other. For the hierarchical structures, we apply a maximum node capacity of 5, which yields an average fill rate of about 4 objects per node. This yields 12000 to 13000 leafs for each of the structures. For the grid file, we split each dimension equally in 7 intervals, which yields about 16000 grids. VA-files are built over bitstring identifiers, so we used the nearest power of 2, which is 8 for the number of splits per dimension. This results in about 32000 cells.

Figure 11 shows results over the 100 functions. It shows the minimum, maximum, and average number of leafs or grids explored to guarantee discovery of the optimal answer over the set of functions. Each index yielded the same correct optimal data point for each function, and these optimal answers were scattered all over the data space. Each structure performed well for these optimization queries, the average ratio of the data points accessed is below 0.0025 for all the access structures. As an example, the R-tree accesses an average of 28.1/12000 leafs for an access ratio of 0.0023. Even the worst performing structure over the worst case function still prunes over 99% of the search space.

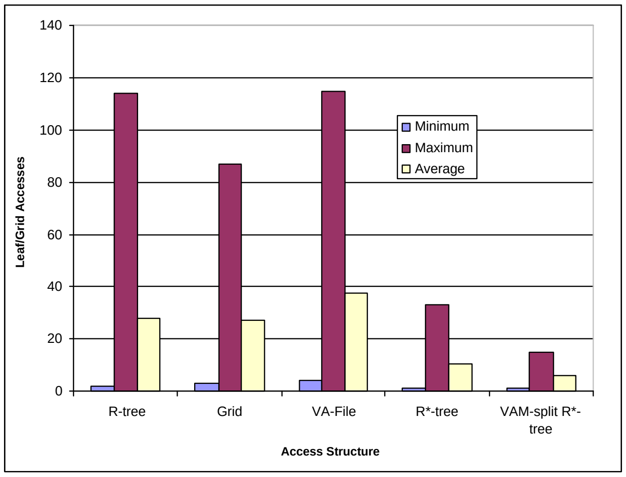

Figure 11: Accesses vs. Index Type, Optimizing Random Function, 5-D Uniform Random, 50k Points

Results between hierarchical access structures are as expected. The R*-tree looks to avoid some of the partition overlaps produced by the R-tree insertion heuristics and it shows better performance. The VAM-split R*-tree bulk loads the data into the index and can avoid more partition overlaps than an index built using dynamic insertion. As expected, this yields better performance than the dynamically constructed R*-tree.

While the VA-file results in more raw grid accesses than the grid-file, it actually has a better grid access ratio. The VA-file accesses on average a ratio of 0.0012 of its grids while the grid-file accesses on average a ratio of 0.0017 of its grids. The access ratio improvement is a result of the greater resolution for these particular VA-file and grid file constructions.

If we continued this experiment for higher dimensions, the non-hierarchical structures would outperform the hierarchical structures because the partition overlaps become more pronounced and unavoidable in hierarchical structures at higher dimensionality.

Incorporating Problem Constraints during Query Processing The proposed framework processes queries in conjunction with any set of convex problem constraints. We compare the proposed framework to alternative methods for handling constraints in optimization query processing. For a constrained NN type query we could process the query according to traditional NN techniques and when we find a result, check it against the constraints until we find a point that meets the constraints. Alternatively, we could perform a query on the constraints themselves, and then perform distance computations on the result set and select the best one. Figure 12 shows the number of pages accessed using our technique compared to these two alternatives as the constrained area varies. Constrained kNN queries were performed over the 8-D R*-tree built for the Color Histogram data set.

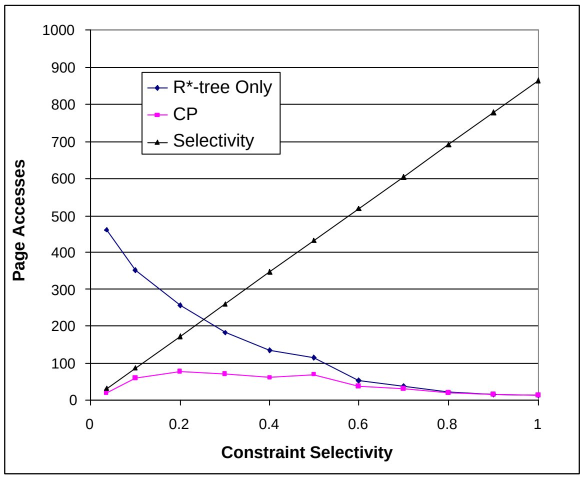

Figure 12: Pages Accesses vs. Constraints, NN Query, R*-Tree, 8-D Histogram, 100k Points

In the figure, the R*-tree Only line shows the number of page accesses required when we use traditional NN techniques, and then check if the result meets the constraints. As the problem becomes less constrained, the more likely a discovered NN will fall within the constraints, and we can terminate the search. The Selectivity line is a lower bound on the number of pages that would be read to obtain the points within the constraints. Clearly, as the problem becomes more constrained, the fewer pages will need to be read to find potential answers. The CP line shows the results for our framework, which processes original problem constraints as part of the search process and does not further process any partitions that do not intersect constraints.

We demonstrate better results with respect to page accesses except in cases where the constraints are very small (and we can prune much of the space by performing a query over the constraints first) or when the NN problem is not constrained (where our framework reduces to the traditional unconstrained NN problem). When there are problem constraints, we prune search paths that can not lead to feasible solutions. Our search drills down to the leaf partition that either contains the optimal answer within the problem constraints, or yields the best potential answer within constraints.

7. CONCLUSIONS

In this paper we present a general framework to model optimization queries. We present a query processing framework to answer optimization queries in which the optimization function is convex, query constraints are convex, and the underlying access structure is made up of convex partitions. We provide specific implementations of the processing framework for hierarchical data partitioning and non-hierarchical space partitioning access structures. We prove the I/O optimality of these implementations. We experimentally show that the framework can be used to answer a number of different types of optimization queries including nearest neighbor queries, linear function optimization, and non-linear function optimization queries.

The ability to handle general optimization queries within a single framework comes at the price of increasing the computational complexity when traversing the access structure. We show a slight increase in CPU times in order to perform convex programming based traversal of access structures in comparison with equivalent non-weighted and unconstrained versions of the same problem.

Since constraints are handled during the processing of partitions in our framework, and partitions and search paths that do not intersect with problem constraints are pruned during the access structure traversal, we do not need to examine points that do not intersect with the constraints. Furthermore, we only access points in partitions that could potentially have better answers than the actual optimal database answers. This results in a significant reduction in required accesses compared to alternative methods in the cases where the inclusion of constraints makes these alternatives less effective.

The proposed framework offers I/O-optimal access of whatever access structure is used for the query. This means that given an access structure we will only read points from partitions that have minimum objective function values lower than the k^{th} actual best answer. Because the framework is built to best utilize the access structure, it captures the benefits as well as the flaws of the underlying structure. Essentially, it demonstrates how well the particular access structure isolates the optimal contour within the problem constraints to access structure partitions. If the optimal contour and problem constraints can be isolated to a few leaf partitions, the access structure will yield nice results. However, if the optimal contour crosses many partitions, the performance will not be as good. As such, the framework can be used to measure page access performance associated with using different indexes and index types to answer certain classes of optimization queries, in order to determine which structures can most effectively answer the optimization query type. Database researchers and administrators can use this technique as a benchmarking tool to evaluate the performance of a wide range of index structures available in the literature.

To use this framework, one does not need to know the objective function, weights, or constraints in advance. The system does not need to compute a queried function for all data objects in order to find the optimal answers. The technique provides optimization of arbitrary convex functions, and does not incur a significant penalty in order to provide this generality. This makes the framework appropriate for applications and domains where a number of different functions are being optimized or when optimization is being performed over different constrained regions and the exact query parameters are not known in advance.

8. REFERENCES

[1] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. *PODS*, pages 78–86, 1997.

[2] Christian Bohm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.

[3] S. Boyd and L. Vandenberghe. *Convex Optimization.* Cambridge University Press, New York, NY. USA, 2004.

[4] Y-C. Chang, L. Bergman, V. Castelli, C-S. Li, M-L. Lo, and R. J. Smith. The onion technique: indexing for linear optimization queries. *ACM SIGMOD*, 2000.

[5] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 2003.

[6] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi. Vector approximation based indexing for non-uniform high dimensional data sets. In *CIKM '00*.

[7] Volker Gaede and Oliver Gunther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[8] A. Gionis, P. Indyk, and R. Motwani. Similarity searching in high dimensions via hashing. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 518–529, Edinburgh, Sootland, UK, September 1999.

[9] V.M. Gornitz, R.C. Daniels, T.W. White, and K.R. Birdwell. The development of a coastal risk assessment database: Vulnerability to sea-level rise in the u.s. southeast. *Journal of Coastal Research*, 1994.

[10] M. Grant, S. Boyd, and Y. Ye. *Disciplined convex programming.* Kluwer (Nonconvex Optimization and its Applications series), Dordrecht, 2005. In press.

[11] Gisli R. Hjaltason and Hanan Samet. Ranking in spatial databases. In *Symposium on Large Spatial Databases*, pages 83–95, 1995.

[12] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD Conference*, 2001.

[13] Becla. J. and D. Wang. Lessons learned from managing a petabyte. In *CIDR*, pages 70–83, 2005.

[14] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *SIGMOD*, 2000.

[15] M. S. Lobo. *Robust and Convex Optimization with Applications in Finance*. PhD thesis, The Department of Electrical Engineering, Stanford University, March 2000.

[16] B. S. Manjunath. Airphoto dataset. http://vivaldi.ece.ucsb.edu/Manjunath/research.htm, May 2000.

[17] B. S. Manjunath and W. Y. Ma. Texture features for browsing and retrieval of image data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(8):837–842, August 1996.

[18] R. P. Mount. *The Office of Science Data-Management Challenge*. DOE Office of Advanced Scientific Computing Research, March–May 2004.

[19] Y. Nesterov and A. Nemirovsky. A general approach to polynomial-time algorithms design for convex programming. Technical report, Centr. Econ. & Math. Inst., USSR Acad. Sci., Moscow, USSR, 1988.

[20] Y. Nesterov and A. Nemirovsky. *Interior-Point Polynomial Algorithms in Convex Programming*. SIAM, Philadelphia, PA 19101, USA, 1994.

[21] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, and P. Yanker. The QBIC project: Querying images by content using color, texture and shape. In *Proc. of the SPIE Conf. 1908 on Storage and Retrieval for Image and Video Databases*, volume 1908, pages 173–187, February 1993.

[22] N. Roussopoulos, S. Kelly, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, 1995.

[23] SciDAC. Scientific data management center. http://sdm.lbl.gov/sdmcenter/, 2002.

[24] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. *VLDB*, 2004.

[25] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, 1998.

[26] Z. Zhang, S. Hwang, K. C.-C. Chang, M. Wang, C. Lang, and Y. Chang. Boolean + ranking: Querying a database by k-constrained optimization. In *SIGMOD*, 2006.