IMPROVING QUERY PERFORMANCE THROUGH APPLICATION-DRIVEN PROCESSING AND RETRIEVAL

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Michael Albert Gibas, B.S., M.S.

* * * * *

The Ohio State University

2008

Dissertation Committee:
Hakan Ferhatosmanoglu, Adviser
Hui Fang
Atanas Rountev

Approved by

Adviser
Graduate Program in
Computer Science and
Engineering
The proliferation of massive data sets across many domains and the need to gain meaningful insights from these data sets highlight the need for advanced data retrieval techniques. Because I/O cost dominates the time required to answer a query, sequentially scanning the data and evaluating each data object against query criteria is not an effective option for large data sets. An effective solution should require reading as small a subset of the data as possible and should be able to address general query types. Access structures built over single attributes also may not be effective because 1) they may not yield performance that is comparable to that achievable by an access structure that prunes results over multiple attributes simultaneously and 2) they may not be appropriate for queries with results dependent on functions involving multiple attributes. Indexing a large number of dimensions is also not effective, because either too many subspaces must be explored or the index structure becomes too sparse at high dimensionalities. The key is to find solutions that allow for much of the search space to be pruned while avoiding this ‘curse of dimensionality’. This thesis pursues query performance enhancement using two primary means 1) processing the query effectively based on the characteristics of the query itself and 2) physically organizing access to data based on query patterns and data characteristics. Query performance enhancements are described in the context of several novel applications
including 1) Optimization Queries, which presents an I/O-optimal technique to answer queries when the objective is to maximize or minimize some function over the data attributes, 2) High-Dimensional Index Selection, which offers a cost-based approach to recommend a set of low dimensional indexes to effectively address a set of queries, and 3) Multi-Attribute Bitmap Indexes, which describes extensions to a traditionally single-attribute query processing and access structure framework that enables improved query performance.
ACKNOWLEDGMENTS

First of all, I would like to thank my wife, Moni Wood, for her love and support. She has made many sacrifices and expended much effort to help make this dream a reality. I would not have been to complete this dissertation without her.

I also would not have been able to accomplish this without the help of my family. I thank my mother Marilyn, my sister Susan and my grandmother Margaret for their support. I thank my late father Murray for providing an example to which I have always aspired. Moni’s family was also understanding and supportive of the travails of Ph.D research.

I want to express my appreciation to my adviser, Hakan Ferhatosmanoglu, for his guidance and encouragement over the years. He has been generous with his time, effort, and advice during my graduate studies. I am thankful for the technical discussions we have had and the opportunities he has given me to work on interesting projects.

Many people in the department have helped me to get to this point. I specifically want to thank my dissertation committee members, Dr. Atanas Rountev and Dr. Hui Fang, for their efforts and valuable comments on my research. I wish to thank Dr. Bruce Weide for his help and support during my studies, Dr. Rountev for my early exposure to research and paper writing, and Dr. Jeff Daniels for giving me the opportunity to work on an interdisciplinary geo-physics project.
Many friends and members of the Database Research Group provided valuable guidance, discussions, and critiques over the past several years. I want to especially thank Guadalupe Canahuate for her help and friendship during this time. Also thanks to her family for providing enough Dominican coffee to get through all the paper deadlines.

My Ph.D. research was supported by NSF grant NSF IIS -0546713.
VITA

May 10, 1969 .......................... Born - Omaha, Nebraska

1993 ........................................ B.S. Electrical Engineering,
Ohio State University, Columbus Ohio

2004 ........................................ M.S. Computer Science and Engineering,
Ohio State University, Columbus Ohio

1989-1992 .............................. Electrical Engineering Co-op,
British Petroleum,
Various Locations

1994-2002 .............................. Project Manager,
Acquisition Logistics Engineering,
Worthington, Ohio

2003-present ........................... Graduate Assistant,
Office Of Research,
Department of Computer Science and Engineering,
Department of Geology
Ohio State University,
Columbus, Ohio

PUBLICATIONS

Research Publications

Michael Gibas, Guadalupe Canahuate, Hakan Ferhatosmanoglu, Online Index Recommendations for High-Dimensional Databases Using Query Workloads. IEEE Transactions on Knowledge and Data Engineering (TDKE) Journal, February 2008 (Vol. 20, no.2), pp. 246-260


Guadalupe Canahuate, Michael Gibas, Hakan Ferhatosmanoglu, Update Conscious Bitmap Indexes. *International Conference on Scientific and Statistical Database Management (SSDBM)*, Banff, Canada, 2007/July

Guadalupe Canahuate, Michael Gibas, Hakan Ferhatosmanoglu, Indexing Incomplete Databases, *International Conference on Extending Database Technology (EDBT)*, Munich, Germany, 2006/March, pp. 884-901

Michael Gibas, Developing Indices for High Dimensional Databases Using Query Access Patterns, Masters Thesis, June 2004


**FIELDS OF STUDY**
Major Field: Computer Science and Engineering
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapters</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overall Technical Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Novel Database Management Applications</td>
<td>2</td>
</tr>
<tr>
<td>2. Modeling and Processing Optimization Queries</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Related Work</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Query Model Overview</td>
<td>10</td>
</tr>
<tr>
<td>2.3.1 Background Definitions</td>
<td></td>
</tr>
<tr>
<td>2.3.2 Common Query Types Cast into Optimization Query Model</td>
<td>13</td>
</tr>
<tr>
<td>2.3.3 Optimization Query Applications</td>
<td>15</td>
</tr>
<tr>
<td>2.3.4 Approach Overview</td>
<td>16</td>
</tr>
<tr>
<td>2.4 Query Processing Framework</td>
<td>19</td>
</tr>
<tr>
<td>2.4.1 Query Processing over Hierarchical Access Structures</td>
<td>21</td>
</tr>
<tr>
<td>2.4.2 Query Processing over Non-hierarchical Access Structures</td>
<td>26</td>
</tr>
<tr>
<td>2.4.3 Non-Covered Access Structures</td>
<td>28</td>
</tr>
<tr>
<td>2.5 I/O Optimality</td>
<td>28</td>
</tr>
<tr>
<td>2.6 Performance Evaluation</td>
<td>33</td>
</tr>
<tr>
<td>2.7 Conclusions</td>
<td>45</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Sample Optimization Objective Function and Constraints $F(\vec{\theta}, A)$: $(x - 1)^2 + (y - 2)^2$, $o(\text{min})$, $g_1(\vec{\alpha}, A): y - 6 \leq 0$, $g_2(\vec{\alpha}, A): -y + 3 \leq 0$, $h_1(\vec{\beta}, A): x - 5 = 0$, $k = 1$</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Optimization Query System Functional Block Diagram</td>
<td>20</td>
</tr>
<tr>
<td>2.3 Example of Hierarchical Query Processing</td>
<td>25</td>
</tr>
<tr>
<td>2.4 Cases of MBRs with respect to $R$ and optimal objective function contour</td>
<td>30</td>
</tr>
<tr>
<td>2.5 Cases of partitions with respect to $R$ and optimal objective function contour</td>
<td>32</td>
</tr>
<tr>
<td>2.6 Accesses, Random weighted kNN vs kNN, R*-tree, 8-D Histogram, 100k pts</td>
<td>35</td>
</tr>
<tr>
<td>2.7 CPU Time, Random weighted kNN vs kNN, R*-tree, 8-D Histogram, 100k pts</td>
<td>36</td>
</tr>
<tr>
<td>2.8 Accesses, Random weighted kNN vs kNN, VA-File, 60-D Sat, 100k pts</td>
<td>38</td>
</tr>
<tr>
<td>2.9 Page Accesses, Random Weighted Constrained k-LP Queries, R*-Tree, 8-D Color Histogram, 100k pts</td>
<td>40</td>
</tr>
<tr>
<td>2.10 Accesses vs. Index Type, Optimizing Random Function, 5-D Uniform Random, 50k Points</td>
<td>43</td>
</tr>
<tr>
<td>2.11 Pages Accesses vs. Constraints, NN Query, R*-Tree, 8-D Histogram, 100k Points</td>
<td>44</td>
</tr>
</tbody>
</table>
3.1 Index Selection Flowchart 61
3.2 Dynamic Index Analysis Framework 73
3.3 Costs in Data Object Accesses for Ideal, Sequential Scan, AutoAdmin, Naive, and the Proposed Index Selection Technique using Relaxed Constraints 81
3.4 Baseline Comparative Cost of Adaptive versus Static Indexing, random Dataset, synthetic Query Workload 85
3.5 Baseline Comparative Cost of Adaptive versus Static Indexing, random Dataset, clinical Query Workload 86
3.6 Baseline Comparative Cost of Adaptive versus Static Indexing, stock Dataset, hr Query Workload 87
3.7 Comparative Cost of Online Indexing as Support Changes, random Dataset, synthetic Query Workload 89
3.8 Comparative Cost of Online Indexing as Support Changes, stock Dataset, hr Query Workload 90
3.9 Comparative Cost of Online Indexing as Major Change Threshold Changes, random Dataset, synthetic Query Workload 92
3.10 Comparative Cost of Online Indexing as Major Change Threshold Changes, stock Dataset, hr Query Workload 93
3.11 Comparative Cost of Online Indexing as Multi-Dimensional Histogram Size Changes, random Dataset, synthetic Query Workload 94
4.1 A WAH bit vector. 102
4.2 Query Execution Example over WAH compressed bit vectors. 103
4.3 Query Execution Times vs Total Bitmap Length, ANDing 6 bitmaps, 2M entries 105
4.4 Query Execution Times vs Number of Attributes queried, Point Queries, 2M entries versus 106
4.5 Simple multiple attribute bitmap example showing the combination
two single attribute bitmaps. . . . . . . . . . . . . . . . . . . . . . . 108

4.6 Query Execution Time for each query type. . . . . . . . . . . . . . 125

4.7 Number of bitmaps involved in answering each query type. . . . . 126

4.8 Total size of the bitmaps involved in answering each query type. . . 127

4.9 Query Execution Times, Point Queries, Traditional versus Multi-Attribute
Query Processing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128

4.10 Query Execution Times as Query Ranges Vary, Landsat Data . . . . 129

4.11 Single and Multi-Attribute Bitmap Index Size Comparison . . . . . 130
CHAPTER 1

Introduction

If Edison had a needle to find in a haystack, he would proceed at once with the diligence of the bee to examine straw after straw until he found the object of his search... I was a sorry witness of such doings, knowing that a little theory and calculation would have saved him ninety per cent of his labor. - Nikola Tesla, 1931

In order to make better use of the massive quantities of data being generated by and for modern applications, it is necessary to find information that is in some respect interesting within the data. Two primary challenges of this task are specifying the meaning of interesting within the context of an application and searching the data set to find the interesting data efficiently. In the case of our proverbial haystack, we can think of each piece of straw as a data object and needles as a data objects that are interesting. While we could guarantee that we find the most interesting needle by examining each and every piece of straw, we would be much better served by limiting our search to a small region within the haystack.

1.1 Overall Technical Motivation

Analogous to our physical haystack example, the query response time during data exploration tasks using a database system is affected both by the physical organization
of the data and by the way in which the data is accessed. As such, query performance can be enhanced by traversing the data more effectively during query resolution or by organizing access to the data so that it is more suitable for the query. Depending on the application, queries can vary widely on a per query basis. Therefore, it is highly desirable to process generic queries at run-time in a way that is effective for each particular query. Physically reorganizing data or data access structures is time consuming and therefore should be performed for a set of queries rather than by trying to optimize an organization for a single query.

This thesis targets query performance enhancement through two sources 1) run-time query processing such that the search paths and intermediate operations during query resolution are appropriate for a given generic query, 2) data organization such that the access structures available during query resolution are appropriate for the overall query patterns and data distribution and allow for significant data space pruning during query resolution. Query performance enhancement is explored using three introduced applications, motivated and described in the next section.

1.2 Novel Database Management Applications

Optimization Queries. 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 [41]. 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 (our needle). Additionally, the object attributes queried, function parametric values, and data constraints
can vary on a per-query basis. 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 solution of each query type, (ii) take advantage of query constraints to prune search space, and (iii) maintain efficient performance when the scoring criteria or optimization function changes.

A general optimization model and query processing framework are proposed for optimization queries, an important and significant subset of data exploration tasks. The model covers those queries where some function is being minimized or maximized over object attributes, possibly within a set of constraints. The query processing framework can be used in those cases where the function being optimized and the set of constraints is convex. For access structures where the underlying partitions are convex, the query processing framework is used to access the most promising partitions and data objects and is proven to be I/O-optimal.

**High-Dimensional Index Selection.** With respect to the physical organization of the data, access structures or indexes that cover query attributes can be used to prune much of the data space and greatly reduce the required number of page accesses to answer the query. However, due to the oft-cited curse of dimensionality, indexes built over many dimensions actually perform worse than sequential scan. This problem can be addressed by using sets of lower dimensional indexes that effectively answer a large portion of the incoming queries. This approach can work well if the characteristics of future queries are well known, but in general this is not the case,
and an index set built based on historical queries may be completely ineffective for new queries.

In the case of physical organization, query performance is affected by creating a number of low dimension indexes using the historical and incoming query workloads, and data statistics. Historical query patterns are data mined to find those attribute sets that are frequently queried together. Potential indexes are evaluated to derive an initial index set recommendation that is effective in reducing query cost for a large portion of queries such that the set of indexes is within some indexing constraint. A set of indexes is obtained that in effect reduces the dimensionality of the query search space, and is effective in answering queries. A control feedback mechanism is incorporated so that the performance of newly arriving queries can be monitored, and if the current index set is no longer effective for the current query patterns, the index set can be changed. This flexible framework that can be tuned to maximize indexing accuracy or indexing speed which allows it to be used for the initial static index recommendation and also online dynamic index recommendation.

**Multi-Attribute Bitmap Indexes.** The nature of the database management system used and the way in which data is physically accessed also affects query execution times. Bitmap indexes have been shown to be effective as an data management system for many domains such a data warehousing and scientific applications. This owes to the fact that they are a column-based storage structure and only the data needed to address a query is ever retrieved from disk and that their base operations are performed using hardware supported fast bitwise operations. Static bitmap indexes have traditionally built over each attribute individually. The query model for selection queries is simple and involves ORing together bitcolumns that match query
criteria within an attribute and ANDing inter-attribute intermediate results. As a result, the bitmap index structure is limited to those domains that it is naturally suited for without change. As well query resolution does not take advantage of potential multi-attribute pruning and possible reductions in the number of binary operations.

Multi-attribute bitmap indexes are introduced to extend the functionality and applicability of traditional bitmap indexes. For many queries, query execution times are improved by reducing size of bitmaps used in query execution and by reducing the number of binary operations required to resolve the query without excessive overhead for queries that do not benefit from multi-attribute pruning. A query processing model where multi-attribute bitmaps are available for query resolution is presented. Methods to determine attribute combination candidates are provided.
CHAPTER 2

Modeling and Processing Optimization Queries

The most exciting phrase to hear in science, the one that heralds new discoveries, is not ‘Eureka!’ (I found it!) but ‘That’s funny ...’ - Isaac Asimov

One challenge associated with data management is determining a cost-effective way to process a given query. Given the time expense of I/O, minimizing the I/O required to answer a query is a primary strategy in query optimization. Scientific discovery and business intelligence activities are frequently exploring for data objects that are in some way special or unexpected. Such data exploration can be performed by iteratively finding those data objects that minimize or maximize some function over the data attributes. While the answer to a given query can be found by computing the function for all data objects and finding those objects that yield the most extreme values, this approach may not be feasible for large data sets. The optimization query framework presented in this chapter provides a method to generically handle an important broad class of queries, while ensuring that, given an access structure, the query is I/O-optimally processed over that structure.
2.1 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 [41]. 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) [48] and its variants [37], top-k queries [21, 51], 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 proactively 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 are provided in Section 2.3.3.

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.

### 2.2 Related Work

There are few published techniques that cover some instances of model-based optimization queries. One is the “Onion technique” [14], 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 [32] 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 [60] 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.

2.3 Query Model Overview

2.3.1 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 \( R(a_1, a_2, \ldots, a_n) \), 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 2.1 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 points \( b \) and \( d \) represent the best and worst point within the constraints for the objective function respectively. Point \( c \) could be the actual point in the database that meets the constraints and minimizes the objective function.

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 chapter.
Figure 2.1: Sample Optimization Objective Function and Constraints $F(\vec{\theta}, A) : (x - 1)^2 + (y - 2)^2$, $o(\text{min})$, $g_1(\vec{\alpha}, A) : y - 6 \leq 0$, $g_2(\vec{\alpha}, A) : -y + 3 \leq 0$, $h_1(\vec{\beta}, A) : x - 5 = 0$, $k = 1$

**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.

2.3.2 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) [27]. 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 $(a_0^1, a_0^2, \ldots, a_0^n)$ 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

$$WNN(a_1, a_2, \ldots, a_n) = \sqrt{w_1(a_1 - a_0^1)^2 + w_2(a_2 - a_0^2)^2 + \ldots + w_n(a_n - a_0^n)^2},$$

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

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

\[
g_i = l_i - a_i <= 0, g_{n+i} = a_i - u_i <= 0, i = 1, 2, \ldots, n
\]

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 \(l \leq 3a_1 - 2a_2 \leq u\)
2.3.3 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 opti-
mization 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.

### 2.3.4 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 [42], it is stated that a CP problem can be very efficiently solved with polynomial-time algorithms. The readers can refer to [43] 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 [38]. 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.” ([10] 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 ([43]). 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.
2.4 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 2.2 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.

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 2.4.1 and non-hierarchical structures in 2.4.2.

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 2.5 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.

### 2.4.1 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 [25, 9].
The algorithm is analogous to existing optimal NN processing [31], 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
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: $b[1]...b[k] \leftarrow \text{null}.$
2: $F(b[1])...F(b[k]) \leftarrow +\infty$
3: $L \leftarrow (\text{root})$
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 $P_{\text{child}}$ of $P$
11: $\minObjVal = \text{CPSolve}(OF,P_{\text{child}},C)$
12: if $\minObjVal < F(b[k])$
13: insert $P_{\text{child}}$ 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.

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 2.3 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 work-
list 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

24
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 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.
2.4.2 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 [25, 9, 53, 23].

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 [53] 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 $v$ in 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.

Algorithm 2: I/O Optimal Query Processing over Non-hierarchical Structures.
2.4.3 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) [25, 9]. 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.

2.5 I/O Optimality

In this section, we will show that the proposed query processing technique achieves optimal I/O for each implementation in Section 2.4. 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 [5], 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 2.4 shows different cases of the various position relations between a partition, $R$ and the optimal contour under 2 dimensions. $a$ is an optimal data point in the data base. We will use $\text{minObjVal}(\text{Partition}, R)$ 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 $R$. 
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 $\text{minObjVal}(A, R) = \text{minObjVal}(C, R) = +\infty$. B is pruned because $\text{minObjVal}(B, R) > F(a)$ and E will be accessed earlier than B in the algorithm because $\text{minObjVal}(E, R) < \text{minObjVal}(B, R)$. 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 $CP_0, CP_1, \ldots, CP_{k-1}$. Because the objective function is convex, we have

\[
F(a) \geq \text{minObjVal}(CP_0, R) \geq \text{minObjVal}(CP_1, R) \geq \ldots \\
\geq \text{minObjVal}(CP_k, R)
\]
and

\[ \minObjVal(D, R) > F(a) \geq \minObjVal(CP_0, R) \geq \minObjVal(CP_1, R) \geq \ldots \geq \minObjVal(CP_k, R) \]

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 \( F(a) \), 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 2.5 shows different possible cases of the partition position. Partitions \( A, B, C, D, \) and \( E \) are the same as defined for Figure 2.4.
Figure 2.5: Cases of partitions with respect to $R$ and optimal objective function contour

Without loss of generality, assume that only these five partitions contain some data points, and partition $E$ contains an optimal data point, $a$, 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 $\text{minObjVal}(E, R) < \text{minObjVal}(B, R)$), 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, $\text{minObjVal}(D, R) > \text{minObjVal}(E, R)$. 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., $F(a) \geq \text{minObjVal}(D, R)$, which also means the virtual solution corresponding to ($\text{minObjVal}(D, R)$) is contained by the optimal contour (because of the convexity of function $F$). Therefore, the virtual solution corresponding to ($\text{minObjVal}(D, R)$) $\in D \cap R \cap$ 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 the assumption that partition $D$ does not intersect the intersection of $R$ and the optimal contour.

2.6 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 solution, 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 que-ries 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 [40]. These datasets are widely used for performance evaluation of index structures and similarity search algorithms [39, 26]. We used a clinical trials patient data set and a stock price data set to perform real world optimization queries.

**Weighted kNN Queries** Figure 2.6 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,
Figure 2.6: Accesses, Random weighted kNN vs kNN, R*-tree, 8-D Histogram, 100k pts

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 2.7 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.

Figure 2.7: CPU Time, Random weighted kNN vs kNN, R*-tree, 8-D Histogram, 100k pts
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 queries. For this reason, we perform similar experiments for high-dimensional data using VA-files.

Figure 2.8 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.

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
Figure 2.8: Accesses, Random weighted kNN vs kNN, VA-File, 60-D Sat, 100k pts 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 Linear Optimization Queries** We also explore Constrained Weighted-Linear Optimization queries. Figure 2.9 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 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 2.3.3. 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 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 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 $\sum_{i=1}^{5} (\text{random()} \times x_i^2 - \text{random()} \times x_i)$, 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 using the algorithm appropriate for the structure. 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 objects in the partitions that we retrieve. 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 partitions relatively close to each other. The number of partitions for the hierarchical structures is between 12000 and 13000, the grid file 16000, and the VA-File 32000.

Figure 2.10 shows results over the 100 functions. It shows the minimum, maximum, and average object access ratio 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 throughout 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. Even the worst performing structure over the worst case function still prunes over 99% of the search space.

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.

The VA-File has greater resolution than the grid-file in this particular test, and therefore achieves better access ratios.

**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 2.11 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 2.11: Pages Accesses vs. Constraints, NN Query, R*-Tree, 8-D Histogram, 100k Points

In the figure, the R*-tree 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.

2.7 Conclusions

In this chapter 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 implemen-
tations. 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.
CHAPTER 3

Online Index Recommendations for High-Dimensional Databases using Query Workloads

When solving problems, dig at the roots instead of just hacking at the leaves. - Anthony J. D’Angelo.

In order to effectively prune search space during query resolution, access structures that are appropriate for the query must be available to the query processor at query run-time. Appropriate access structures should match well with query selection criteria and allow for a significant reduction in the data search space. This chapter describes a framework to determine a meaningful set of appropriate access structures for a set of queries considering query cost savings estimated from both query attributes and data characteristics. Since query patterns can change over time, rendering statically determined access structures ineffective, a method for monitoring performance and recommending access structure changes is provided.

3.1 Introduction

An increasing number of database applications, such as business data warehouses and scientific data repositories, deal with high dimensional data sets. As the number of dimensions/attributes and overall size of data sets increase, it becomes essential to efficiently retrieve specific queried data from the database in order to effectively utilize
the database. Indexing support is needed to effectively prune out significant portions of the data set that are not relevant for the queries. Multi-dimensional indexing, dimensionality reduction, and RDBMS index selection tools all could be applied to the problem. However, for high-dimensional data sets, each of these potential solutions has inherent problems.

To illustrate these problems, consider a uniformly distributed data set of 1,000,000 data objects with several hundred attributes. Range queries are consistently executed over five of the attributes. The query selectivity over each attribute is 0.1, so the overall query selectivity is $1/10^5$ (i.e. the answer set contains about 10 results). An ideal solution would allow us to read from disk only those pages that contain matching answers to the query. We could build a multi-dimensional index over the data set so that we can directly answer any query by only using the index. However, performance of multi-dimensional index structures is subject to Bellman’s curse of dimensionality [4] and degrades rapidly as the number of dimensions increases. For the given example, such an index would perform much worse than sequential scan.

Another possibility would be to build an index over each single dimension. The effectiveness of this approach is limited to the amount of search space that can be pruned by a single dimension (in the example the search space would only be pruned to 100,000 objects).

For data-partitioning indexes such as the R-tree family of indexes, data is placed in a partition that contains the data point and could overlap with other partitions. To answer a query, all potentially matching search paths must be explored. As the dimensionality of the index increases, the overlaps between partitions increase and
at high enough dimensions the entire data space needs to be explored. For space-
partitioning structures, where partitions do not overlap and data points are associated
with cells which contain them (e.g. grid files), the problem is the exponential explosion
of the number of cells. A 100 dimensional index with only a single split per attribute
results in $2^{100}$ cells. The index structure can become intractably large just to identify
the cells. This phenomenon is thoroughly described in [54].

Another possible solution would be to use some dimensionality reduction tech-
nique, index the reduced dimension data space, and transform the query in the
same way that the data was transformed. However, the dimensionality reduction
approaches are mostly based on data statistics, and perform poorly especially when
the data is not highly correlated. They also introduce a significant overhead in the
processing of queries.

Another possible solution is to apply feature selection to keep the most important
attributes of the data according to some criteria and index the reduced dimension-
ality space. However, traditional feature selection techniques are based on selecting
attributes that yield the best classification capabilities. Therefore, they also select
attributes based on data statistics to support classification accuracy rather than focusing
on query performance and workload in a database domain. As well, the selected
features may offer little or no data pruning capability given query attributes.

Commercial RDBMS’s have developed index recommendation systems to identify
indexes that will work well for a given workload. These tools are optimized for the
domains for which these systems are primarily employed and the indexes that the
systems provide. They are targeted towards lower dimension transactional databases
and do not produce results optimized for single high-dimensional tables.
Our approach is based on the observation that in many high dimensional database applications, only a small subset of the overall data dimensions are popular for a majority of queries and recurring patterns of dimensions queried occur. For example, Large Hadron Collider (LHC) experiments are expected to generate data with up to 500 attributes at the rate of 20 to 40 per second [45]. However, the search criterion is expected to consist of 10 to 30 parameters. Another example is High Energy Physics (HEP) experiments [49] where sub-atomic particles are accelerated to nearly the speed of light, forcing their collision. Each such collision generates on the order of 1-10MBs of raw data, which corresponds to 300TBs of data per year consisting of 100-500 million objects. The queries are predominantly range queries and involve mostly around 5 dimensions out of a total of 200.

We address the high dimensional database indexing problem by selecting a set of lower dimensional indexes based on joint consideration of query patterns and data statistics. This approach is also analogous to dimensionality reduction or feature selection with the novelty that the reduction is specifically designed for reducing query response times, rather than maintaining data energy as is the case for traditional approaches. Our reduction considers both data and access patterns, and results in multiple and potentially overlapping sets of dimensions, rather than a single set. The new set of low dimensional indexes is designed to address a large portion of expected queries and allow effective pruning of the data space to answer those queries.

Query pattern evolution over time presents another challenging problem. Researchers have proposed workload based index recommendation techniques. Their long term effectiveness is dependent on the stability of the query workload. However, query access patterns may change over time becoming completely dissimilar from the
patterns on which the index set were originally determined. There are many common reasons why query patterns change. Pattern change could be the result of periodic time variation (e.g. different database uses at different times of the month or day), a change in the focus of user knowledge discovery (e.g. a researcher discovery spawns new query patterns), a change in the popularity of a search attribute (e.g. current events cause an increase in queries for certain search attributes), or simply random variation of query attributes. When the current query patterns are substantially different from the query patterns used to recommend the database indexes, the system performance will degrade drastically since incoming queries do not benefit from the existing indexes. To make this approach practical in the presence of query pattern change, the index set should evolve with the query patterns. For this reason, we introduce a dynamic mechanism to detect when the access patterns have changed enough that either the introduction of a new index, the replacement of an existing index, or the construction of an entirely new index set is beneficial.

Because of the need to proactively monitor query patterns and query performance quickly, the index selection technique we have developed uses an abstract representation of the query workload and the data set that can be adjusted to yield faster analysis. We generate this abstract representation of the query workload by mining patterns in the workload. The query workload representation consists of a set of attribute sets that occur frequently over the entire query set that have non-empty intersections with the attributes of the query, for each query. To estimate the query cost, the data set is represented by a multi-dimensional histogram where each unique value represents an approximation of data and contains a count of the number of
records that match that approximation. For each possible index for each query, the estimated cost of using that index for the query is computed.

Initial index selection occurs by traversing the query workload representation and determining which frequently occurring attribute set results in the greatest benefit over the entire query set. This process is iterated until some indexing constraint is met or no further improvement is achieved by adding additional indexes. Analysis speed and granularity is affected by tuning the resolution of the abstract representations. The number of potential indexes considered is affected by adjusting data mining support level. The size of the multi-dimensional histogram affects the accuracy of the cost estimates associated with using an index for a query.

In order to facilitate online index selection, we propose a control feedback system with two loops, a fine grain control loop and a coarse control loop. As new queries arrive, we monitor the ratio of potential performance to actual performance of the system in terms of cost and based on the parameters set for the control feedback loops, we make major or minor changes to the recommended index set.

The contributions of this work can be summarized as follows:

1. Introduction of a flexible index selection technique designed for high-dimensional data sets that uses an abstract representation of the data set and query workload. The resolution of the abstract representation can be tuned to achieve either a high ratio of index-covered queries for static index selection or fast index selection to facilitate online index selection.

2. Introduction of a technique using control feedback to monitor when online query access patterns change and to recommend index set changes for high-dimensional data sets.
3. Presentation of a novel data quantization technique optimized for query workloads.

4. Experimental analysis showing the effects of varying abstract representation parameters on static and online index selection performance and showing the effects of varying control feedback parameters on change detection response.

The rest of the chapter is organized as follows. Section 3.2 presents the related work in this area. Section 3.3 explains our proposed index selection and control feedback framework. Section 3.4 presents the empirical analysis. We conclude in Section 3.5.

3.2 Related Work

High-dimensional indexing, feature selection, and DBMS index selection tools are possible alternatives for addressing the problem of answering queries over a subspace of high-dimensional data sets. As described below, each of these methods provides less than ideal solutions for the problem of fast high-dimensional data access. Our work differs from the related index selection work in that we provide a index selection framework that can be tuned for speed or accuracy. Our technique is optimized to take advantage of multi-dimensional pruning offered by multi-dimensional index structures. It takes into consideration both data and query characteristics and can be applied to perform real-time index recommendations for evolving query patterns.

3.2.1 High-Dimensional Indexing

A number of techniques have been introduced to address the high-dimensional indexing problem, such as the X-tree [6], and the GC-tree [28]. While these index
structures have been shown to increase the range of effective dimensionality, they still suffer performance degradation at higher index dimensionality.

### 3.2.2 Feature Selection

Feature selection techniques [7, 36, 29] are a subset of dimensionality reduction targeted at finding a set of untransformed attributes that best represent the overall data set. These techniques are also focused on maximizing data energy or classification accuracy rather than query response. As a result, selected features may have no overlap with queried attributes.

### 3.2.3 Index Selection

The index selection problem has been identified as a variation of the Knapsack Problem and several papers proposed designs for index recommendations [34, 55, 3, 24, 17, 13] based on optimization rules. These earlier designs could not take advantage of modern database systems’ query optimizer. Currently, almost every commercial Relational Database Management System (RDBMS) provides the users with an index recommendation tool based on a query workload and using the query optimizer to obtain cost estimates. A query workload is a set of SQL data manipulation statements. The query workload should be a good representative of the types of queries an application supports.

Microsoft SQL Server’s AutoAdmin tool [15, 16, 1] selects a set of indexes for use with a specific data set given a query workload. In the AutoAdmin algorithm, an iterative process is utilized to find an optimal configuration. First, single dimension candidate indexes are chosen. Then a candidate index selection step evaluates the queries in a given query workload and eliminates from consideration those candidate
indexes which would provide no useful benefit. Remaining candidate indexes are evaluated in terms of estimated performance improvement and index cost. The process is iterated for increasingly wider multi-column indexes until a maximum index width threshold is reached or an iteration yields no improvement in performance over the last iteration.

Costs are estimated using the query optimizer which is limited to considering those physical designs offered by the DBMS. In the case of SQL Server, single and multi-level B+-trees are evaluated. These index structures can not achieve the same level of result pruning that can be offered by an index technique that indexes multiple dimensions simultaneously (such as R-tree or grid file). As a result the indexes suggested by the tool often do not capture the query performance that could be achieved for multidimensional queries.

MAESTRO (METU Automated indEx Selection Tool)[19] was developed on top of Oracle’s DBMS to assist the database administrator in designing a complete set of primary and secondary indexes by considering the index maintenance costs based on the valid SQL statements and their usage statistics automatically derived using SQL Trace Facility during a regular database session. The SQL statements are classified by their execution plan and their weights are accumulated. The cost function computed by the query optimizer is used to calculate the benefit of using the index.

For DB2, IBM has developed the DB2Adviser [52] which recommends indexes with a method similar to AutoAdmin with the difference that only one call to the query optimizer is needed since the enumeration algorithm is inside the optimizer itself.
These commercial index selection tools are coupled to physical design options provided by their respective query optimizers and therefore, do not reflect the pruning that could be achieved by indexing multiple dimensions together.

3.2.4 Automatic Index Selection

The idea of having a database that can tune itself by automatically creating new indexes as the queries arrive has been proposed [35, 18]. In [35] a cost model is used to identify beneficial indexes and decide when to create or drop an index at runtime. [18] proposes an agent-based database architecture to deal with automatic index creation. Microsoft Research has proposed a physical design alerter [11] to identify when a modification to the physical design could result in improved performance.

3.3 Approach

3.3.1 Problem Statement

In this section we define the problem of index selection for a multi-dimensional space using a query workload.

A query workload $W$ consists of a set of queries that select objects within a specified subspace in the data domain. More formally, we define a workload as follows:

**Definition 4** A workload $W$ is a tuple $W = (D, DS, Q)$, where $D$ is the domain, $DS \subseteq D$ is a finite subset (the data set), and $Q$ (the query set) is a set of subsets of $DS$.

In our case, the domain is $\mathbb{R}^d$, where $d$ is the dimensionality of the dataset, the instance $DS$ is a set of $n$ tuples $t = \{(a_1, a_2, ..., a_d) | a_i \in \mathbb{R}\}$, and the query set $Q$ is a set of range queries.
Finding the answers to a query, when no index is present, reduces to scanning all the points in the dataset and testing whether the query conditions are met. In this scenario we can define the cost of answering the query as the time it takes to scan the dataset, i.e. the time to retrieve the data pages from disk. The assumption is that the time spend doing I/O dominates the time required to perform the simple bound comparisons. In the case that an index is present, the cost of answering the query can be lower. The index can identify a smaller set potential matching objects and only those data pages containing these objects need to be retrieved from disk. The degree to which an index prunes the potential answer set for a query determines its effectiveness for the query.

Our problem can be defined as finding a set of indexes $I$, given a multi-dimensional dataset $DS$, a query workload $W$, an optional indexing constraint $C$, an optional analysis time constraint $t_a$, that provides the best estimated cost over $W$. In the context of this problem an index is considered to be the set of attributes that can be used to prune subspace simultaneously with respect to each attribute. Therefore, attribute order has no impact on the amount of pruning possible.

The overall goal of this work is to develop a flexible index selection framework that can be tuned to achieve effective static and online index selection for high-dimensional data under different analysis constraints.

For static index selection, when no constraints are specified, we are aiming to recommend the set of indexes that yields the lowest estimated cost for every query in a workload for any query that can benefit from an index. In the case when a constraint is specified, either as the minimum number of indexes or a time constraint, we want to recommend a set of indexes within the constraint, from which the queries
can benefit the most. When there is a time-constraint, we need to automatically adjust the analysis parameters to increase the speed of analysis.

For online index selection, we are striving to develop a system that can recommend an evolving set of indexes for incoming queries over time, such that the benefit of index set changes outweighs the cost of making those changes. Therefore, we want an online index selection system that differentiates between low-cost index set changes and higher cost index set changes and also can make decisions about index set changes based on different cost-benefit thresholds.

3.3.2 Approach Overview

In order to measure the benefit of using a potential index over a set of queries, we need to be able to estimate the cost of executing the queries, with and without the index. Typically, a cost model is embedded into the query optimizer to decide on the query plan, whether the query should be answered using a sequential scan or using an existing index. Instead of using the query optimizer to estimate query cost, we conservatively estimate the number of matches associated with using a given index by using a multi-dimensional histogram abstract representation of the dataset. The histogram captures data correlations between only those attributes that could be represented in a selected index. The cost associated with an index is calculated based on the number of estimated matches derived from the histogram and the dimensionality of the index. Increasing the size of the multi-dimensional histogram enhances the accuracy of the estimate at the cost of abstract representation size.

While maintaining the original query information for later use to determine estimated query cost, we apply one abstraction to the query workload to convert each
query into the set of attributes referenced in the query. We perform frequent itemset
mining over this abstraction and only consider those sets of attributes that meet a
certain support to be potential indexes. By varying the support, we affect the speed
of index selection and the ratio of queries that are covered by potential indexes. We
further prune the analysis space using association rule mining by eliminating those
subsets above a certain confidence threshold. Lowering the confidence threshold im-
proves analysis time by eliminating some lower dimensional indexes from considera-
tion but can result in recommending indexes that cover a strict superset of the queried
attributes.

Our technique differs from existing tools in the method we use to determine the
potential set of indexes to evaluate and in the quantization-based technique we use
to estimate query costs. All of the commercial index wizards work in design time.
The Database Administrator (DBA) has to decide when to run this wizard and over
which workload. The assumption is that the workload is going to remain static over
time and in case it does change, the DBA would collect the new workload and run the
wizard again. The flexibility afforded by the abstract representation we use allows
it to be used for infrequent, index selection considering a broader analysis space or
frequent, online index selection.

In the following two sections we present our proposed solution for index selection
which is used for static index selection and as a building block for the online index
selection.
3.3.3 Proposed Solution for Index Selection

The goal of the index selection is to minimize the cost of the queries in the workload given certain constraints. Given a query workload, a dataset, the indexing constraints, and several analysis parameters, our framework produces a set of suggested indexes as an output. Figure 3.1 shows a flow diagram of the index selection framework. Table 3.1 provides a list of the notations used in the descriptions.

We identify three major components in the index selection framework: the initialization of the abstract representations, the query cost computation, and the index selection loop. In the following subsections we describe these components and the dataflow between them.

Initialize Abstract Representations

The initialization step uses a query workload and the dataset to produce a set of Potential Indexes $P$, a Query Set $Q$, and a Multi-dimensional Histogram $H$, according to the support, confidence, and histogram size specified by the user. The description of the outputs and how they are generated is given below.

- Potential Index Set $P$
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>Potential Set of Indexes, the set of attribute sets under consideration as a suggested indexes</td>
</tr>
<tr>
<td>$Q$</td>
<td>Query Set, a representation of the query workload, for each query. It consists of the attribute sets in $P$ that intersect with the query attributes, query ranges, and estimated query costs</td>
</tr>
<tr>
<td>$H$</td>
<td>Multi-Dimensional Histogram, used as a workload-optimized abstract representation of the data set</td>
</tr>
<tr>
<td>$S$</td>
<td>Suggested Indexes, the set of attribute sets currently selected as recommended indexes</td>
</tr>
<tr>
<td>$i$</td>
<td>attribute set currently under analysis</td>
</tr>
<tr>
<td>support</td>
<td>the minimum ratio of occurrence of an attribute in a query workload to be included in $P$</td>
</tr>
<tr>
<td>confidence</td>
<td>the maximum ratio of occurrence of an attribute subset to the occurrence of a set before the subset is pruned from $P$</td>
</tr>
<tr>
<td>histogram size</td>
<td>the number of bits used to represent a quantized data object</td>
</tr>
</tbody>
</table>

Table 3.1: Index Selection Notation List

The potential index set $P$ is a collection of attribute sets that could be beneficial as an index for the queries in the input query workload. This set is computed using traditional data mining techniques. Considering the attributes involved in each query from the input query workload to be a single transaction, $P$ consists of the sets of attributes that occur together in a query at a ratio greater than the input support. Formally, support of a set of attributes $A$ is defined as:

$$S_A = \sum_{i=1}^{n} \frac{1}{n} \begin{cases} 1 & \text{if } A \subseteq Q_i \\ 0 & \text{otherwise} \end{cases}$$

where $Q_i$ is the set of attributes in the $i^{th}$ query and $n$ is the number of queries.

For instance, if the input support is 10%, and attributes 1 and 2 are queried together in greater than 10 percent of the queries, then a representation of the set
of attributes \{1,2\} will be included as a potential index. Note that because a subset of an attribute set that meets the support requirement will also necessarily meet the support, all subsets of attribute sets meeting the support will also be included as a potential index (in the example above both the sets \{1\} and \{2\} will be included). As the input support is decreased, the number of potential indexes increases. Note that our particular system is built independently from a query optimizer, but the sets of attributes appearing in the predicates from a query optimizer log could just as easily be substituted for the query workload in this step.

If a set occurs nearly as often as one of its subsets, an index built over the subset will likely not provide much benefit over the query workload if an index is built over the attributes in the set. Such an index will only be more effective in pruning data space for those queries that involve only the subset’s attributes. In order to enhance analysis speed with limited effect on accuracy, the input confidence is used to prune analysis space. Confidence is the ratio of a set’s occurrence to the occurrence of a subset.

While data mining the frequent attribute sets in the query workload in determining \(P\), we also maintain the association rules for disjoint subsets and compute the confidence of these association rules. The confidence of an association rule is defined as the ratio that the antecedent (Left Hand Side of the rule) and consequent (Right Hand Side of the rule) appear together in a query given that the antecedent appears in the query. Formally, confidence of an association rule \{set of attributes \(A\}\rightarrow\{set of attributes \(B\)\}, where \(A\) and \(B\) are disjoint, is defined as:
\[
C_{A \rightarrow B} = \frac{\sum_{i=1}^{n} \left\{ \begin{array}{ll}
1 & \text{if } (A \cup B) \subseteq Q_i \\
0 & \text{otherwise}
\end{array} \right\}}{\sum_{i=1}^{n} \left\{ \begin{array}{ll}
1 & \text{if } A \subseteq Q_i \\
0 & \text{otherwise}
\end{array} \right\}}
\]

where \(Q_i\) is the set of attributes in the \(i^{th}\) query and \(n\) is the number of queries.

In our example, if every time attribute 1 appears, attribute 2 also appears then the confidence of \(\{1\} \rightarrow \{2\} = 1.0\). If attribute 2 appears without attribute 1 as many times as it appears with attribute 1, then the confidence \(\{2\} \rightarrow \{1\} = 0.5\). If we have set the \textit{confidence} input to 0.6, then we will prune the attribute set \(\{1\}\) from \(P\), but we will keep attribute set \(\{2\}\).

We can also set the confidence level based on attribute set cardinality. Since the cost of including extra attributes that are not useful for pruning increases with increased indexed dimensionality, we want to be more conservative with respect to pruning attribute subsets. The \textit{confidence} can contain more than one value depending on set cardinality.

While the apriori algorithm was appropriate for the relatively low attribute query sets in our domain, a more efficient algorithm such as the FP-Tree [30] could be applied if the attribute sets associated with queries are too large for the apriori technique to be efficient. While it is desirable to avoid examining a high-dimensional index set as a potential index, another possible solution in the case where a large number of attributes are frequent together would be to partition a large closed frequent itemset into disjoint subsets for further examination. Techniques such as CLOSET [44] could be used to arrive at the initial closed frequent itemsets.

- Query Set \(Q\)
The query set Q is the abstract representation of the query workload is initialized by associating the potential indexes that could be beneficial for each query with that query. These are the indexes in the potential index set P that share at least one common attribute with the query. At the end of this step, each query has an identified set of possible indexes for that query.

- Multi-Dimensional Histogram H

An abstract representation of the data set is created in order to estimate the query cost associated with using each query's possible indexes to answer that query. This representation is in the form of a multi-dimensional histogram H. A single bucket represents a unique bit representation across all the attributes represented in the histogram. The input histogram size dictates the number of bits used to represent each unique bucket in the histogram. These bits are designated to represent only the single attributes that met the input support in the input query workload. If a single attribute does not meet the support, then it can not be part of an attribute set appearing in P. There is no reason to sacrifice data representation resolution for attributes that will not be evaluated. The number of bits that each of the represented attributes gets is proportional to the log of that attribute's support. This gives more resolution to those attributes that occur more frequently in the query workload.

Data for an attribute that has been assigned b bits is divided into $2^b$ buckets. In order to handle data sets with uneven data distribution, we define the ranges of each bucket so that each bucket contains roughly the same number of points. The histogram is built by converting each record in the data set to its representation in bucket numbers. As we process data rows, we only aggregate the count of rows with each unique bucket representation because we are just interested in estimating
query cost. Note that the multi-dimensional histogram is based on a scalar quantizer
designed on data and access patterns, as opposed to just data in the traditional case.
A higher accuracy in representation is achieved by using more bits to quantize the
attributes that are more frequently queried.

For illustration, Table 3.2 shows a simple multi-dimensional histogram example.
This histogram covers 3 attributes and uses 1 bit to quantize attributes 2 and 3, and
2 bits to quantize attribute 1, assuming it is queried more frequently than the other
attributes. In this example, for attributes 2 and 3 values from 1-5 quantize to 0, and
values from 6-10 quantize to 1. For attribute 1, values 1 and 2 quantize to 00, 3 and
4 quantize to 01, 5-7 quantize to 10, and 8 and 9 quantize to 11. The .’s in the ‘value’
column denote attribute boundaries (i.e. attribute 1 has 2 bits assigned to it).

Note that we do not maintain any entries in the histogram for bit representa-
tions that have no occurrences. So we can not have more histogram entries than
records and will not suffer from exponentially increasing the number of potential
multi-dimensional histogram buckets for high-dimensional histograms.

<table>
<thead>
<tr>
<th>Sample Dataset</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>$A_2$</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Histogram</th>
<th>Value</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>00.0.0</td>
<td>00.0.0</td>
<td>2</td>
</tr>
<tr>
<td>00.0.1</td>
<td>01.0.0</td>
<td>1</td>
</tr>
<tr>
<td>01.1.0</td>
<td>01.1.1</td>
<td>1</td>
</tr>
<tr>
<td>10.1.0</td>
<td>11.0.0</td>
<td>2</td>
</tr>
<tr>
<td>11.0.1</td>
<td>11.0.1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.2: Histogram Example
Query Cost Calculation

Once generated, the abstract representations of the query set \( Q \) and the multi-dimensional histogram \( H \) are used to estimate the cost of answering each query using all possible indexes for the query. For a given query-index pair, we aggregate the number of matches we find in the multi-dimensional histogram looking only at the attributes in the query that also occur in the index (bits associated with other attributes are considered to be don’t cares in the query matching logic). To estimate the query cost, we then apply a cost function based on the number of matches we obtain using the index and the dimensionality of the index. At the end of this step, our abstract query set representation has estimated costs for each index that could improve the query cost. For each query in the query set representation, we also keep a current cost field, which we initialize to the cost of performing the query using sequential scan. At this point, we also initialize an empty set of suggested indexes \( S \).

• Cost Function

A cost function is used to estimate the cost associated with using a certain index for a query. The cost function can be varied to accurately reflect a cost model for the database system. For example, one could apply a cost function that amortized the cost of loading an index over a certain number of queries or use a function tailored to the type of index that is used. Many cost functions have been proposed over the years. For an R-Tree, which is the index type used for this work, the expected number of data page accesses is estimated in [22] by:

\[
A_{nn,mm,FBF} = \left( \sqrt[d]{\frac{1}{C_{eff}} + 1} \right)^d
\]
where $d$ is the dimensionality of the dataset and $C_{eff}$ is the number of data objects per disk page. However, this formula assumes the number of points $N$ approaches to infinity and does not consider the effects of high dimensionality or correlations.

A more recently proposed cost model is given in [8] where the expected number of pages accesses is determined as:

$$A_{r,em,ui}(r) = \left( 2r \cdot \sqrt[d]{\frac{N}{C_{eff}} + 1 - \frac{1}{C_{eff}}} \right)^d$$

where $r$ is the radius of the range query, $d$ is the dataset dimensionality, $N$ is the number of data objects, and $C_{eff}$ is the capacity of a data page.

While these published cost estimates can be effective to estimate the number of page accesses associated with using a multi-dimensional index structure under certain conditions, they have certain characteristics that make them less than ideal for the given situation. Each of the cost estimate formulas require a range radius. Therefore the formulas break down when assessing the cost of a query that is an exact match query in one or more of the query dimensions. These cost estimates also assume that data distribution is independent between attributes, and that the data is uniformly distributed throughout the data space.

In order to overcome these limitations, we apply a cost estimate that is based on the actual matches that occur over the multi-dimension histogram over the attributes that form a potential index. The cost model for R-trees we use in this work is given by

$$(d^{(d/2)} \times m)$$

where $d$ is the dimensionality of the index and $m$ is the number of matches returned for query matching attributes in the multi-dimensional histogram. Using actual matches
eliminates the need for a range radius. It also ties the cost estimate to the actual
data characteristics (i.e. incorporates both data correlation between attributes and
data distribution, while the published models will produce results that are dependent
only on the range radius for a given index structure). The cost estimate provided
is conservative in that it will provide a result that is at least as great as the actual
number of matches in the database.

By evaluating the number of matches over the set of attributes that match the
query, the multi-dimensional subspace pruning that can be achieved using different
index possibilities is taken into account. There is additional cost associated with
higher dimensionality indexes due to the greater number of overlaps of the hyperspaces
within the index structure, and additional cost of traversing the higher dimension
structure. A penalty is imposed on a potential index by the dimensionality term.
Given equal ability to prune the space, a lower dimensional index will translate into
a lower cost.

The cost function could be more complicated in order to more accurately model
query costs. It could model query cost with greater accuracy, for example by crediting
complete attribute coverage for coverage queries. It could also reflect the appropriate
index structures used in the database system, such as B+-trees. We used this partic-
ular cost model because the index type was appropriate for our data and query sets,
and we assumed that we would retrieve data from disk for all query matches.

Index Selection Loop

After initializing the index selection data structures and updating estimated query
costs for each potentially useful index for a query, we use a greedy algorithm that takes
into account the indexes already selected to iteratively select indexes that would be
appropriate for the given query workload and data set. For each index in the potential index set \( P \), we traverse the queries in query set \( Q \) that could be improved by that index and accumulate the improvement associated with using that index for that query. The improvement for a given query-index pair is the difference between the cost for using the index and the query’s current cost. If the index does not provide any positive benefit for the query, no improvement is accumulated. The potential index \( i \) that yields the highest improvement over the query set \( Q \) is considered to be the best index. Index \( i \) is removed from potential index set \( P \) and is added to suggested index set \( S \). For the queries that benefit from \( i \), the current query cost is replaced by the improved cost.

After each \( i \) is selected, a check is made to determine if the index selection loop should continue. The input indexing constraints provides one of the loop stop criteria. The indexing constraint could be any constraint such as the number of indexes, total index size, or total number of dimensions indexed. If no potential index yields further improvement, or the indexing constraints have been met, then the loop exits. The set of suggested indexes \( S \) contains the results of the index selection algorithm.

At the end of a loop iteration, when possible, we prune the complexity of the abstract representations in order to make the analysis more efficient. This includes actions such as eliminating potential indexes that do not provide better cost estimates than the current cost for any query and pruning from consideration those queries whose best index is already a member of the set of suggested indexes. The overall speed of this algorithm is coupled with the number of potential indexes analyzed, so the analysis time can be reduced by increasing the support or decreasing the confidence.
Different strategies can be used in selecting a best index. The strategy provided assumes a indexing constraint based on the number of indexes and therefore uses the total benefit derived from the index as the measure of index 'goodness'. If the indexing constraint is based on total index size, then benefit per index size unit may be a more appropriate measure. However, this may result in recommending a lower-dimensioned index and later in the algorithm a higher-dimensioned index that always performs better. The recommendation set can be pruned in order to avoid recommending an index that is non-useful in the context of the complete solution.

3.3.4 Proposed Solution for Online Index Selection

The online index selection is motivated by the fact that query patterns can change over time. By monitoring the query workload and detecting when there is a change on the query pattern that generated the existing set of indexes, we are able to maintain good performance as query patterns evolve. In our approach, we use control feedback to monitor the performance of the current set of indexes for incoming queries and determine when adjustments should be made to the index set. In a typical control feedback system, the output of a system is monitored and based on some function involving the input and output, the input to the system is readjusted through a control feedback loop. Our situation is analogous but more complex than the typical electrical circuit control feedback system in several ways:

1. Our system input is a set of indexes and a set of incoming queries rather than a simple input, such as an electrical signal.

2. The system output must be some parameter that we can measure and use to make decisions about changing the input. Query performance is the obvious
parameter to monitor. However, because lower query performance could be related to other aspects rather than the index set, our decision making control function must necessarily be more complex than a basic control system.

3. We do not have a predictable function to relate system input and output because of the non-determinism associated with new incoming queries. For example, we may have a set of attributes that appears in queries frequently enough that our system indicates that it is beneficial to create an index over those attributes, but there is no guarantee that those attributes will ever be queried again.

Control feedback systems can fail to be effective with respect to response time. The control system can be too slow to respond to changes, or it can respond too quickly. If the system is too slow, then it fails to cause the output to change based on input changes in a timely manner. If it responds too quickly, then the output overshoots the target and oscillates around the desired output before reaching it. Both situations are undesirable and should be designed out of the system.

Figure 3.2 represents our implementation of dynamic index selection. Our system input is a set of indexes and a set of incoming queries. Our system simulates and estimates costs for the execution of incoming queries. System output is the ratio of potential system performance to actual system performance in terms of database page accesses to answer the most recent queries. We implement two control feedback loops. One is for fine grain control and is used to recommend minor, inexpensive changes to the index set. The other loop is for coarse control and is used to avoid very poor system performance by recommending major index set changes. Each control feedback loop has decision logic associated with it.
System Input

The system input is made up of new incoming queries and the current set of indexes $I$, which is initialized to be the suggested indexes $S$ from the output of the initial index selection algorithm. For clarity, a notation list for the online index selection is included as Table 3.3.

System

The system simulates query execution over a number of incoming queries. The abstract representation of the last $w$ queries stored as $W$, where $w$ is an adjustable window size parameter. $W$ is used to estimate performance of a hypothetical set of indexes $I_{new}$ against the current index set $I$. This representation is similar to the one kept for query set $Q$ in the static index selection. In this case, when a new query $q$ arrives, we determine which of the current indexes in $I$ most efficiently answers
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I$</td>
<td>Current set of attribute sets used as indexes</td>
</tr>
<tr>
<td>$I_{new}$</td>
<td>Hypothetical set of attribute sets used as indexes</td>
</tr>
<tr>
<td>$w$</td>
<td>window size</td>
</tr>
<tr>
<td>$W$</td>
<td>abstract representation of the last $w$ queries</td>
</tr>
<tr>
<td>$q$</td>
<td>the current query under analysis</td>
</tr>
<tr>
<td>$P$</td>
<td>Current Potential Indexes, the set of attribute sets in consideration to be indexes before $q$ arrives</td>
</tr>
<tr>
<td>$P_{new}$</td>
<td>New Potential Indexes, the set of attribute sets in consideration to be indexes after $q$ arrives</td>
</tr>
<tr>
<td>$i_q$</td>
<td>the attribute set estimated to be the best index for query $q$</td>
</tr>
</tbody>
</table>

Table 3.3: Notation List for Online Index Selection

this query and replace the oldest query in $W$ with the abstract representation of $q$.

We also incrementally compute the attribute sets that meet the input *support* and *confidence* over the last $w$ queries. This information is used in the control feedback loop decision logic. The system also keeps track of the current potential indexes $P$, and the current multi-dimensional histogram $H$.

**System Output**

In order to monitor the performance of the system, we compare the query performance using the current set of indexes $I$ to the performance using a hypothetical set of indexes $I_{new}$. The query performance using $I$ is the summation of the costs of queries using the best index from $I$ for the given query. Consider the possible new indexes $P_{new}$ to be the set of attribute sets that currently meet the input *support* and *confidence* over the last $w$ queries. The hypothetical cost is calculated differently based on the comparison of $P$ and $P_{new}$, and the identified best index $i_q$ from $P$ or $P_{new}$ for the new incoming query:
1. \( P = P_{\text{new}} \) and \( i \) is in \( I \). In this case we bypass the control loops since we could do no better for the system by changing possible indexes.

2. \( P = P_{\text{new}} \) and \( i \) is not in \( I \). We recompute a new set of suggested indexes \( I_{\text{new}} \) over the last \( w \) queries. The hypothetical cost is the cost over the last \( w \) queries using \( I_{\text{new}} \).

3. \( P \neq P_{\text{new}} \) and \( i \) is in \( I \). In this case we bypass the control loops since we could do no better for the system by changing possible indexes.

4. \( P \neq P_{\text{new}} \) and \( i \) is not in \( I \). We traverse the last \( w \) queries and determine those queries that could benefit from using a new index from \( P_{\text{new}} \). We compute the hypothetical cost of these queries to be the real number of matches from the database. Hypothetical cost for other queries is the same as the real cost.

The ratio of the hypothetical cost, which indicates potential performance, to the actual performance is used in the control loop decision logic.

**Fine Grain Control Loop**

The fine grain control loop is used to recommend low cost, minor changes to index set. This loop is entered in case 2 as described above when the ratio of hypothetical performance to actual performance is below some input \textit{minor change threshold}. Then the indexes are changed to \( I_{\text{new}} \), and appropriate changes are made to update the system data structures. Increasing the input \textit{minor change threshold} causes the frequency of minor changes to also increase.
Coarse Control Loop

The coarse control loop is used to recommend more costly, but changes with greater impact on future performance to the index set. This loop is entered in case 4 as described above when the ratio of hypothetical performance to actual performance is below some input major change threshold. Then the static index selection is performed over the last $w$ queries, abstract representations are recomputed, and a new set of suggested indexes $I_{new}$ is generated. Appropriate changes are made to update the system data structures to the new situation. Increasing the input major change threshold increases the frequency of major changes.

3.3.5 System Enhancements

In the following subsections, we present two system enhancements that provide further robustness and scalability to our framework.

Self Stabilization

Control feedback systems can be either too slow or too responsive in reacting to input changes. In our application, a system that is slow to respond results in recommending useful indexes long after they first could have a positive effect. It could also fail to recommend potentially useful indexes if thresholds are set so that the system is insensitive to change. A system that is too responsive can result in system instability, where the system continuously adds and drops indexes.

The system performance and rate of index change can be monitored and used in order to tune the control feedback system itself. If the actual number of query results is much lower than the estimated cost using the recommended index set over a window of queries, this indicates a non-responsive system. When this condition is
detected, the system can be made more responsive by cutting the window size. This increases the probability that $P_{new}$ will be different from $P$, and we can reperform to analysis applying a reduced support. This gives us a more complete $P$ with respect to answering a greater portion of queries.

If the frequency of index change is too high with little or no improvement in query performance, an oversensitive system or unstable query pattern is indicated. We can reduce the sensitivity by increasing the window size or increasing the support level during recomputation of new recommended indexes.

**Partitioning the Algorithm**

The system can also be applied to environments where the database is partitioned horizontally or vertically at different locations. A solution at one extreme is to maintain a centralized index recommendation system. This would maintain the abstract representation and collect global query patterns over the entire system. Recommended indexes would be determined based on global trends. This approach would allow for the creation of indexes that maximize pruning over the global set of dimensions. However, it would not optimize query performance at the site level. A single set of indexes would be recommended for the entire system.

At the other extreme would be to perform the index recommendation at each site. In this case, a different abstract representation would be built based on the data at each specific site given requests to the data at that site. Indexes would be recommended based on the query traffic to the data at that site. This allows tight control of the indexes to reflect the data at each site. However, index-based pruning based on attributes and data at multiple locations would not be possible.
This approach also requires traffic to each site that contains potentially matching data.

A hybrid approach can provide the benefits of index recommendation based on global data and query patterns while optimizing the index set at different requesting locations. A global set of indexes can be centrally recommended based on a global abstract representation of the data set with low support thresholds (the centralized location will store a larger set of globally good indexes). The query patterns emanating from each site can be mined in order to find which of those global indexes would be appropriate to maintain at the local site, eliminating some traffic.

3.4 Empirical Analysis

3.4.1 Experimental Setup

Data Sets

Several data sets were used during the performance of experiments. The variation in data sets is intended to show the applicability of our algorithm to a wide range of data sets and to measure the effect that data correlation has on results. Data sets used include:

- *random* - a set of 100,000 records consisting of 100 dimensions of uniformly distributed integers between 0 and 999. The data is not correlated.

- *stocks* - a set of 6500 records consisting of 360 dimensions of daily stock market prices. This data is extremely correlated.
• *mlb* - a set of 33619 records of major league pitching statistics from between the years of 1900 and 2004 consisting of 29 dimensions of data. Some dimensions are correlated with each other, while others are not at all correlated.

**Analysis Parameters**

The effect of varying several analysis input parameters including support, multi-dimensional histogram size, and online indexing control feedback decision thresholds was analyzed. Unless otherwise specified, the *confidence* parameter for the experiments is 1.0.

**Query Workloads**

It is desirable to explore the general behavior of database interactions without inadvertently capturing the coupling associated with using a specific query history on the same database. Therefore, query workload files were generated by merging synthetic query histories and query histories from real-world applications with different data sets. Histories and data were merged by taking a random record from a data set and the numerical identifier of the attributes involved in the synthetic or historical query in order to generate a point query. So, if a historical query involved the 3rd and 5th attribute, and the *n*th record was randomly selected from the data set, a SELECT type query is generated from the data set where the 3rd attribute is equal to the value of the 3rd attribute of the *n*th record and the 5th attribute is equal to the value of 5th attribute of the *n*th record. This gives a query workload that reflects the attribute correlations within queries, and has a variable query selectivity. Unless otherwise stated each query in experiments is a point query with respect to
the attributes covered by the query. Attributes that are not covered by the query can be any value and still match the query.

These query histories form the basis for generating the query workloads used in our experiments:

1. synthetic - 500 randomly generated queries. The distribution of the queries over the first 200 queries is 20% involve attributes \{1,2,3,4\} together, 20% \{5,6,7\}, 20%, \{8,9\}, and the remaining queries involve between 1 and 5 attributes that could be any attribute. Over the last 300 queries, the distribution shifts to 20% covering attributes \{11,12,13,14\}, 20% \{15,16,17\}, 20% \{18,19\}, and the remaining 40% are between 1 to 5 attributes that could be any attribute.

2. clinical - 659 queries executed from a clinical application. The query distribution file has 64 distinct attributes.

3. hr - 35,860 queries executed from a human resources application. The query distribution file has 54 attributes. Due to the size of this query set, some initial portion of the queries are used for some experiments.

3.4.2 Experimental Results
Index Selection with Relaxed Constraints

Figure 3.3 shows how accurately the proposed static index selection technique can perform with relaxed analysis constraints. For this scenario, a low support value, 0.01, is used. There are no constraints placed on the number of selected indexes. And 256 bits are used for the multi-dimension histogram. Figure 3.3 shows the cost of performing a sequential scan over 100 queries using the indicated data sets, the estimated cost of using recommended indexes, and the true number of answers for
the set of queries. For comparative purposes, costs for indexes proposed by AutoAdmin and a naive algorithm are also provided. The naive algorithm uses indexes for the first $n$ most frequent itemsets in the query patterns, where $n$ is the number of indexes suggested by the proposed algorithm. Note that the cost for sequential scan is discounted to account for the fact that sequential access is significantly cheaper than random access. A factor of 10 is applied as the ratio of random access cost to sequential access cost. While the actual ratio of a given environment is variable, regardless of any reasonable factor used, the graph will show the cost of using our indexes much closer to the ideal number of page accesses than the cost of sequential scan.

Table 3.4 compares the proposed index selection algorithm with relaxed constraints against SQL Server’s index selection tool, AutoAdmin [15], using the data sets and query workloads indicated.
For the stock dataset using both the clinical and hr workloads, both algorithms suggest indexes which will improve all of the queries. Since the selectivity of these queries is low (the queries return a low number of matches using any index that contains a queried attribute), the amount of the query improvement will be very similar using either recommended index set. The proposed algorithm executes in less time and generates indexes which are more representative of the query patterns in the workload and allow for greater subspace pruning. For example, the most common query in the clinical workload is to query over both attributes 11 and 44 together. SQL Server’s index recommendations are all single-dimension indexes for all attributes that appear in the workload. However, our first index recommendation is a 2-dimension index built over attributes 11 and 44.

For the mlb dataset, SQL Server quickly recommended no indexes. Our index selection takes longer in this instance, but finds indexes that improve 87 % of the queries. These are all the queries that have selectivities low enough that an index can be beneficial. It should be noted that the indexes selected by AutoAdmin are appropriate based on the cost models for the underlying index structure used in

<table>
<thead>
<tr>
<th>Data Set/Workload</th>
<th>Tool</th>
<th>Analysis Time(s)</th>
<th>% Queries Improved</th>
<th>Number of Indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>stock/clinical</td>
<td>AutoAdmin</td>
<td>450</td>
<td>100</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Proposed</td>
<td>110</td>
<td>100</td>
<td>18</td>
</tr>
<tr>
<td>stock/hr</td>
<td>AutoAdmin</td>
<td>338</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Proposed</td>
<td>160</td>
<td>100</td>
<td>16</td>
</tr>
<tr>
<td>mlb/clinical</td>
<td>AutoAdmin</td>
<td>15</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Proposed</td>
<td>522</td>
<td>87</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 3.4: Comparison of proposed index selection algorithm with AutoAdmin in terms of analysis time and % queries improved
SQL Server. Since the underlying index type for SQL Server is B+-trees, which do not index multiple dimensions concurrently, the single dimension indexes that are recommended are the least costly possible indexes to use for the query set. For this particular experiment, a single dimensional index is not able to prune the subspace enough to make the application of that index worthwhile compared to sequential scan.

Effect of Support and Confidence

Table 3.5 presents results on analysis complexity and expected query improvement as support and confidence are varied. The results are shown for the stock data set over the clinical query workload. Results show the total number of query-index pairs analyzed over the query set in the index selection loop and the total estimated improvement in query performance in terms of data objects accessed over the query set as the index selection parameters vary. As confidence decreases, we maintain fewer potential indexes in \( P \) and need to analyze fewer attribute sets per index. This decreases analysis time but shows very little effect on overall performance.

Using a confidence level of 0\% is equivalent to using the maximal itemsets of attributes that meet support criteria as recommended indexes. For this example, the strategy yields nearly identical estimated cost although only 34-44\% of the query/index pairs need to be evaluated.

Baseline Online Indexing Results

A number of experiments were performed to demonstrate the effectiveness and characteristics of the adaptive system. In each of the experiments that show the performance of the adaptive system over a sequence of queries, an initial set of indexes is recommended based on the first 100 queries given the stated analysis parameters.
Table 3.5: Comparison of Analysis Complexity and Query Performance as support and confidence vary, stock dataset, clinical workload

<table>
<thead>
<tr>
<th>Support</th>
<th>Query/Index Pairs</th>
<th>Total Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidence</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>229</td>
<td>229</td>
</tr>
<tr>
<td>4</td>
<td>198</td>
<td>198</td>
</tr>
<tr>
<td>6</td>
<td>185</td>
<td>185</td>
</tr>
<tr>
<td>8</td>
<td>178</td>
<td>178</td>
</tr>
<tr>
<td>10</td>
<td>170</td>
<td>170</td>
</tr>
</tbody>
</table>

New queries are evaluated, and depending on the parameters of the adaptive system, changes are made to index set. These index set changes are considered to take place before the next query. The estimated cost of the last 100 queries given the indexes available at the time of the query are accumulated and presented. This demonstrates the evolving suitability of the current index set to incoming queries.

Figures 3.4 through 3.6 show a baseline comparison between using the query pattern change detection and modifying the indexes and making no change to the initial index set. A number of data set and query history combinations are examined. The baseline parameters used are 5% support, 128 bits for each multi-dimension histogram entry, a window size of 100, an indexing constraint of 10 indexes, a major change threshold of 0.9 and a minor change threshold of 0.95.

Figure 3.4 shows the comparison for the random data set using the synthetic query workload. At query 200, this workload drastically changes, and this is evident in the query cost for the static system. The performance gets much worse and does not get better for the static system. For the online system, the performance degrades a little.
when the query patterns are changing and then improve again once the indexes have changed to match the new query patterns.

The synthetic query workload was generated specifically to show the effect of a changing query pattern. Figure 3.5 shows a comparison of performance for the random data set using the clinical query workload. This real query workload also changes substantially before reverting back to query patterns similar to those on which the static index selection was performed. Performance for the static system degrades substantially when the patterns change until the point that they change back. The adaptive system is better able to absorb the change.
Figure 3.5: Baseline Comparative Cost of Adaptive versus Static Indexing, *random* Dataset, *clinical* Query Workload
Figure 3.6: Baseline Comparative Cost of Adaptive versus Static Indexing, stock Dataset, hr Query Workload

Figure 3.6 shows the comparison for the stock data set using the first 2000 queries of the hr query workload. The adaptive system shows consistently lower cost than the static system and in places significantly lower cost for this real query workload.

The effect of range queries was also explored. The random data set was examined using the synthetic workload using ranges instead of points. As the ranges for a given attribute increased from a point value to 30% selectivity, the cost saved for top 3 proposed indexes (which were the index sets [1,2,3,4], [5,6,7], and [8,9] for all ranges), the overall cost saved decreased. This is due to the increase in the size of the answer
set. When the range for each attribute reached 30%, no index was recommended because the result set became large enough that sequential scan was a more effective solution.

**Online Index Selection versus Parametric Changes**

Changing online index selection parameters changes the adaptive index selection in the following ways:

**Support** - decreasing the support level has the effect of increasing the potential number of times the index set will be recalculated. It also makes the recalculation of the recommended indexes themselves more costly and less appropriate for an online setting. Figure 3.7 shows the effect of changing support on the online indexing results for the random data set and synthetic query workload, while Figure 3.8 shows the effect on the stock data set using the first 600 queries of hr query workload. These graphs were generated using the baseline parameters and varying support levels. As expected, lower support levels translate to lower initial costs because more of the initial query set are covered by some beneficial index. For the synthetic query workload, when the patterns changed, but do not change again (e.g. queries 100-200 in Figure 3.7), lower support levels translated to better performance. However, for the hr query workload, the 10% support threshold yields better performance than the 6% and 8% thresholds for some query windows. The frequent itemsets change more frequently for lower support levels, and these changes dictate when decisions occur. These runs made decisions at points that turned out to be poor decisions, such as eliminating an index that ended up being valuable. Little improvement in cost performance is achieved between 4% support and 2% support. Over-sensitive control feedback can
Figure 3.7: Comparative Cost of Online Indexing as Support Changes, random Dataset, synthetic Query Workload

degrade actual performance, independent of the extra overhead that oversensitive control causes.

**Online Indexing Control Feedback Decision Thresholds** - increasing the thresholds decreases the response time of affecting change when query pattern change does occur. It also has the effect of increasing the number of times the costly reanalysis occurs. In a real system, one would need to balance the cost of continued poor performance against the cost of making an index set change. Figure 3.9 shows the effect of varying the major change threshold in the coarse control loop for the *random*
Figure 3.8: Comparative Cost of Online Indexing as Support Changes, *stock* Dataset, *hr* Query Workload
data set and synthetic query workload. Figure 3.10 shows the effect of changing the major change threshold for the stock data set and hr query workload. These graphs were generated using the baseline parameters (except that the random graph uses a support of 10%), and only varying the major change threshold in the coarse control loop. The major change threshold is varied between 0 and 1. Here, a value of 0 translates to never making a change, and a value of 1 means making a change whenever improvement is possible. The graphs show no real benefit once the major change threshold increases (and therefore frequency of major changes) beyond 0.6. Figure 3.9 shows that a value of 1 or 0.9 are best in terms of response time when a change occurs, but a value of 0.3 shows the best performance once the query patterns have stabilized. This indicates that this value should be carefully tuned based on the expected frequency of query pattern changes.

Multi-dimensional Histogram Size - increasing the number of bits used to represent a bucket in the multi-dimensional histogram improves analysis accuracy at the cost of histogram generation time and space. Experiments demonstrated that if the representation quality of the histogram was sufficient, very little benefit was achieved through greater resolution. Figure 3.11 shows an example of the effect of varying the multi-dimensional histogram size. Using a 32 bit size for each unique histogram bucket yields higher costs than by using greater histogram resolution. One reason for this is that artificially higher costs are calculated because of the lower histogram resolution. As well, some beneficial index changes are not recommended because of these overly conservative cost estimates.
Figure 3.9: Comparative Cost of Online Indexing as Major Change Threshold Changes, *random* Dataset, *synthetic* Query Workload
Figure 3.10: Comparative Cost of Online Indexing as Major Change Threshold Changes, stock Dataset, hr Query Workload
Figure 3.11: Comparative Cost of Online Indexing as Multi-Dimensional Histogram Size Changes, random Dataset, synthetic Query Workload
3.5 Conclusions

A flexible technique for index selection is introduced that can be tuned to achieve different levels of constraints and analysis complexity. A low constraint, more complex analysis can lead to more accurate index selection over stable query patterns. A more constrained, less complex analysis is more appropriate to adapt index selection to account for evolving query patterns. The technique uses a generated multi-dimension histogram to estimate cost, and as a result is not coupled to the idiosyncrasies of a query optimizer, which may not be able to take advantage of knowledge about correlations between attributes. Indexes are recommended in order to take advantage of multi-dimensional subspace pruning when it is beneficial to do so.

These experiments have shown great opportunity for improved performance using adaptive indexing over real query patterns. A control feedback technique is introduced for measuring performance and indicating when the database system could benefit from an index change. By changing the threshold parameters in the control feedback loop, the system can be tuned to favor analysis time or pattern change recognition. The foundation provided here will be used to explore this tradeoff and to develop an improved utility for real-world applications. The proposed technique affords the opportunity to adjust indexes to new query patterns. A limitation of the proposed approach is that if index set changes are not responsive enough to query pattern changes then the control feedback may not affect positive system changes. However, this can be addressed by adjusting control sensitivity or by changing control sensitivity over time as more knowledge is gathered about the query patterns.
From initial experimental results, it seems that the best application for this approach is to apply the more time consuming no-constraint analysis in order to determine an initial index set and then apply a lightweight and low control sensitivity analysis for the online query pattern change detection in order to avoid or make the user aware of situations where the index set is not at all effective for the new incoming queries.

In this thesis, the frequency of index set change has been affected through the use of the online indexing control feedback thresholds. Alternatively, the frequency of performance monitoring could be adjusted to achieve similar results and to appropriately tune the sensitivity of the system. This monitoring frequency could be in terms of either a number of incoming queries or elapsed time.

The proposed online change detection system utilized two control feedback loops in order to differentiate between inexpensive and more time consuming system changes. In practice, the fine grain control threshold was not triggered unless we contrived a situation such that it would be triggered. The kinds of low cost changes that this threshold would trigger are not the kinds of changes that make enough impact to be that much better than the existing index set. This would change if the indexing constraints were very low, and one potential index is now more valuable than a suggested index.

Index creation is quite time-consuming. It is not feasible to perform real-time analysis of incoming queries and generate new indexes when the patterns change. Potential indexes could be generated prior to receiving new queries, and when indicated by online analysis, moved to active status. This could mean moving an index from local storage to main memory, or from remote storage to local storage depending
on the size of the index. Additionally, the analysis could prompt a server to create a potential index as the analysis becomes aware that such an index is useful, and once it is created, it could be called on by the local machine.
CHAPTER 4

Multi-Attribute Bitmap Indexes

The significant problems we have cannot be solved at the same level of thinking with which we created them. - Albert Einstein (attributed).

An access structure built over multiple attributes affords the opportunity to eliminate more of a potential result set within a single traversal of the structure than a single attribute structure. However, this added pruning power comes with a cost. Query processing becomes more complex as the number of potential ways in which a query can be executed increases. The multi-attribute access structure may not be effective for general queries as a multi-attribute index can only prune results for those attributes that match selection criteria of the query. This chapter presents multi-attribute bitmap indexes. It describes the changes required when modifying a traditionally single attribute access structure to handle multiple attributes in terms of enhancing the query processing model and adding cost-based index selection.

4.1 Introduction

Bitmap indexes are widely used in data warehouses and scientific applications to efficiently query large-scale data repositories. They are successfully implemented in commercial Database Management Systems, e.g., Oracle [2], IBM [46], Informix [33],
Sybase [20], and have found many other applications such as visualization of scientific data [56]. Bitmap indexes can provide very efficient performance for point and range queries thanks to fast bit-wise operations supported by hardware.

In their simplest and most popular form, equality encoded bitmaps also known as projection indexes, are constructed by generating a set of bit vectors where a '1' in the \(i\)th position of a bit vector indicates that the \(i\)th tuple maps to the attribute-value associated with that particular bit vector. In general, the mapping can be used to encode a specific value, category, or range of values for a given attribute. Each attribute is encoded independently.

Queries are then executed by performing the appropriate bit operations over the bit vectors needed to answer the query. A range query over a value-encoded bitmap can be answered by ORing together the bit vectors that correspond to the range for each attribute, and then ANDing together these intermediate results between attributes. The resultant bit vector has set bits for the tuples that fall within the range queried.

One disadvantage of bitmap indexes is the amount of space they require. Compression is used to reduce the size of the overall bitmap index. General compression techniques are not desirable as they require bitmaps to be decompressed before executing the query. Run length encoders, on the contrary, allow the query execution over the compressed bitmaps. The two most popular are Byte-Aligned Bitmap Code (BBC) [2] and Word Aligned Hybrid (WAH) [57]. The CPU time required to answer queries is related the total length of the compressed bitmaps used in the query computation.
These traditional bitmap indexes are analogous to single dimension indexes in traditional indexing domains. However, just as traditional multi-attribute indexes can be more effective in pruning data search space and reducing query time, multi-attribute bitmap indexes could be applied to more efficiently answer queries. This cost savings comes from two sources. First, the number of bitwise operations is potentially reduced as more than one attribute are evaluated simultaneously. Second, the bitvectors built over two or more attributes are sparser, which allows for greater compression. The resulting smaller bitmaps makes the processing of the bitmap faster which translates into faster query execution time.

As an example, consider a 2-dimensional query over a large dataset. In order to avoid sequential scan over the data we build a index access structure that can direct the search by pruning the space. If single dimensional indexes, such as B+-Trees or projection indexes, are available over the two attributes queried, the query would find candidates for each attribute and then combine the candidate sets to obtain the final answer. In the case of B+-Trees, the set intersection operation is used to combine the sets. In the case of projection indexes, the AND operation is used to combine the bitmaps. However, if a 2-dimensional index such as an R-tree or Grid File exists over the two queried attributes, then the data space can be pruned simultaneously over both dimensions. This results in a smaller result set if the two dimensions indexed are not highly correlated. Also a set intersection operation is not required. Similarly, a multi-dimensional bitmap over a combination of attributes and values identifies those records that match the query combination and avoids a further bit operation in order to answer the query criteria.
In this thesis, the use of Multi-Attribute Bitmaps is proposed to improve the performance of queries in comparison with traditional single attribute indexes. The goals of this research are to:

- Evaluate of query performance of multi-attribute bitmap enhanced data management system compared to the current bitmap index query resolution model.

- Support ad hoc queries where information about expected queries is not available and support the experimental analysis provided herein. We propose an expected cost-improvement based technique to evaluate potential attribute set combinations.

- Given a selection query over a set of attributes and a set of single and multiple attribute bitmaps available for those attributes, determine the bitmap query execution steps that results in the best query performance with minimal overhead.

The proposed approaches to attribute combination and query processing are based on measuring the expected query cost savings of options. Although the techniques incorporate heuristics in order to control analysis search space and query execution overhead, experiments show significant opportunity for query execution performance improvement.

4.2 Background

Bitmap Compression. Bitmap tables need to be compressed to be effective on large databases. The two most popular compression techniques for bitmaps are the Byte-aligned Bitmap Code (BBC) [2] and the Word-Aligned Hybrid (WAH) code
Unlike traditional run length encoding, these schemes mix run length encoding and direct storage. BBC stores the compressed data in Bytes while WAH stores it in Words. WAH is simpler because it only has two types of words: literal words and fill words. The most significant bit indicates the type of word we are dealing with. Let \( w \) denote the number of bits in a word, the lower \( (w-1) \) bits of a literal word contain the bit values from the bitmap. If the word is a fill, then the second most significant bit is the fill bit, and the remaining \( (w-2) \) bits store the fill length. WAH imposes the word-alignment requirement on the fills. This requirement is key to ensure that logical operations only access words.

Figure 4.1 shows a WAH bit vector representing 128 bits. In this example, we assume 32 bit words. Under this assumption, each literal word stores 31 bits from the bitmap, and each fill word represents a multiple of 31 bits. The second line in Figure 4.1 shows how the bitmap is divided into 31-bit groups, and the third line shows the hexadecimal representation of the groups.

The last line shows the values of WAH words. Since the first and third groups do not contain greater than a multiple of 31 of a single bit value, they are represented as literal words (a 0 followed by the actual bit representation of the group). The fourth group is less than 31 bits and thus is stored as a literal. The second group contains a multiple of 31 0’s and therefore is represented as a fill word (a 1, followed by the 0 fill
The first three words are regular words, two literal words and one fill word. The fill word 80000002 indicates a 0-fill of two-word long (containing 62 consecutive zero bits). The fourth word is the active word, it stores the last few bits that could not be stored in a regular word. Another word with the value nine, not shown, is needed to stores the number of useful bits in the active word. Logical operations are performed over the compressed bitmaps resulting in another compressed bitmap.

Bit operations over the compressed WAH bitmap file have been shown to be faster than BBC (2-20 times) [57] while BBC gives better compression ratio. In this paper we use WAH to compress the bitmaps due to the impact on query execution times.

**Relationship Between Bitmap Index Size and Bitmap Query Execution Time.** Using WAH compression, queries are executed by performing logical operations over the compressed bitmaps which result in another compressed bitmap. As an example, consider the two WAH compressed bitvectors in Figure 4.2.

First, the first word of each bitvector is decoded, and it is determined that they are both literal words. Therefore, the logical AND operation is performed over the two words and the resultant word is stored as the first word in the result bitmap. When the next word is read from both bitmaps, we have two fill words. The fill word from B1 is a zero-fill word compressing 2 words, and the fill word from B2 is a one-fill
word compressing 3 words. In this case we operate over 2 words simultaneously by
ANDing the fills and appending a fill word into the result bitmap that compresses
two words (the minimum between the word counts). Since we still have one word
left from B2, we only read a word from B1 this time, and AND it with the all 1
word from B2. Finally, the last two words are read from the two bitmaps and are
ANDed together as they are literal words. A full description of the query execution
process using WAH can be found in [59]. Query execution time is proportional to
the size of the bitmap involved in answering the query [56]. To illustrate the effect
of bit density and consequently bitmap size on query times over compressed bitmaps
we perform the following experiment. We generated 6 uniformly random bitmaps
representing 2 million entries for a number of different bit densities and ANDed the
bitmaps together. Figure 4.3 shows average query time against the total bitmap
length of the bitmaps used as operands in the series of operations.

The expected size of a random bitmap with \( N \) bits compressed using WAH is
given by the formula [59]:

\[
m_R(d) \approx \frac{N}{w - 1} \left( 1 - (1 - d_B) \right)^{2^{w-2}} - d^{2^{w-2}}
\]

where \( w \) is the word size, and \( d \) is the bit density.

The lower the bit density, the more quickly and more effectively intermediate
results can be compressed as the number of AND operations increases. Figure 3 shows
a clear relationship between query times and bitmap length. The different slopes of
the curve are due to changes in frequency of the type of bit operations performed
over the word size chunks of the bitmap operands. At low bit densities, there is a
greater portion of fill word to fill word operations. As the bit density increases, more
fill word to literal word operations occur. At high bit densities, literal word to literal
word operations become more frequent. As these inter-word operations have different costs, the overall query time to bitmap length relationship is not linear. However it is clear that query execution time is related to overall bitmap length.

**Relationship Between Number of Bitmap Operations and Bitmap Query Execution Time.** Figure 4.4 shows AND query times over WAH-compressed bitmaps as the number of bitmaps ANDed varies. Again the bitmaps are randomly generated with the reported bit densities for 2 million entries and the number of bitmaps involved in a AND query is varied. There is clear relationship between query times and the number of bitmap operations.
Since query performance is related to the total bitmap length and number of bitmap operations, a logical question is "can we modify the representation of the data in such a way that we can decrease these factors?" One could combine attributes and generate bitmaps over multiple attributes in order to accomplish this. This is because a combination over multiple attributes has a lower bit density and is potentially more compressible, and already has an operation that may be necessary for query execution built into its construction. However, several research questions arise in this scenario such as which attributes should be combined, how can the additional indexes be utilized, and what is the additional burden added by the richer index sets.
Technical Motivation for Multi-Attribute Bitmaps. Figure 4.5 shows a simple example combining two single attributes with cardinality 2, into one 2-attribute bitmap with cardinality 4. In the table, the combined column labeled b1.b2 means that the value of attribute 1 maps to b1 and the value for attribute 2 maps to b2. Such a representation can have the effect of reducing the size of the bitmaps that are used to answer a query and can also reduce the number of operations. As attributes are combined, bitcolumns become sparser, and the opportunity for compressing that column increases, potentially decreasing the size of a bitcolumn. Additionally, if a bit operation is already incorporated in the multi-attribute bitmap construction, then it does not need to be evaluated as part of the query execution.

One could think of multi-attribute bitmaps as the AND bitmap of two or more bitmaps from different attributes. One could argue that multi-attribute bitmaps are reducing the dimensionality of the dataset but increasing the cardinality of the attributes involved in the new dataset. Although bitmap indexes have historically been discouraged for high-cardinality attributes, the advances in compression techniques and query processing over compressed bitmaps have made bitmaps an effective solution for attributes with cardinalities on the order of several hundreds [58].

As an illustrative example of enhancing compression from combining attributes, consider a bitmap index built over two attributes, each with cardinality 10, where the data is uniformly randomly distributed independently for each attribute for 5000 tuples. Using WAH compression, there are very few instances where there are long enough runs of 0’s to achieve any compression (since out of 10 random tuples, we would expect 1 set bit for a value). In such a test which matched this scenario, very
little compression was achieved and bitmaps representing each of the 20 attribute-value pairs were between 640 and 648 bytes long.

As an alternative, we can generate a bitmap index for the combination of the two attributes. Now the cardinality of the combined attributes is 100, and the expected number of set bits in a run of 100 tuples for a single value is 1. We can expect much greater compression for this situation. In the given scenario, the generated size of the 100 2-D bitmaps was between 220 and 400 bytes.

Now consider a range query such that attribute 1 must be between values 1 and 2, and attribute 2 must be equal to 3. The operations using the traditional bitmap processing to answer the query is ((ATT1=1) OR (ATT1=2)) AND (ATT2=3). The time required is dependent on the total length of the bitmaps involved. Each of these bitmaps as well as the resultant intermediate bitmap from the OR operation is 648 bytes, for a total of 2592 bytes. For the multi-attribute case, the query can be answered by (ATT1=1,ATT2=3) OR (ATT1=2,ATT2=3). These bitmaps are 268 and 260 bytes respectively, for a total of 528 bytes. In this scenario, we can reduce
both the size of the bitmaps involved in the query execution and also the number of bitmaps involved.

This example demonstrates the opportunity to improve query execution by representing multiple attribute values from separate attributes in a bitmap. In general, point queries would always benefit from the multi-attribute bitmaps as long as all the combined attributes are queried. However, if only some of the attributes were queried or a large range from one or more attributes were queried, then the single attribute bitmaps would perform better than the multi-attribute bitmaps. This is the reason why we propose to supplement the single attribute bitmaps with the multi-attribute bitmaps and not to replace the single attribute bitmaps. Multi-attribute bitmaps would only be used in the queries that result in an improved query execution time.

Supplementing single attribute bitmaps with additional bitmaps that improve query execution time has also been done in [50], where lower resolution bitmaps were used in addition to the single attribute bitmaps or high resolution bitmaps. One could think of this approach as storing the OR of several bitmaps from the same attribute together.

4.3 Proposed Approach

In this section we present our proposed solution for the problems presented in the previous section. The primary overall tasks can be summarized as multi-attribute bitmap index selection and query execution. In index selection, we determine which attributes are appropriate or the most appropriate to combine together. This can be performed with varying degrees of knowledge about a potential query set. Once candidate attribute combinations are determined, the combined bitmaps are generated.
In query execution, the candidate pool of bitmaps are explored to determine a plan for query execution.

4.3.1 Multi-attribute Bitmap Index Selection for Ad Hoc Queries

In many cases the attributes that should be combined together as multi-attribute bitmaps are obvious given the context of the data and queries. Whenever multiple attributes are frequently queried together over a single bitmap value, that combination of attributes is a candidate for multi-attribute indexing. As an example, consider a database system behind a car search web form. If the form has categorical entries for color and model, then a multi-attribute bitmap built over these two attributes can directly filter those data objects that meet both search criteria without further processing.

In order to support situations where no contextual information is available about queries and in order to allow for comparative performance evaluation, we provide a cost improvement based candidate evaluation system. The goal behind bitmap index selection is to find sets of attributes that, if combined, can yield a large benefit in terms of query execution time. The analogy within the traditional multi-attribute index selection domain would be finding attributes that are not individually effective in pruning data space, but can effectively prune the data space when combined together. An effective index selection technique for bitmap indexes should be cognizant of the relationships between attributes. As well, the selectivity estimated for a given attribute combination should accurately reflect the true data characteristics.

Since query execution time is dependent both on the total size of the operands involved in a bitmap computation and also the number of bitmap operations, we use
these measures to determine the potential benefit for a given attribute combination. Algorithm 3 shows how candidate multi-attribute indexes are evaluated and selected in terms of potential benefit.

**Determining Index Selection Candidates**

Algorithm 3: Selecting Attribute Combination Candidates.

```python
SelectCombinationCandidates (Ordered set of attributes A, dLimit, cardLimit)

notes:
attSets - is a list of Attribute Sets and corresponding goodness measures
selected - are a set of recommended attribute sets to build indexes over

1: attSets = [{a_i}, 0] | a_i ∈ A
2: currentDim = 1
3: while (currentDim < dLimit AND newSetsAdded)
4:   unset newSetsAdded
5:   for each as_i ∈ attSets, s.t. |as_i| = currentDim
6:     for each attribute a_j, j = (maxAtt(as_i)+1) to |A|
7:       if (as_i.card * a_j.card < cardLimit)
8:         as_j = [as_i ∪ {a_j}, detGoodness(as_i, a_j)]
9:       if goodness(as_j) > goodness(as_i)
10:          add as_j to attSets; set newSetsAdded
11:     currentDim += 1
12:   sort attSets by goodness
13: ca = ∅
14: while ca ≠ A
15:   attribute set as = next set from attSets
16:   if as ∩ ca = ∅
17:     ca = ca ∪ as.attributes
18: include as in selected
```

Algorithm 3: Selecting Attribute Combination Candidates.
The algorithm is somewhat analogous to the Apriori method for frequent itemset mining in that a given candidate set is only evaluated if it’s subset is considered beneficial. The algorithm takes several parameters as input. These include an array, $A$, of the attributes under analysis. The parameter $dLimit$ provides a maximum attribute set combination size. The maximum cardinality allowed for a given prospective combination can be controlled by the $cardLimit$ parameter.

The algorithm evaluates increasingly larger attribute sets (loop starting at line 3). Each unique combination is evaluated (lines 5-6). Initially all size 1 attribute sets are combined with each other size 1 attribute set such that each unique size 2 attribute set is evaluated. In the next iteration of the outer loop, each potentially beneficial size 2 set is evaluated with relevant size 1 attributes (note that by evaluating only those singleton sets of attributes numbered larger than the largest in the current set, we can evaluate all unique set instances). If the number of bitmaps generated by this combination is excessive (line 7), the set will not be evaluated. The $card$ value associated with an attribute set is the number of value combinations possible for the attribute set, and reflects the number of bitmaps that would have to be generated to index based on the set. A goodness measure is evaluated for the proposed combination (line 8). If the goodness of the combination exceeds the goodness of its ancestor, then the combination set is added to the list of attribute sets for further evaluation during the next loop iteration. The term $newSetsAdded$ in line 3 is a boolean condition indicating if any attribute set was added during the last loop iteration.

Once the loop terminates, the sets in $attSets$ are sorted by their goodness measures (line 12). Then sets of attributes that are disjoint from the covered attributes,
ca, are greedily selected from the sorted sets and added to the recommended bitmap indexes.

**Estimating Performance Gain for a Potential Index**

\[
\text{detGoodness} \text{ (attribute set } a_s, \text{ attribute } a_j) \\
1: \text{ set } \text{probQT} \text{ and } \text{probSR} \\
2: \text{ goodness}=0 \\
3: \text{ for each bitcolumn } b_c_k \text{ in } a_s \\
4: \quad \text{ for each bitcolumn } b_c_j \text{ in } a_j \\
5: \quad \text{ bitcolumn } c_b = \text{AND}(b_c_k, b_c_j) \\
6: \quad \text{savings} = b_c_k.\text{length} + b_c_j.\text{length} + \\
\quad \quad \quad b_c_k.\text{creationLength} - c_b.\text{length} \\
7: \quad \text{ if savings} > 0 \\
8: \quad \quad \text{goodness+} = \text{savings} \times c_b.\text{matches} \\
9: \quad \text{goodness*} = (\text{probQT} \times \text{probSR})^{|a_s|+1} \\
10: \text{ return goodness}
\]

**Algorithm 4:** Determine goodness of including an attribute in a multi-attribute bitmap.

The goodness of a particular attribute set is computed by Algorithm 4. The variables, \text{probQT} and \text{probSR} describe global query characteristics and are used to estimate diminishing benefits as index dimensionality increases. The \text{probQT} parameter represents the probability that an attribute is queried. As with traditional multi-attribute index, a 2-attribute index is not as useful for a single attribute query. The \text{probSR} parameter is the probability that the query range for an attribute will be small enough such that a multi-attribute index over the query criteria will still be beneficial.
Each unique value combination of the entire attribute set (combinations generated in lines 4-5) is evaluated in terms of bitmap length savings over a corresponding single attribute bitmap solution. The goodness of an attribute set is measured as the weighted average of the bitlength savings of each value combination (line 8). The savings of a particular combination is measured as the sum of the compressed lengths of the bitmap operands and the compressed length of the bitmaps needed to create the first operand, minus the compressed length of the combined bitmap. For size 1 attribute sets, the bitcolumn creation length is 0. However, for a multiple attribute bitmap this value is the length of all the constituent bitmaps used to create it. The total savings reflects the difference in total bitmap operand length to answer the combination under analysis.

If the savings for a particular combination is negative, then its negative savings is not accumulated (line 7). This is because this option will not be selected during the query execution for this combination and will not incur an additional penalty. Each combination is weighted by its frequency of occurrence. This strategy assumes that the query space is similar in distribution to the data space. The final goodness measure for an attribute set is also modified based on the probability that benefit will actually be realized for the query (line 9).

This technique finds attributes for which their combinations will lead to a reduction in the number of operations as well as a reduction in the constituent bitmap lengths. Both of these factors can provide a performance gain depending on the query criteria. Our approach takes into account the characteristics of the data by weighting and measuring the effectiveness of specific inter-attribute combinations. The index selection solution space is controlled by limiting parameters and through
greedy selection of potential solutions. Although the indexes recommended are not
 guaranteed to be optimal, they can be determined efficiently. Later experiments show
 that the reduction in the number of total operations has a more substantial effect on
 query execution time than reduction of bitmap length, (i.e. seemingly less than ideal
 attribute combinations can still provide performance gains).

 The behavior of the algorithm can be affected with slight alteration. For example,
 if index space is critical, we can order the pairs in line 12 of the Select Combination
 Candidates algorithm (Algorithm 3) by goodness over combined attribute size. If the
 query is equally likely to occur in the data space, then the weighting in line 8 of the
determine Goodness algorithm (Algorithm 4) can be removed.

 Once bitmap combinations are recommended, each combined bitmap can be gen-
erated by assigning an appropriate identifier to the bitmap and ANDing together the
 applicable single attribute bitmaps. Once multi-attribute bitmaps are available in the
 system, we need a mechanism to decide when single attribute and/or multi-attribute
 bitmaps would be used. In other words, we need to decide on a query execution plan
 for each query.

 4.3.2 Query Execution with Multi-Attribute Bitmaps

 One issue associated with using multiple representations for data indexes, is that
 the query execution becomes more complex. In order to take advantage of the most
 useful representation, the potential execution plans must be evaluated and the op-
tion that is deemed better should be performed. To be effective, the complexity
 added by the additional representations can not be more costly than the benefits
 attained by incorporating them. We take a number of measures in order to limit the
overhead associated with selecting the best query execution option. We keep track of those attribute sets that have bitmap representations (either single attribute or multi-attribute). We use a unique mapping for bitmap identification. We explore potential query execution options using inexpensive operations.

Global Index Identification

In order to identify the potential index search space, we maintain a data structure of those attributes and combined attributes for which we have bitmap indexes constructed. If we have available a single attribute bitmap for the first attribute, we will include a set identified as [1] in this structure. If we have available a multi-attribute bitmap built over attributes 1, 2, and 3 then we include the set [1,2,3] in the structure. Upon receiving a query, we compare the set representation of the attributes in the query to the sets in the index identification structure. Those sets that have non-empty intersections with the query set are candidates for use in query execution.

Bitmap Naming

We need to be able to easily translate a query to the set of bitmaps that match the criteria of a query. In order to do this we devise a one-to-one mapping that directly translates query criteria to bitmap coverage. The key is made up of the attributes covered by the bitmap and the values for those attributes. For example, a multi-attribute bitmap over attributes 9 and 10 where the value for attribute 9 is 5 and the value for attribute 10 is 3 would get "[9,10]5.3" as its key. The attributes are always stored in numerical order so that there is only one possible mapping for various set orders, and the values are given in the same order as the attributes. Finding relevant bitmaps is a matter of translating the query into its key to access the bitmaps.
Query Execution

During query execution we need a lightweight method to check which potential query execution will provide the fastest answer to the query. The first step is to compare the set of attributes in the query to the sets of attributes for which we have constructed bitmap indexes. Those indexes which cover an attribute involved in the query are candidates to be used as part of the solution for the query. Once candidate indexes are determined, we compute the total length of the bitmaps required to answer the query.

A recursive function is used to compute the total length of all matching bitmaps in order to handle arbitrary sized attribute sets. The complete unique hash key is generated in the base case of the recursive method and the length of the matching bitmaps is obtained and returned to the calling caller. For the recursive case, the unique key is built up and passed to the recursive call, and the callee length results are tabulated.

We just need to sum the lengths of bitmaps that match the query criteria, which we can compute using the recursive function. For example, if the total length of matching bitmaps for the multi-attribute bitmap over attributes [1,2] is less than the total length for both attributes [1] and [2], then the multi-attribute bitmap should be used to answer the query. However, if the multi-attribute bitmaps have greater total length, then the single attribute bitmaps should be used.

The logic applied to determine the query execution plan needs to be lightweight so as to not add too much overhead to handle the increased complexity of bitmap representation. So rather than incorporating logic that guarantees an optimal minimal total bitmap length, we approximate this behavior using heuristics. We order the
potential attribute sets in descending order based on the amount of length saved by using the index. As a baseline, all potential single attribute indexes get a value of 0 for the amount of length saved. Multi-attribute indexes get a value which is the difference between the length of the single attribute indexes that would be required to answer the query and the length of the multi-attribute index to answer the query. Then we process the query by greedily selecting non-covered attribute sets from this order until the query is answered. Those multi-attribute indexes that have positive benefit compared to single attribute indexes will have positive value, and will be processed before any single attribute index. Those that have negative values, would actually slow down the query, and will not be processed before their constituent single attribute indexes.

Algorithm 5 provides the algorithm for executing selection queries when the entire bitmap index has both single attribute and multi-attribute bitmap indexes available to process the query. It involves two major steps. In the first step, the potential indexes are prioritized based on how promising they are with respect to saving total bitmap length during query execution. The second step performs the bit operations required to answer the query. The algorithm assumes that the bitmap indexes available are identified and that we can access any bitcolumn through a unique hashkey.

In lines 1-5, we compute the total size of the bitmaps that match the query considering those attributes that intersect between the query and the index attribute set currently under analysis. For example, consider a query over values 1 and 2 for attribute 1, and and over value 3 for attribute 2 where we have bitmaps available for the attribute sets [1], [2], and [1,2] combined. Table 4.1 shows the query matching keys for each of the potential indexes. At the end of this section, we will have a total
**SelectionQueryExecution** (Available Indexes $AI$, BitColumn HasTable $BI$, query $q$)

**notes:**
$AI$ - contains identification of sets of indexed attributes, with *length* and *saved* fields and query matching *keys* list
$BI$ - provides access to bitcolumn using unique key
$B1$ - a bitcolumn of all 1’s

// Prioritize Query Plan
1:   for each attribute set as in $AI$
2:      if $as.attributes \cap q.attributes \neq \emptyset$
3:         generate query matching keys for as
4:         attach query matching keys to as
5:         sum query matching bitmap lengths
6:   for each attribute set as in $AI$
7:      if $as.numAtts == 1$
8:        $as.saved = 0$
9:      else
10:         for each size 1 attribute subset $a$ in $as$
11:            $as.saved += a.length$
12:            $as.saved -= as.length$
13:         sort $AI$ by saved field

// Execute Query
14:   attribute set $ca = null$
15:   queryResult = $B1$
16:   while $ca \neq q.attributes$
17:      $as = next attribute set from $AI$
18:      if $as.attributes \cap ca \neq \emptyset$
19:         $ca = ca \cup as.attributes$
20:         subresult = $BI.get(as.keys[0])$
21:         for $i = 2$ to $|as.keys|$
22:            subresult = OR(subresult,$BI.get(as.keys[i])$)
23:   return queryResult

**Algorithm 5:** Query Execution for Selection Queries when Combined Attribute Bitmaps are Available.
query matching bitmap length for each available index. Note that the computation of bitmap length in line 5 is not actually a simple sum. The bit density of each matching bitmap is estimated based on its length. Since the matching bitmaps will not overlap, these bit densities are summed to arrive at an effective bit density of the resultant OR bitmap. Length of the resultant bitmap is then derived using effective bit density. If the bit density is within the range of about 0.1 to 0.9, then its value can not be determined. In these cases, we conservatively estimate its value as 0.5. This does not adversely affect query operation since the multi-attribute option would not be selected anyway.

<table>
<thead>
<tr>
<th>Available Index</th>
<th>Matching Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>[1,1],[1,2]</td>
</tr>
<tr>
<td>[2]</td>
<td>[2,3]</td>
</tr>
<tr>
<td>[1,2]</td>
<td>[1,2,1,3],[1,2,2,3]</td>
</tr>
</tbody>
</table>

Table 4.1: Sample Query Matching Keys for Query Attributes 1 and 2.

In lines 6-12, we use this initial information in order to compute the total bitmap length saved by using a multi-attribute bitmap in relation to using the relevant single attribute alternatives. For a multiple attribute index, we sum up the query matching bitmap length associated with each single attribute subset of the multi-attribute attribute set. The difference between this length and the total length of the query matching bitmaps for the multi-attribute set is the amount of bitmap length saved by using the multi-attribute set. All single attribute indexes are assigned a saved value of 0. Those multi-attribute indexes that reflect a cost savings will get a positive value.
for saved, while those that are more costly will get a negative value. In line 13, we sort the potential indexes in terms of this saved value.

In lines 14 and 15, we initialize a set that indicates the attributes covered by the query so far and initialize the query result to all 1's. Then we start processing the query in order of how promising the prospective indexes were and continue until all query attributes are covered (line 15). We get the next best index, and check if it's attributes have already been covered (line 17,18). If not, we will process a subquery using the index. We update the covered attribute set and OR together all bitcolumns that match the queries relevant to the query and the index. We AND together this result to build the subqueries into the overall query result. When we have covered all query attributes, the query is complete and we return the result.

4.3.3 Index Updates

Bitmap indexes are typically used in write-once read-mostly databases such as data warehouses. Appends of new data are often done in batches and the whole index is rebuilt from scratch once the update is completed. Bitmap updates are expensive as all columns in the bitmap index need to be accessed. In [12], we propose the use of a padword to avoid accessing all bitmaps to insert a new tuple but only to update the bitmaps corresponding to the inserted values. In other words, the number of bitmaps that need to be updated when a new tuple is appended in a table with $A$ attributes is $A$ as opposed to $\sum_{i=1}^{A} c_i$, where $c_i$ is the number of bitmap values (or cardinality) for attribute $A_i$. This approach, which clearly outperforms the alternative, is suitable when the update rate is not very high and we need to maintain the index up-to-date in an online manner.
Updating multi-attribute bitmaps is even more efficient than updating traditional bitmaps. If multi-attributes bitmaps are also padded we only need to access the bitmaps corresponding to the combined values (attributes) that are being inserted. For example, consider the previous table with A attributes where multi-attribute bitmaps are built over pair of attributes, the cost of updating the multi-attribute bitmaps is half of the cost of updating the traditional bitmaps, as only $\frac{A^2}{2}$ bitmaps need to be accessed.

4.4 Experiments

4.4.1 Experimental Setup

Experiments were executing using a 1.87GHz HP Pavilion PC with 2GB RAM and Windows Vista operating system. The bitmap index selection, bitmap creation and query execution algorithms were implemented in Java.

Data

Several data sets are used during experiments. They are characterized as follows:

- *synthetic* - 12 attributes, 1M records, attributes 1-3 have cardinality 2, 4-6 have cardinality 5, 7-9 have cardinality 10, and 10-12 have cardinality 25. Attributes 1,4,7,10 follow uniform random distribution, attributes 2,5,8, and 11 follow Zipf distribution with $s$ parameter set to 1, attributes 3,6,9, and 12 follow Zipf distribution with $s$ parameter set to 2.

- *uniform* - uniformly distributed random data set. It has 12 attributes, each with cardinality 10, 1M records.
• *census* - a discretized version of the US Census 1990 raw data set. The raw dataset is from the U.S. Department of Commerce Census Bureau website. The discretized dataset comes from the UCI Machine Learning Repository. We used the first 1 million instances and 44 attributes. The cardinality of these attributes varies from 2 to 16. The data set can be obtained from [47]

• *HEP* - a bin representation of High-Energy Physics data. It has 12 attributes. The cardinality per dimensional varies between 2 and 12. There are a total of 122 bitcolumns for this data set. This data set is not uniformly distributed. This data set is made up of more than 2 million records.

• *Landsat* - a bin representation of satellite imagery data. The data set is 60 attributes, each cardinality 10. It contains over 275000 records.

**Queries**

Query sets were generated by selecting random points from the data and constructing point queries from those points. These points remain in the data set, so that it is guaranteed that there will be at least one match for each query. For those cases where range queries are used, the range is expanded around a selected point so that the range is guaranteed to contain at least one match.

**Metrics**

Our goal is to compare performance of a system reflecting the current state of the art, where only single attribute bitmaps are available, against our proposed approach in which both single and multi-attribute bitmaps are available. We present comparisons in terms of query execution time, number of bitmaps involved in answering a
query, and total bitmap size accessed to answer a query. Query execution assumes
the bitmap indexes are in main memory. Further performance enhancement would
be possible during query execution if the required bitmaps need to be read from disk.

4.4.2 Experimental Results
Index Selection

We executed the index selection algorithm over the synthetic data set. We set the
\textit{cardLimit} parameter to 100 and set the query probability parameters to 1 (i.e. no
degradation at higher number of attributes). The attribute sets that were selected
to combine together were $[1,4,7]$, $[2,5,8]$, and $[3,6,9]$. This solution makes sense in
the context of the performance metric of weighted benefit. We would expect more
expected benefit for combinations for which the number of value combinations ap-
proaches the cardinality limit, because a sparser bitmap can be more effectively com-
pressed. We would also expect more expected benefit from combining attributes that
are independent. Therefore set $[1,4,7]$ is selected first, because it’s value combination
cardinality is 100, and the values are independent. The other combinations have
greater inter-attribute dependency because they follow Zipf distributions. Attributes
10, 11, and 12 do not combine with any others in this scenario because there are
no free attributes that would be within cardinality constraints. The average query
execution of 100 point queries using only single attribute bitmaps was 41.47ms, while
it was 17.39ms using the recommended indexes. The time for using recommended
index set includes an average overhead of 0.25ms to explore the richer set of options
and select the best query execution plan.

After executing index selection with parameters \textit{probQT} and \textit{probSR} set to 0.5,
we instead get 2 attribute recommendations. The proposed set of indexes is $[7,8]$,
Table 4.2: Query Definitions for 2 Attributes with cardinality 10 and uniform random distribution.

[1,10], [2,11], [4,9], [5,6], and [3,12]. Each of these has a value combination cardinality greater than or equal to 50 and selection order is dependent on cardinality and then inter-attribute independence.

Query Execution

Controlled Query Types. We ran 6 query types over bitmaps built over 2 attributes with uniform random distribution and cardinality 10. The queries are detailed in Table 4.2 and varied in the length of the range. A range of 1, means that the query for that attribute is a point query. Q1, for example, is a point query on both attributes, and Q3 selects one value from the first attribute and five values from the second attribute. Note that in this case, 5 is the worst case scenario. Any range bigger than five could take the complement of the non-queried range to answer the query.

Figures ?? and 4.8 show the query execution time, number of bitmaps involved in answering each query type, and total size of the bitmaps involved in answering the query, respectively. In these figures, The ‘Only 1-Dim’ label refers to the case where only Single Attribute Bitmaps are available, the ‘Only M-Dim’ label refers to the
case where the queries are forced to use Multi-Attribute Bitmaps, and the ‘Proposed’
label refers to the case where the query execution plan is selected considering both
indexes available.

For queries Q1-Q4, the multi-attribute bitmaps are always selected, and they are
significantly faster than the single-attribute options. For queries Q5-Q6, if single and
multi-attribute bitmaps are available, single attribute is always selected. However,
the best is always slightly faster than the proposed technique due to the overhead of
evaluating and selecting the best option. For these queries, the overhead was on the
order of 0.2ms. The proposed technique usually selects the option that involves the
shortest original bitmaps required to answer the query. The exception here is Q5.
The single dimension index is used because it involves fewer bitmap operations and
involves less total bitmap operand length over the set of operations that need to take
place.

![Figure 4.6: Query Execution Time for each query type.](image)

Figure 4.6: Query Execution Time for each query type.
Figure 4.7: Number of bitmaps involved in answering each query type.

Query Execution for Point Queries. Figure 4.9 demonstrates the potential query speedup associated with our proposed query processing technique when multi-attribute bitmaps are available to answer queries. The single-D bar shows the average query execution time required to answer 100 point queries using traditional bitmap index query execution, where points are randomly selected from the data set (there is at least one query match). For the multi-attribute test, indexes were selected and generated with a maximum set size of 2. So a suite of bitcolumns representing 2-attribute combinations is available for query execution as well as the single attribute bitmaps. Each single attribute is represented in exactly one attribute pair. The results show significant speedup in query execution for point queries. Point queries over the uniform, landsat, and hep datasets experience speedups of factors of 5.01, 3.23, and 3.17 respectively.
Similar results were achieved for the census data set. Indexes were selected without a dimensionality limit and with a cardinality limit of 100. The query execution time using only single attribute bitmaps was 83.01 ms while the query execution time was only 24.92 ms when both, single attribute and multi-attribute bitmaps were available.

**Query Execution Over Range Queries.** Figure 4.10 shows the impact of using the single-attribute bitmaps instead of the proposed model as the queries trend from being pure point queries to gradually more range-like queries. Average query execution times over 100 queries are shown. The queries are constructed by making the matching range of an attribute either a single value, or half of the attribute cardinality. Whether an attribute is a point or a range is randomly determined. The x-axis shows the probability that a given pair of attributes in the query represents a range (e.g. .1 means that there is 90% probability that 2 query attributes are both over single values). The graph shows significant speedup for point queries. Although the
speedup ratio decreases as the queries become more range-like, the multi-attribute alternative maintains a performance advantage until nearly all the query attributes are over ranges. This is because there are still some cases where using the multi-attribute bitmap is useful.

**Index Size**

Figure 4.11 shows bitmap index sizes for each data set. The lower portion of the bar shows the total size of the bitmaps for all of the single attribute indexes. The upper portion of the bar shows the size of the suite of 2-attribute bitmaps (each attribute represented exactly once in the suite). The entire bar represents the space required for both the single and 2-attribute bitmaps. Even though the multi-attribute
Figure 4.10: Query Execution Times as Query Ranges Vary, Landsat Data

bitmaps represent many more columns than the single attribute bitmaps (e.g., the single dimension landsat are covered by 600 bitcolumns, while its 2-attribute counterpart is represented by 3000 bitmaps (30 attribute pairs, 100 bitmaps per pair)), the compression associated with the combined bitmaps means that the space overhead for the multi-attribute representations is not as severe as may be expected.
Figure 4.11: Single and Multi-Attribute Bitmap Index Size Comparison

Bitmap Operations versus Bitmap Lengths

Since query execution enhancement is derived from two sources, reduction in bitmap operations and reduction in bitmap length, it is valuable to know the relative impacts of each one. To this end, we tested query execution after performing index selection using different strategies.

Table 4.3 shows the query speedup for point queries using different strategies to determine the attributes that should be combined together. The speedup is measured over 100 random point queries taken from HEP data. For each listed strategy, a suite of size 2 attribute sets is determined differently. Randomly selected attributes simply combines random pairs of attributes. The other 2 strategies use Algorithm 3 to generate goodness factors for each combination based on bitmap length saved and weighted by result density. The goodness using best greedily selects non-intersecting
Attribute Combination Strategy  | Query Speedup
--- | ---
1  Algorithm 1, Selecting Best Combinations  | 3.15
2  Randomly Selected Attributes  | 2.79
3  Algorithm 1 Selecting Worst Beneficial Combinations  | 2.67

Table 4.3: Query Speedup, HEP Data, Using Different Attribute Combination Strategies

pairs starting from the top of the list, while the other technique greedily selects non-overlapping pairs from the bottom. The results show that significant speedup can be achieved simply by combining reasonable attributes together, and that query execution can be further enhanced by careful selection of the indexes.

4.5 Conclusion

In this paper, we propose the use of multi-attribute bitmap indexes to speed up the query execution time of point and range queries over traditional bitmap indexes. We introduce an index selection technique in order to determine which attributes are appropriate to combine as multi-attribute bitmaps. The technique is driven by a performance metric that is tied to actual query execution time. It takes into account data distribution and attribute correlations. It can also be tuned to avoid undesirable conditions such as an excessive number of bitmaps or excessive bitmap dimensionality.

We introduce a query execution model when both traditional and multi-attribute bitmaps are available to answer a query. The query execution takes advantage of the multi-attribute bitmaps when it is beneficial to do so while introducing very little overhead for cases when such bitmaps are not useful. Utilizing our index selection
and query execution techniques, we were able to consistently achieve up to 5 times query execution speedup for point queries while introducing less than 1% overhead for those queries where multi-attribute indexing was not useful.
CHAPTER 5

Conclusions

Data retrieval for a given application is characterized by the query types, query patterns over time, and the distribution and characteristics of the data itself. In order to maximize query performance, the query processing and access structures should be tailored to match the characteristics of the application.

The optimization query framework presented herein describes an I/O-optimal query processing technique to process any query within the broad class of queries that can be stated as a convex optimization query. For such applications, the query processing follows a access structure traversal path that matches the function being optimized while respecting additional problem constraints.

While resolving queries, the possible query execution plans evaluated by a data management system are limited by available indexes. The online high-dimensional index recommendation system provides a cost-based technique to identify access structures that will be beneficial for query processing over a query workload. It considers both the query patterns and the data distribution to determine the benefit of different alternatives. Since workloads can shift over time, a method to determine when the current index set is no longer effective is also provided.
The multi-attribute bitmap indexes presents both query processing and index selection for the traditionally single attribute bitmap index. Query processing selects a query execution plan to reduce the overall number of operations. The index selection task finds attribute combinations that lead to the greatest expected query execution improvement.

The primary end goal of the research presented in this thesis is to improve query execution time. Both the run-time query processing and design-time access methods are targeted as potential sources of query time improvement. By ensuring that the run-time traversal and processing of available access structures matches the characteristic of a given query and that the access structures available match the overall query workload and data patterns, improved query execution time is achieved.
BIBLIOGRAPHY


