4-2008

Interactive Visual Exploration of Large Relation

Mustafa Sanver
Western Michigan University

Follow this and additional works at: https://scholarworks.wmich.edu/dissertations

Part of the Computer Sciences Commons

Recommended Citation
https://scholarworks.wmich.edu/dissertations/812
INTERACTIVE VISUAL EXPLORATION OF LARGE RELATION DATA

by

Mustafa Sanver

A Dissertation
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Doctor of Philosophy
Department of Computer Science
Dr. Li Yang, Advisor

Western Michigan University
Kalamazoo, Michigan
April 2008
INTERACTIVE VISUAL EXPLORATION OF LARGE RELATION DATA

Mustafa Sauver, Ph.D.

Western Michigan University, 2008

As data become ubiquitous and immense in many activities from business decision making to scientific analysis, effective analysis of such data becomes an important research area. Data analysis requires active involvement of human beings and visualization is a powerful tool to capture profound insights from data. However, interactive visual exploration of massive data poses fundamental technical challenges to both data visualization and database management systems. Millions of billions of data records clutter the screen and existing database management systems are inadequate for overview-and-drill down data access for interactive data exploration.

We have proposed a density-based methodology to address the above challenge. We present multiresolution data aggregation as an intermediate representation of data between visualization tools and databases. Data aggregated at multiple resolutions are stored in internal nodes of a partition-based high dimensional tree index while the individual records are stored in leaf nodes. Such a piggyback ride of aggregated data efficiently supports resolution-based data access patterns of large relational data. Thus, multiresolution data aggregation conveys density-based data input to visualizations. Therefore, many existing visualization techniques which are inefficient in visualizing large data can accept this new data input and participate in analysis of large data.
To demonstrate the feasibility and effectiveness of the new representation, a client-server visualization tool is developed. The server, DBMS, is a kdb-tree variant to organize data in multiple resolutions and the client, v.i.s, is a dynamic, flexible, and extendible visualization framework. We have extended 3D footprint splatting with grand tour, density-based parallel coordinates, and density-based scatterplot matrices to render data aggregations. The framework provides a data-independent and plug-in-based environment for quick development and integration of new visualization techniques, in addition to the linking mechanism to interconnect its components.
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI Microform 3303475
Copyright 2008 by ProQuest LLC.
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 E. Eisenhower Parkway
PO Box 1346
Ann Arbor, MI  48106-1346
ACKNOWLEDGMENTS

First and foremost, I submit my gratitude to my advisor Dr. Li Yang for his continuous support, guidance, and encouragement from the very beginning to the completion of my dissertation. It couldn’t have been possible to pursue and finish the work contained in this dissertation today without his help.

Secondly, I thank the members of my dissertation committee, Dr. Elise de Doncker, Dr. Karlis Kaugars, and Dr. Bernard Han for reviewing my work and giving their valuable feedback.

I am very grateful to my friends, colleagues, and mentors for their motivation, inspiration, and assistance in many ways, especially Dr. Mohsen Guizani, Dr. Aba Al-Hajj, Dr. Ghassen Ben Brahim, Dr. Wassim El-Hajj, Osama Awwad, and Dr. Zikri Altun.

In addition, my very special praises go to Mr. Aydin Bolak in remembrance of him and his generous support on my education through Turkpetrol Foundation.

Lastly, my profound love and thanks are due to my mother and my sisters and their families for their emotional support and prayers. They were very encouraging at every step of my studies. My wife Tugba and my daughter Ebrar deserve the cheeriest applauds and appreciation for their continuous patience, enthusiasm, and bearing with me for long hours of work. I devote this dissertation to my late father, Hasan Sanver.

Mustafa Sanver
# TABLE OF CONTENTS

ACKNOWLEDGMENTS .................................................................................. ii

LIST OF TABLES ......................................................................................... vi

LIST OF FIGURES ....................................................................................... vii

CHAPTER

1 INTRODUCTION ....................................................................................... 1
  1.1 Introduction ......................................................................................... 1
  1.2 Challenges of Large Data Visualization ............................................. 2
  1.3 Density-based Methodology ............................................................. 5
  1.4 Overview-and-drill-down and Multiresolution .................................. 6
  1.5 Summary and Dissertation Organization ....................................... 8

2 RELATED WORK ...................................................................................... 11

3 MULTiresOLUTION DATA AGGREGATION ........................................... 17
  3.1 Introduction ......................................................................................... 17
  3.2 Description ......................................................................................... 17
  3.3 Briefly Access Methods ..................................................................... 20
  3.4 High Dimensional Indexes and kdB-tree ......................................... 22
  3.5 Cascading Split Problem ................................................................... 23

4 DATA AGGREGATION TREE .............................................................. 25

iii
Table of Contents—Continued

CHAPTER

4.1 Introduction ............................................................... 28
4.2 Data Aggregation Tree .................................................... 25
4.3 External Memory and TPIE ................................................. 30
4.4 TPIE’s kdB-tree Implementation ......................................... 30
4.5 DA-tree Implementation with TPIE ....................................... 32
4.6 Splitting in a DA-tree ...................................................... 35
4.7 Discussion ................................................................. 38

5 VISUALIZATION AND SOFTWARE ARCHITECTURE .................. 41

5.1 Introduction ............................................................... 41
5.2 The Server ................................................................. 43
  5.2.1 Input File ............................................................ 43
  5.2.2 Metafont .............................................................. 44
  5.2.3 kdB-tree Data Management System (KDBMS) ..................... 46
  5.2.4 Data Generator ...................................................... 49
5.3 Network Messages and SQL Specification ............................ 52
  5.3.1 Network Messages .................................................. 52
  5.3.2 Output Format ..................................................... 53
  5.3.3 Resolution Levels .................................................. 54
  5.3.4 SQL Specification ................................................ 56
5.4 The Visualization Client ............................................... 58
| CHAPTER |
|------------------|------------------|
| 5.4.1 Introduction | 58 |
| 5.4.2 Footprint Splatting | 59 |
| 5.4.3 Density-based Parallel Coordinates | 65 |
| 5.4.4 Record-based Parallel Coordinates | 74 |
| 5.4.5 Scatterplot Matrices | 75 |
| 5.4.6 User Interaction | 77 |
| 5.4.7 Colormap Management | 80 |
| 5.4.8 Color Variable | 81 |
| 5.4.9 Other GUIs | 84 |
| 5.5 Conclusion | 85 |

<table>
<thead>
<tr>
<th>6 VISUALIZATION FRAMEWORK: mVis</th>
<th>87</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Introduction</td>
<td>87</td>
</tr>
<tr>
<td>6.2 Related Work</td>
<td>88</td>
</tr>
<tr>
<td>6.3 Components of mVis</td>
<td>90</td>
</tr>
<tr>
<td>6.3.1 Library Manager</td>
<td>92</td>
</tr>
<tr>
<td>6.3.2 Data Drivers</td>
<td>92</td>
</tr>
<tr>
<td>6.3.3 Data Manager</td>
<td>96</td>
</tr>
<tr>
<td>6.3.4 Clustering and Cluster Manager</td>
<td>96</td>
</tr>
<tr>
<td>6.3.5 Visualizers and Renderers</td>
<td>97</td>
</tr>
<tr>
<td>6.3.6 Generic Libraries</td>
<td>8</td>
</tr>
<tr>
<td>Chapter</td>
<td>Section</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>6.3.7</td>
<td></td>
</tr>
<tr>
<td>6.3.8</td>
<td></td>
</tr>
<tr>
<td>6.3.9</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.1</td>
</tr>
<tr>
<td></td>
<td>7.2</td>
</tr>
<tr>
<td></td>
<td>7.3</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**LIST OF TABLES**

5.1 Format of a Metafont ........................................... 44
5.2 Commands List of the Console ................................... 50
5.3 List of Network Message Codes ................................. 53
5.4 List of Query Result Formats ................................... 53
5.5 SQL Specification .................................................. 57
5.6 Mappings from Data-Space to Screen-Space .................... 72

7.1 Size of DA-trees ................................................... 109
7.2 Results for Resolution by Level Queries ....................... 111
7.3 Summary Information of PUMS Data Sets ....................... 114
<table>
<thead>
<tr>
<th>Figure Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>New Data Representation</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>An Example kdB-tree: 2dB-tree</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Partitioning a kdB-tree Node (adapted from [39])</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>Format of Non-leaf Disk Page</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>TPIE's kdB-tree Implementation</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>DA-tree</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>Root-split</td>
<td>36</td>
</tr>
<tr>
<td>4.5</td>
<td>DA-tree Non-leaf Splitting Methods</td>
<td>37</td>
</tr>
<tr>
<td>4.6</td>
<td>Resolution Levels of a Region</td>
<td>40</td>
</tr>
<tr>
<td>5.1</td>
<td>Client-server Architecture</td>
<td>41</td>
</tr>
<tr>
<td>5.2</td>
<td>Components of KDBMS</td>
<td>43</td>
</tr>
<tr>
<td>5.3</td>
<td>Input Files of the Server</td>
<td>45</td>
</tr>
<tr>
<td>5.4</td>
<td>Internal Design of KDBMS</td>
<td>48</td>
</tr>
<tr>
<td>5.5</td>
<td>The Two-step Bootstrapping Process for Footprint Splitting</td>
<td>60</td>
</tr>
<tr>
<td>5.6</td>
<td>Sample Footprints</td>
<td>62</td>
</tr>
<tr>
<td>5.7</td>
<td>Splatting Using Footprints With and Without Bootstrapping</td>
<td>63</td>
</tr>
<tr>
<td>5.8</td>
<td>Clustered Boston Housing Data [22]</td>
<td>65</td>
</tr>
</tbody>
</table>
List of Figures—Continued

5.9 Density-based Parallel Coordinates for MDA .................................. 69
5.10 Demonstration of Domain Mappings ........................................ 74
5.11 Record-based Parallel Coordinates ........................................... 74
5.12 Scatterplot Matrices ............................................................. 76
5.13 Colormap GUI ..................................................................... 80
5.14 Color Variable Selection GUI ............................................... 84
5.15 Various GUIs ....................................................................... 84

6.1 mVis Framework Components .................................................. 91
6.2 Example SQL Code: Using kdb-driver .................................... 94
6.3 Linking .................................................................................. 100
6.4 Hello World! GUI in mVis ...................................................... 106

7.1 Timing of Building a DA-tree by Insertion ............................... 108
7.2 Building a DA-tree by Insertion ............................................... 110
7.3 Query by Level ..................................................................... 112
7.4 Query by Count ................................................................... 113
7.5 Screen Snapshot of 25K PUMS Data .................................... 119
7.6 Screen Snapshot of 5%PUMS Data ....................................... 116
7.7 Screen Snapshot of 5%PUMS Data—Continued ....................... 117
CHAPTER 1

INTRODUCTION

1.1 Introduction*

Large relational data are becoming ubiquitous in many disciplines. Some of data repositories are governmental data archives, customer and transactional data, scientific experimental data, data collected from various sensors, surveys, and simulations, features extracted from large textural and multimedia streams. Visual exploration gives analytical insight to its users to understand the data better. However, there are two major issues:

1. Information extraction from large relational data is computationally intensive operation,

2. Effective, scalable, and interactive visual representation and its rendering have limitations.

A framework which solves these problems will find immediate applications in many areas from scientific research to business to homeland security.

Visual exploration of a massive amount of data poses fundamental challenges to both data visualization and database management systems. A major finding of this project is the density-based methodology to interactively explore large relational data sets. It uses multiresolution data aggregation as a common representation of data between relational databases and visualization tools. Data aggregated at multiple resolutions are stored in

*This research project is sponsored by the National Science Foundation (IIS-0414857, 10/01/2004 - 09/30/2008). Professor Li Yang at Western Michigan University is the Principal Investigator.
internal nodes of a partition-based high dimensional tree index. Such a piggyback ride of aggregated data supports the overview-and-drill-down data access pattern for interactive data exploration.

1.2 Challenges of Large Data Visualization

The meaning of large data evolves with the advancement of hardware technology. According to a survey by Winter Corporation, data volumes are doubling annually [12]. By large data, we usually imply data that are multivariate and contain so many records that can not be loaded into main memory at once. Large relational data have the following features: (1) they reside on the secondary memory and are usually managed by database management systems; (2) they are in the relational format, and thus are usually assumed as sets of points in high dimensional Euclidean space; (3) they are often collected from multiple sources and are distributed; (4) they are often augmented with new data records and are growing in size.

Large relational data may be high dimensional i.e. a few hundred. One way to deal with the high dimensionality of data is to project high-dimensional points to points in low dimensional space [20]. Data compression and dimension reduction techniques reduce the dimensions but the number still remains large. The projection can be linear to provide an intuitive view or nonlinear to emphasize certain features. Linear projection is characterized by an $n \times m$ projection matrix that orthogonally transforms $nD$ points into $m$-space where $m < n$. The dimension $m$ of the destination space is usually two or three, so that the geometry can be perceived by human eye. There are an infinite number of possibilities to
project high-dimensional data onto lower dimensions. Finding interesting projections is an issue in multivariate data.

For large data sets, loading the entire data set into main memory and processing in memory are no longer valid. Yet users have the same demand for interactivity and response time as they have for small data sets. To address this issue, visualization techniques need to work closely with database techniques. However, visual data exploration is by no means a straightforward database application. So, visual exploration of large relational data leads to fundamental challenges to databases as well.

Cognitive and perceptual visual techniques combined with interactions greatly facilitate the job of analysts, decision makers, scientists, and engineers [94]. Performance, scalability, levels of interaction, and support for panning, zooming, picking and brushing are some comparison parameters that are necessary for interactive visualization tools. These metrics become crucial when the data is large. Thus, performance and efficiency of interactions diminish which makes rendering burdensome to users.

Many visualization techniques have been developed to display data consisting of more than three dimensions (discussed in Chapter 2 in detail). While these techniques provide attractive tools to explore small to medium sized data, they become inadequate to provide most of the comparison parameters in the case of large data. Large data sets pose fundamental challenges to visual data exploration in the following aspects:

1. Memory limit: The basic assumption to load the entire data set into memory and process in memory is no longer valid, while users have the same demand for interac-
tivity and response time as they have for small data sets.

2. **Display limit**: The limited display space is not big enough to display large data sets. Large number of data records cause occlusion and clutter which makes the interpretation difficult.

To address memory limit, visualization techniques need to work closely with database management systems in order to deal with large data sets. Existing techniques have not considered, or rather, have no need to consider the requirement on databases in the exploration of small data sets. Current exploration tools provide basic capabilities, such as query and search of data records in memory, but provide very little to support these operations in backend databases. In a typical scenario, a user selects an interesting subset of data and decides to view more detail and consequently an SQL data aggregation query is issued to the backend database. The cost of such an operation is often prohibitive for interactive data exploration.

What techniques to use on visualization clients is important under shadow of the above limitations. **Visualization of large data requires a larger display area that has physical limitation.** A pixel of a raster display device is perhaps the most economical means to present a data record. For a small data set, therefore, scatterplot is an excellent metaphor for visualization. **The ability to draw scatterplots is a common feature of many data visualization tools. However, scatterplot loses its effectiveness and occlusion occurs as the number of data records becomes large.**

These limitations not only make the visualization difficult but also supported interac-
tivity operations and their performance on client visualizers fade away.

1.3 Density-based Methodology

Limitations discussed above indicate that large relational data in their raw form are rarely appropriate for direct visual presentation. Instead of displaying individual records, a system should transform large amounts of data into forms that facilitate visual exploration and analytical understanding. We propose a new data representation to deal with some of the mentioned issues and limitations. Density-based methodology is a technique to generate a compact representation of large data. Visualization and analytical tools present a density function of data to agglomerate. This form of compact representation makes it much easier to understand and analyze large data. It uses multiresolution data aggregation (MDA) as a common representation of data between relational databases and visualization tools (Figure 1.1). Data aggregated at multiple resolutions are stored in internal nodes of a partition-based high dimensional tree index. The new representation exhibits some advantages. First, a piggyback ride of aggregated data supports the overview-and-drill-down data access pattern for interactive data exploration. Second, it has built-in support for visual interaction and data scalability. Third, existing visualization techniques can be extended to process this representation with a little effort.

Figure 1.1 New Data Representation
Density-based methodology has additional advantages. It displays density representation instead of individual points. Resolution of density representation is defined as the level of its detail. Depending on the resolution, size of density representation is much smaller than the size of the original relational data. This reduces the computational cost of data visualization on the client. In addition, it reduces the traffic burden on network connections between visualization client and database server, and between database servers managing distributed data sets.

Likewise, density-based representation of data provides support to preserve the privacy of individual data records. In multi-user environments, privacy preservation is a serious issue in analytical visualization. Density representations of data can be organized in a hierarchy of resolutions. Therefore, resolution gives a new dimension of privacy preservation. Permissions can be given to each user to access the data till a specific resolution. In this way, density-based representation of data restricts access to individual data records and, therefore, preserves the privacy of individual data records.

1.4 Overview-and-drill-down and Multiresolution

Overview-and-drill-down is a method for interactive visual exploration. It is a viewing operation to navigate from overview to the more detailed view (drill-down). The drill-down process may continue until individual data records are accessed if the user has permission. Density representations at multiple resolutions facilitate the drill-down operation. Such density representations should support user interactions across different resolution levels. The user must be able to change between these levels in a way that is easy to understand.
and track. Such an operation is prohibitive for interactive data exploration if not supported by a backend database.

Recent advances in data warehousing and OLAP have introduced new operations, such as data cube, slice and dice, and iceberg queries, which may benefit data visualization. However, these operations are not enough to prepare data for interactive data exploration. While these operators are useful to aggregate data on subsets of variables, they do not support data exploration operations such as zooming between different resolutions. Interactive exploration operations such as panning, zooming, picking and brushing require special database queries that are beyond the capability of current databases which are traditionally optimized to access data record by record.

Data aggregation is supported to aggregate data records into groups and to present the groups in a summary form. Aggregation functions (count sum minimum, maximum etc.) are well-defined in SQL. Each group is characterized by a range of values of each aggregation variable and represents a hyper-rectangle in the high dimensional space. Groups can be further aggregated into bigger groups to have multiresolution data aggregation. This process effectively builds a hierarchy of multiresolution data aggregations. One problem is how to organize these aggregated data according to their scopes and resolutions. This is not supported in traditional database management systems.

R-tree [15] and its variations have been widely used to index one dimensional data. However, multivariate data needs high dimensional indexing. Traditional database management systems also lack this requirement.
1.5 Summary and Dissertation Organization

Interactive visual data explorations expose challenges to both data visualization and database systems. On one side, there are millions of billions of data records that clutter the screen. On the other side, database management systems are inadequate for overview-and-drill-down data access for interactive data exploration. Existing visualization techniques have to be extended and new techniques have to be developed to address issues of interactive visual exploration.

In this work, we suggest the use of a density representation instead of individual data records as a data interchange mechanism between database and visualization tools. To support the overview-and-drill-down data access pattern, relational data are aggregated into density representations and are organized in multiple resolutions. To organize the data aggregated at multiple resolutions, we propose to piggyback the aggregated data onto internal nodes of a high-dimensional tree index. Visual interactions such as zooming and panning can therefore be performed by index-only queries that visit the aggregated data in internal nodes of the tree index.

To demonstrate the feasibility of this approach, we have developed a software tool for interactive visual exploration of large relational data. Three existing visualization techniques are implemented. Scatterplot matrices and parallel coordinates are extended and integrated to accept multiresolution data aggregations from the server, while 3D footprint splatting are implemented to accept binned data. On the database side, we used a modified kdB-tree [81] data structure as an external high dimensional index to organize multireso-
Multiresolution data aggregation brings technical challenges for long-term research in data visualization, database, and data mining. Chapter 3 describes the rationale and basis of multiresolution data aggregations, in addition to discussing high-dimensional index structures and point access methods. We focus on point access methods and particularly analyze a kdB-tree. The cascade split problem is described in this chapter.

Chapter 4 explicates design and implementation detail of the data aggregation tree (DA-tree) in which we organize the multidimensional data aggregations. After briefly explaining the mechanics of underlying external memory access, we elucidate our splitting algorithms to address the cascade split problem and investigate strengths and weaknesses of DA-trees. We compare the internal design of the DA-tree with the kdB-tree implementation of THE to address some issues such as reduced fan-out and performance after a piggyback ride of data aggregations into internal nodes of a high-dimensional tree index.

Chapter 5 focuses on our client-server visualization architecture. The server, KDBMS, is discussed in great detail regarding how to build and manage a DA-tree and how to transmit density-based aggregation data to the visualization clients. On the client side, we develop a visualization software (mVis) to extend existing visualization techniques to accept the proposed data representation as their input. mVis constitutes 2D footprint splitting
with grand tour, density-based parallel coordinates, density-based scatterplot matrices, and their partner visualizers for record-based input. The two-step bootstrapping and splatting processes are delineated along with the footprint splatting visualization. Also, specific information about the user interfaces and visualizer interactions are outlined in the remainder of the chapter.

Chapter 6 outlines our design of mVis framework, which is a dynamic, flexible, and extendible visualization framework. The design patterns used, the components and the linking mechanism for interconnecting them are discussed. Also, we demonstrate how it is easy to develop and integrate new visualizations with the provided templates and the toolbox objects.

Chapter 7 disseminates our findings from the aforementioned client-server tool. First, performance information is collected from experiments on real world data sets and synthetic data is presented. Then, visual analysis and representation of data aggregations in multiple resolutions are illustrated.

In Chapter 8, we present overall conclusions and major contributions of our work in addition to future enhancements and research directions of the described density-based representation and architecture.
CHAPTER 2

RELATED WORK

Visualization of relational data has been extensively studied [61]. Many techniques such as scatterplot, scatterplot matrices [8], parallel coordinates [55], stick figures [79], dimensional stacking [66] and virtual world [36] have been developed to display data with more than three dimensions. Data visualization systems have also been developed using these techniques. XmdvTool [100] is a tool that integrates scatterplot matrices, parallel coordinates, star glyphs, and dimensional stacking by linked brushing. Visualization of data clusters is supported in systems like 3D-Eye [52].

In order to deal with large data sets, the above techniques were often augmented with hierarchical multiresolution techniques, usually with some degree of interaction and navigation, to convey aggregated information. Fekete [38] displayed one million data records with a pixel-oriented technique without any aggregation. However, most large data sets contain much more than one million data records. Wong and Bergeron [107] described the construction of a multiresolution display using wavelet approximations. Wegman and Luo [102] developed another approach which suggests overplotting translucent data points lines so that sparse areas fade away while dense areas appear emphasized. Fua and Ward [41] proposed a density-based hierarchical parallel coordinates system which is used in XmdvTool to visualize agglomerative data clusters generated by clustering algorithms such as Birch [116]. Stolte, Tang, and Hanrahan [25] extended Polaris [88] to visualize multidimensional data cubes where data values in each dimension are organized into a hierarchical
tree structure. The multi-scale visualization was further extended in [26] to support multiple zooming paths.

Novotny [75] proposed the use of clusters and visual abstraction to reduce the volume of information by reducing the level of abstraction. In [108], Yang et al. used "Value-Relation Display", which is a pixel-oriented technique, where each dimension is represented by a glyph at a Multi-dimensional Scaling (MDS) gradient location in 3D which preserves the relationship between variables. Values of data in each dimension are mapped to pixel colors and displayed in a corresponding glyph. However, in large data sets, many values of a dimension map to the same pixel within a glyph hence occlusion occurs and speed of interaction fades away. Peng et al. [76] presented "Dimension Reordering" to reduce clutters. The approach first defines a visual clutter measure for a selected visualization technique and, then, determines an optimal order of dimensions which minimizes the visual clutter using an optimal ordering algorithm.

One way to visualize high dimensional data is to linearly project the data into 3D space. There are an unlimited number of possible projections and it is important to find interesting ones. "Projection Pursuit" [40] finds a sequence of projections in maximizing a user-defined measure by following a path with the steepest gradients. Projections produced by projection pursuit can be further rendered with smooth transition from one to another by using grand tour [11], which interpolates intermediate projections along a geodesic path from the source projection to the destination projection. Two dimensional data projection and grand tour were implemented in XGobi [90].

"Volume Rendering" is reconstructing an image from the object space to the image
space, by computing contribution of each element (voxel) in data set to image. Several techniques were developed [29, 106, 103]. In Westover's “Footprint Splatting” [103], the final image is generated by computing, for each voxel in the volume data set, its contribution to the final image. The algorithm works by virtually “throwing” the voxels onto the image plane. In this process every voxel in the object space leaves a footprint in the image space that will represent the object.

Grand Tour and Volume Rendering were combined in a3DMap [110, 111] to visualize large data. The idea is to aggregate the data into high dimensional data cubes, project them into 3D and pursue a grand tour, and finally render using 3D footprint splatting. A major problem with this approach is that it does not support multi-resolution data exploration.

For analytical visual representation of multi-dimensional data, parallel coordinates is one of the most intuitive and broadly used visualization technique. Many descendants of the original technique have been proposed. Johansson et al. developed a method of cluster-based parallel coordinates to reveal cluster structures by using high-precision textures [58]. In another related work [57], 3D clustered multi-relational parallel coordinates used to show the relationship of a selected dimension (focus dimension) against the other dimensions. Theisel made the most of space between adjacent axes to reveal/encode more information and replaced straight lines with perturbed lines [93]. In [45], the “crossing problem” was addressed by drawing smooth curves across the axes to allow tracing of a data point and a quadric or cubic middle axis selection were applied. Fanea et al. [34] introduced “Parallel Glyphs” which combines parallel coordinates and star glyphs to benefit advantages of both in reducing clutters. They also applied color scales as texture maps to
glyphs to improve readability of the images.

Data exploration requires database support. Among relational database queries, “Data Aggregation Query” is particularly useful to prepare aggregated data for visual exploration. Since few data records may not create any noticeable visual effect in the visualization of aggregated data, we are often interested in dense data aggregations in which the number of data records exceeds a certain threshold. Such dense data aggregations can be prepared by issuing an “Iceberg Database Query” [35]. Iceberg query addresses the issue of counting to estimate the number of data records. Methods to compute iceberg queries with complex measures [99] or non-monotone measures such as average [48] have been proposed. Parallel algorithms have also been studied in [73].

Another influential development in data aggregation is the “Data Cube Operation“ [46], which can be logically thought of as the union of all data aggregations. Each is obtained by grouping on a subset of all aggregation variables. For visual data exploration, a data cube operator provides data aggregation in high dimensional space as well as further aggregations in lower spaces. This is useful in exploring interesting subsets of data variables. Efficient algorithms for computing data cubes has been a topic of active research with many interesting approaches proposed. However, the data cube operation is fundamentally different to multiresolution data aggregation required in interactive visual data exploration. The former aggregates on subsets of data variables and the latter aggregates data at different resolutions on all applicable variables. Data cube operation does not answer the question of how to aggregate data at multiple resolutions nor how to organize the aggregated data in hierarchy.
Many visualization tools integrate traditional backend databases. In XmdvTool [32], precalculated information is stored with the data and then fetched recursively using caching and prefetching. Then, the recursive step is removed [74] by moving the visual interactions to a labeling tree structure during precomputation. USD [59] finds solutions to store and retrieve unstructured data. GODIVA [70] framework is a stand-alone portable library that provides I/O optimizations and a data management interface for clients. IDEA [77] is an integrated set of tools to support interactive data analysis and exploration. Stonebraker et al. devised a user interface system for database support of scientific visualizations in Tioga [39]. Components are put together into a recipe and multiple browsers, buffering, and optimizers are provided.

In database, to speed up the search for point and region queries, many access methods are proposed. In [42], Gaede classified them into point access methods (PAMs for multidimensional points) and spatial access methods (SAMs for other objects having spatial extensions). Among PAMs are the following tree types: kdB-tree [81], LDB-tree [51], Bkd-tree [80], CRB-tree [44], BV-tree [39], hB-tree [68], and Buddy tree [86]. R-Tree [47], R*-tree [16], X-tree [19] and non-partition methods such as M-tree [27] are SAMs.

Designing access methods in high dimension is difficult, because there is no total ordering that preserves spatial proximity. However, some solutions proposed to maintain the objects, which are close in higher space, also close in lower space. Some researchers investigated space ordering methods to map from a discrete high-dimensional space into a linear order. A dimension reduction technique, for instance, simply maps or reduces multidimensional data into one-dimensional data to take advantage of the worst-case properties
of B-tree [15] indexing. GIMP [115] and space-filling curves [82] (i.e. Z-ordering [56] and UB-tree [14]) are some examples of reduction techniques. Yang et al. proposed a hierarchical dimension reduction for high dimensional data in [49] to reduce clutters.
CHAPTER 3
MULTIRESOLUTION DATA AGGREGATION

3.1 Introduction

In this chapter, we will describe multiresolution data aggregation in detail. After giving a brief explanation about the data access methods and high dimensional indexing data structures, we will provide more information about the base data structure (k-d tree) that we have used to piggyback the aggregation in our implementation.

3.2 Description

Large relational data in the raw format of individual records are inadequate as data sources for interactive data exploration. There are two concerns: (1) what is the proper data representation between databases and visualization tools in order to interactively visualize large relational data; (2) how to support the typical overview-and-drill-down data access pattern and how to combine it with traditional range queries for interactive data exploration.

In this section, we introduce multiresolution data aggregation to address the first concern and we propose data aggregation tree, which piggybacks multiresolution data aggregation onto internal nodes of a partition-based high dimensional tree index, to facilitate database queries of the second concern.

In order to resolve the conflict between user interaction with large relational data and the inability to scan the data set in real time, we suggest using a density representation of data as an intermediate data interchange mechanism between database and visualization.
tools. The following lines will address some of its advantages. In addition to efficiently supplying data to visualization tools, a density representation of data is more friendly to human visual perception than a large number of individual records and, therefore, better facilitates visual data exploration and analytical understanding.

One obvious advantage is that it has a much smaller size than the original relational data. The number of records in the density data depends more on resolution than on the number of records in the original data. This makes data visualization scalable to the size of the original data. Furthermore, many density data records are thin, representing fewer original data records than a predefined threshold. Depending on which technique is used to visualize the data, those thin density data records could be ignored because they hardly contribute any noticeable visual effect to the final visualization. These features of density representation greatly reduce the computational cost of data visualization. In a client/server setting, these features also help to reduce data transfer time between the visualization client and the database server. Another advantage of the density representation of data is that it provides support to preserve the privacy of individual data records. Data resolution is a new dimension for privacy preservation: permissions can be granted to each user according to data resolutions. Thus, a density representation of data enables the granting of permission to each user to access the data till a specific resolution and, therefore, preserves the privacy of individual data records.

A typical practice of visual data exploration is overview-and-drill-down. First, a user asks for an overview of the whole data set. Once the user finds an interesting subset of the data, the user will drill down into that subset and render it at a more detailed resolution.
This process may continue until individual data records are retrieved or until the user has reached his or her permitted resolution. To facilitate overview-and-drill-down, density representations of data should be available at multiple resolutions. The user should be able to change between these resolutions in a way that is easy to understand and track. Unless the aggregated data are pre-computed, however, such a user interaction simply transfers the underlying computational requirement from the visualization client to the backend database and, while addressing network traffic issues, does little to enhance visual interactivity.

In relational databases, density data are prepared through data aggregation queries. A data aggregation query puts data records into groups and applies aggregation functions to each group. Aggregation functions are well-defined in SQL. Each group is characterized by a range of values of each aggregation variable and, thus, represents a hyper-rectangle in high dimensional space. Data aggregation is an expensive database operation which often involves sorting and merging of data records. In order to support real-time user interaction, aggregated data should be pre-computed and stored in the database. This is a computationally expensive database operation.

However, data aggregations can be organized into a hierarchical structure based on data resolution. One problem is how to store these aggregated data so that they can be efficiently accessed by visualization tools. During the past few years, we have focused on data access methods (see surveys in [42] and [6]) to support multiresolution data aggregation. We have found that a partition-based high-dimensional tree index offers an excellent vehicle to organize and piggyback the data aggregated at multiple resolutions, provided that the data have been aggregated according to the regions represented by internal nodes of the tree.
index. We have further found that major database problems (multiresolution data aggregation, optimization of range queries and other interaction-driven user queries) in visual data exploration can be properly answered by index-only queries on such an indexing structure. Such a tree index has two functions:

1. Organize multiresolution data aggregation

2. Accelerate high dimensional range queries issued by user interactions.

We call the tree index a data aggregation tree. It supports visual browsing, drill-down, as well as range queries.

3.3 Briefly Access Methods

Database and algorithm communities have spent considerable time on management and access methods of multidimensional data. Complex structures such as lines, rectangles, polygons become input and output to operations which are computationally more expensive than operations in Relational Database Management Systems. Thus, potent dynamic structures are necessary for insertions, deletions and updates. For large data, in-memory algorithms are inadequate and major modifications for existing ones are inevitable. Thus, large multidimensional and spatial data management demands I/O efficient external memory algorithms. Vitter in [97] provides a comprehensive description of design and implementation of external memory algorithms and analysis of parallel disk model (pDM). Since spatial proximity is not preserved in multidimensional data ordering, application of traditional database indexing to multidimensional data is difficult. In spatial databases, there
is no standardized spatial algebra or query language like SQL. However, there exist special spatial queries such as exact match, point, range or window, intersection, enclosure, adjacency, and nearest neighborhood [6].

A tree index provides a hierarchical organization of data, where internal nodes represent ranges of data and leaf nodes point to individual data records. On secondary storage, tree nodes are organized into disk blocks for the purpose of efficient I/O operations. B-tree [15] and its variations have been widely used for one dimensional indexing. It has excellent worst-case properties such as logarithmic cost of exact-match search, balanced height, and guaranteed minimum space occupancy. The high dimensional indexing problem is defined as to create an indexing structure on secondary storage for high dimensional data such that the structure should be symmetric on all dimensions and should preserve the same set of worst-case properties of the B-tree. This problem has been studied extensively for over thirty years and has been widely recognized as one of the persistent puzzles of computer science [71]. Many multidimensional data access methods have been proposed. However, none has succeeded in preserving all of the B-trees worst-case properties. Data access methods are categorized as spatial (SAMs) and point access methods (PAMS) by Gaede and Günther [42]. Structures in SAMS are to search for lines, and polygons in spatial data, while PAMS structures are to search points and may not support region, enclosure, or overlaps as in spatial methods.
3.4 High Dimensional Indexes and kdB-tree

We perceive relational data as distributions of points in high dimensional space. For multiresolution data aggregation, it must be guaranteed that every data point is aggregated exactly once at a given resolution. Therefore, the high dimensional index on which we choose to piggyback data aggregations must meet all of the following requirements:

1. It is a hierarchical data structure in order to carry multiresolution data aggregations.
2. It provides a point access method (PAM) instead of a spatial access method (SAM).
3. Regions represented by sibling nodes are disjoint with each other (no point is counted more than once).
4. A region represented by a node is totally covered by the union of regions of all of its child nodes (no point is missing).

There exist a few partitioned-based PAMs on secondary storage which comply with the above requirements. These include kdB-tree [81], LSD-tree [51], Buddy-tree [86], hB-tree [68], and Bkd-tree [80] (see Appendix A for a summary). These disk-based methods are based on the in-memory kd-tree [17] index structure. We have chosen kdB-tree as our primary data access method for its simplicity and its direct extension of the kd-tree.

A kdB-tree is basically a hard-disk implementation of the kd-tree, which is a height-balanced binary tree where each internal node partitions the hyper-rectangle represented by the node along a particular dimension. It improves I/O efficiency by storing multiple kd-tree nodes in one disk block. Unlike B-tree, however, a kdB-tree does not guarantee the
minimum page occupancy. Each page stores a splitting domain variable $d$. A page splits around a value of the domain $d$. The new pages have the splitting variable set. There are different strategies for splitting: cyclic, priority, adaptive, random, etc.

Figure 3.1 An Example kdB-tree: 2dB-tree

Figure 3.1 illustrates an example kdB-tree where $k$ is two and cyclic splitting is applied. Each node of a kdB-tree represents a hyper-rectangle in high dimensional space and can be stored in a single disk block. Each non-leaf disk block contains a collection of block pointers (each of which points to another disk block) that partition the hyper-rectangle into smaller hyper-rectangles. Each leaf node stores a set of individual data points (records).

3.5 Cascading Split Problem

The original kdB-tree suffers from a cascading split problem during the data insertion. It occurs as a trade-off of keeping the kdB-tree height balanced. When a block becomes full and a new entry is inserted, kdB-tree splits the block into two blocks in such a way that both contain similar numbers of child nodes. If an internal index block splits, then a split
may cascade to the child nodes and cause the child nodes to split (see Figure 3.2) until the leaf nodes are reached. As a result, it causes unpredictable performance of data insertion and does not guarantee minimum occupancy of internal blocks (low storage utilization).

The manner in which a splitting domain and a domain value are selected depends on the splitting strategy. Many variation and derivation of kdB-tree are proposed to address the problem, such as hB-tree, LSD-tree, Buddy-tree, KDBKD-tree [114], and perfect kdB-tree [67].
CHAPTER 4
DATA AGGREGATION TREE

4.1 Introduction

In this chapter, we will explain technical details and provide in-depth information about the data structure of the data aggregation tree (DA-tree) as a variant of a kdB-tree. Also, the organization of multivariate data aggregations with the DA-tree is analyzed. Splitting strategies, to avoid the cascade splitting problem, are presented too.

The DA-tree is implemented using a freely available templated library, TPIE [96], which provides an interface to external memory algorithms and structures. TPIE comes with a kdB-tree implementation, which we also investigate here. At the end, we compare both of the implementations based on the allocation of data structures for data aggregations.

In this chapter, we used block and node interchangeably because a node of an index tree is stored as a block on the disk.

4.2 Data Aggregation Tree

In order to piggyback the data aggregation information, we have revised the internal structure of kdB-tree blocks and called the new tree structure a DA-tree. Figure 4.1 illustrates the structure of a non-leaf block in a DA-tree. Each non-leaf block contains a header (info) and a body (elements) section. The header contains information of a hyper-rectangle represented by the block. It stores the low and high values on each dimension of the hyper-rectangle and the number of sub-regions (elements). The body encodes a bi-
nary kd-tree. Each non-leaf node of the kd-tree partitions the hyper-rectangle it represents into two smaller hyper-rectangles represented by its two child nodes. The non-leaf node has the format of (dim, val) which records the partitioning (splitting) dimension and the partitioning value. Data aggregation values are kept in leaf nodes of the kd-tree. Each leaf node represents a hyper-rectangle and has the format of (ptr, aggregate-values), where ptr is a block ID pointing to a disk block of the DA-tree representing a hyper-rectangle. Aggregate-values represent a list of data aggregation values and they are COUNT, SUM, MINIMUM, and MAXIMUM of all data records within the hyper-rectangle. The kd-tree nodes in a disk block are stored in a compact preorder format. Pointers between kd-tree nodes are not stored in order to save space. Thus, it yields maximization of the fan-out (number of sub-regions) of non-leaf blocks of the DA-tree. Because the kd-tree is a binary tree, the number of leaf nodes is always one more than the number of non-leaf nodes.

The hyper-rectangle range information, stored in the header of each block, can be computed top-down using the header and partitioning information contained in the parent block. Data aggregation values in a block in contrast to the information can be computed bottom-up from the corresponding aggregation values in its direct child blocks. The user decides which aggregation values are stored when building the DA-tree.
To avoid the aforementioned cascading split problem 3.5, we adapted a splitting strategy which is similar to LSD-trees. (1) When a leaf block is full and has to split, it splits at the median value of all data records within the block along the longest dimension of the hyper-rectangle it represents. Therefore, a leaf block splits into two leaf blocks with an equal number of data records. (2) When a non-leaf block is full and has to split, it splits at the root node of the kd-tree it contains. In other words, it splits by following the first partition of the hyper-rectangle represented by the block. This splitting strategy avoids cascading splits at the cost of creating an unbalanced tree. The strategy keeps hyper-rectangles in a DA-tree as hypercubic as possible, which will help us to explore data at a consistent resolution by visiting nodes at similar levels of the tree. As new data records are inserted, a DA-tree grows in a bottom-up manner.

A unique concern of DA-tree is how to update data aggregation values when new data records are added. This is not a problem as long as the existing indexing structure remains static. A DA-tree is first loaded with data, and space is reserved at non-leaf blocks for data aggregation values. Once all data records are inserted, data aggregations at non-leaf blocks are updated in a bottom-up manner. Aggregation values in a disk block can be computed directly from the corresponding aggregation values in its child blocks. When a new data record is inserted into a leaf block of a DA-tree, it updates data aggregation values in all of its ancestor blocks. This works nicely for data insertion as long as the existing tree structure remains static.

In general, building a tree takes a long time for large data. To overcome this problem, different bulk loading algorithms are proposed. Bulk-loading is to build the high dimen-
sional data index from its initial empty state. Some bulk loading algorithms and proposals in the literature related to loading high dimensional indices of large data can be found in [17], [3], [30], [18], and [31]. There are sort-based, buffer-based, sample-based, and path-based bulk loadings. Most of the bulk loadings are variations of sort-based. The data is sorted during a pre-processing time and then each tuple or record is inserted one by one. Bulk loading could be a problem for partitioned-based high dimensional tree indices of large data sets. So far, most high dimensional bulk loading techniques are restricted to build index trees with overlapping sibling nodes.

Due to the time restriction, we couldn’t invest more on bulk loading in our project. Nevertheless, buffer-tree bulk loading discussed in [31] is worth investigating and integrating to the cost of extra buffer space. Also, another promising approach applicable to partition-based tree indices is path-based bulk loading of [30]. It first builds an in-memory tree index by using a portion of data records. The in-memory index is then flushed to disk and is used to classify the rest of the data records into buckets. The algorithm then grows the index by recursively repeating the above process to each bucket.

Dynamic maintenance of an index is another problem. For example, in an insert operation, cascade splitting of nodes in the subtree causes an update of all aggregated information of nodes. Overflow nodes, from insertion node to the root, require a lot of aggregated data updates. In delete operations, restructuring of nodes triggers aggregate data update again. We are investigating techniques to address these problems by using statistical information of the data and by re-balancing the tree during regular maintenance.

For massive data sets, the number of the internal nodes grows so large such that it
doesn’t fit into the memory. Therefore, external memory algorithms and data structures become a must when in-memory indexing data structures become inadequate to implement a DA-tree.

4.3 External Memory and TPIE

Secondary memory requires efficient external memory (EM) algorithms [97]. Following points have to be taken into consideration.

1. EM algorithms are not as simple as in-memory algorithms, because most of the tree implementations are based on the assumption that internal index fits into the memory. For large data, this assumption doesn’t hold.

2. Communication between internal memory and slower external memory is a performance bottleneck. That is the reason behind EM algorithms which minimize excessive I/O operations to improve the performance. Caching and prefetching play a crucial role and I/O operations require careful programming.

Vitter et al. [98] proposed the Parallel Disk Model (PDM) to design EM algorithms. A Transparent Parallel I/O Environment (TPIE) system [9] facilitates the implementation of I/O efficient programs. TPIE is a set of templated classes and functions written in C++. It provides a programming environment for the development of EM algorithms and data structures. The lowest level of the library is the block transfer engine (BTE) which organizes and accesses the disk blocks. It is based on a single PDM. As of now, BTE supports three disk access mechanisms: \texttt{mmap} (uses memory maps) \texttt{mmap()} and \texttt{munmap()}.
functions), \texttt{ufs} (uses system \texttt{read()} and \texttt{write()} functions) and \texttt{stdio} (uses standard I/O functions). Supported block operations are create, delete, read, and write. It is this layer that implements functionality of a block collection. The top layer of the library is the application method interface (AMI). It provides the services of the BTI to the applications through its block collection class and its block class. Among its other main functions and interfaces are external memory algorithms (scan, sort, merge), I/O efficient data structures (b-tree, kd-tree, kdB-tree, queue, matrix, point, etc.), and a cache for disk blocks in main memory. As of now, AMI's cache manager uses least recently used (LRU) as its caching algorithm. However, it is extendable to derive new caching methods. As a case-study, the TPIE library comes with a kdB-tree implementation. Nevertheless, we have considerably reorganized the internal data structure to implement DA-tree instead of straightforwardly affixing aggregations to internal nodes of the tree as we discuss more in the coming sections.

\textbf{4.4 TPIE's kdB-tree Implementation}

TPIE comes with an implementation of a kdB-tree [9] as a case study to show the usability and performance of the library. In this section, we will summarize some structural definitions of that implementation.

TPIE defines a kdB-tree class as template\texttt{<class c, size_t d> class KdTreeNode}.

The kdB-tree class is comprised of two block collections of the AMI's templated collection class AMI\_collection for storing internal (non-leaf) nodes and leaf nodes as depicted in Figure 4.2. Both of the node classes extend AMI's templated block class of
AMI block<\textit{E}, I> where the template parameter \textit{E} is the type of elements and \textit{I} is the information part (header) of a block. Each block is stored in one logical disk page. A logical page can be an integer multiple of a physical disk page of an OS. Users can set a logical page coefficient through TPIE when building a tree. In this writing, when we use disk page, we mean logical disk page, not necessarily the actual OS's disk page. The following is the declaration of the nodes:

\begin{verbatim}
template<class c, size_t d>
KDBtree_node: AMI_block<region<c,d>, kdbtree_node_info>

Template<class c, size_t d>
DBtree_leaf: AMI_block<point<c,d>, kdbtree_leaf_info>
\end{verbatim}

Briefly, elements, \textit{E}, of a leaf block are a set of \textit{d} dimensional points of type \textit{c}. The information part, \textit{I}, contains the number of points inserted to the region, a pointer to the

Figure 4.2 TPIE's kdB-tree Implementation
next leaf (sibling) node and a splitting dimension for the next split.

Elements of a non-leaf (internal node) block are a set of regions of $E$ and its information $I$ contains the number of children, weight information, and a splitting dimension. Each region contains a $d$-dimensional box which is called a bounding box, a pointer to a subregion which is a block ID, and the type of pointer (internal node or a leaf node). A bounding box keeps low and high bounds for each dimension. Initially all the dimensions are unbounded and set to minus-plus infinities. That means the region is not split. When a region splits, the value of the splitting dimension is set. The region on the smaller side of the split sets a flag to indicate it is high-bounded, while the other split sets its flag to indicate it is low-bounded. The information of an internal node block contains the number of the node and the number of points stored in the subtree descending from the current node.

There are three different splitting strategies defined: cyclic, longest span, and random. Both the internal node and the leaf node apply the same splitting method. The method is specified by the user before building the tree. The leaf node is sorted on the given dimension first and then split. Splitting of an internal node, however, may end up with a cascading splitting problem.

4.5 DA-tree Implementation with TPIE

We altered the structure of non-leaf nodes of the kdB-tree to create the proposed DA-tree for multiresolution data aggregations. While a kdB-tree splits high-dimensional data space into smaller sub-spaces, we accumulate and compute aggregation functions for each sub region. The new structure adds selected aggregate data, speeds-up the query fetching,
and reduces redundant data to keep the fan-out of non-leaf nodes high even after adding new fields. Note that, in the coming lines, we used block and disk block interchangeably.

Figure 4.3 DA-tree

Figure 4.3 illustrates the internal organization of a DA-tree. In the drawing, NODE INFO and LEAF INFO fields are the information I parameter of the AMI template class AMI_block<E,I>. The NODE INFO field keeps the number of subregions (children) and the bounding box of the region. The LEAF INFO field of a DA-tree preserves the number of points inserted, the number of distinct points inserted, a pointer to the next block, and a splitting dimension.

Non-leaf nodes of a DA-tree exhibit an expanded kd-tree which is actually encapsulated in a very compact format in the disk page whose structure is shown in Figure 4.1. The kd-tree in a non-leaf node is called an internal kd-tree (ikd-tree).

- Non-leaf nodes of an ikd-tree hold dimension and value of splits (d, Val) path.
• Leaf nodes of an ikd-tree maintain pointers (blockID) to other nodes (subregions) of DA-tree and type of pointers (leaf or non-leaf). They also maintain aggregation data of the subregions.

Subregions of a region (non-leaf nodes of DA-tree) are pointed or linked by leaf nodes in an ikd-tree. The bounding box of each subregion is computed easily using the bounding box of the current region which is stored in the NODE INFO. Starting from the root of an ikd-tree to a leaf node, we cut (split) the bounds of the region on dimension \(d\) at value \(val\) of \((d,val)\) pair which are collected from each non-leaf nodes on the path.

Declaration of an ikd-tree in TPIE is not straightforward, because two different classes (leaf and non-leaf) need to be stored in a TPIE block structure. However, as we described before, a TPIE block is intended to store homogenous elements. That is, only one class template parameter, \(E\), is allowed to be stored. We solve the problem by assigning a 1-byte (char) element to the \(E\) parameter of the class AMLblock<E,I> to claim all space left after allocation of the information, \(I\), part.

Reserved space of a disk block is utilized by encapsulating an ikd-tree in a compact format. An ikd-tree is stored implicitly as an unordered binary tree [33]. By “implicitly”, we mean that the structure of an internal tree is recorded as compactly as possible instead of explicitly recording all of the tree information like links of nodes in the tree. We called writing an ikd-tree implicitly to the disk block “serialization”. It writes a tree in preorder to the reserved space in a block. It is an unordered writing of the internal nodes and leaf nodes, that is, they are intermixed. All nodes are written sequentially to the reserved space.
preceded by 1 byte type information which indicates if it is a leaf or non-leaf node of the ikd-tree. The very first byte of the block space allocated for elements $E$ of a DA-tree internal node is the type of the root node of the internal tree.

Obviously, a serialization needs reconstruction of the internal tree from the implicit representation. We called this reverse process “de-serialization”. First, it reads the first byte of $E$ as the type of the root node. Then, it recursively builds by reading the left and right children until reaching ikd-tree’s leaf nodes.

A leaf node of a DA-tree contains individual records. Its data structure is the same as the structure of a leaf node of the kdB-tree discussed in Section 4.4.

4.6 Splitting in a DA-tree

Splitting high dimensional index structures may not be a clear-cut process. Especially, an internal node splitting may end up being caught by the pitfall of cascade splitting (Section 3.5). So, splitting compels us to either avoid it happening completely or minimize the performance uncertainty as much as possible.

Splitting strategies of the TPIE’s kdB-tree are cyclic, longest span, and random. In a DA-tree, users can set different methods for leaf and non-leaf nodes, whereas in the kdB-tree implementation they are the same.

For leaf nodes of a DA-tree, in addition to the aforementioned splitting methods, we introduce a new one: a weighted longest span splitting. In this splitting, users can assign a weight or a factor for each dimension of data before building a tree. During the splitting of a leaf node (a region), the weight $W_i$ for dimension $i$ (where $i < d$) is applied to the
The dimension with the longest span is chosen as the splitting dimension. The median of the values is the splitting value of the splitting dimension. If the same values (points) scatter around the median, another median is chosen which distributes the number of points (records) as equally as possible to the splits. Thus, during the search of a DA-tree, only one child is followed. Otherwise, for equal-value medians, both left and right children of a kd-tree need to be searched.

In splitting internal nodes of a DA-tree, we choose a splitting method that does not exist in the kB-tree's implementation. As some other trees like LSD-trees and Buddy-trees do, when a non-leaf node becomes full, it splits from its first split which is the root of a ikd-tree. We called it root-split splitting (Figure 4.4). In the figure, a new ikd-tree leaf node to be inserted to the bottom left node causes a split. First, a new node is created or a new block is allocated (rightmost node in the figure). Then, the tree descending from the right child of the root of the splitting node is moved to the new node while the left child of the root
becomes the new root of the splitting node. After that, a new ikd-tree leaf node is created which points to the new node created. This new leaf node is inserted to the parent DA-tree node (top node in the figure) by splitting its ikd-tree leaf node which points to the split DA-tree node. If the parent node is also full, then the root-split is applied to the parent node. If the splitting node is the root of a DA-tree, then, a new node is created and it becomes the new root of the DA-tree. The new root contains a 2-leaf ikd-tree which points to the split regions.

![DA-tree](image)

(a) Root-split Splitting  (b) Adaptive Splitting

**Figure 4.5** DA-tree Non-leaf Splitting Methods

The advantage of the root-split is that the cascade-splitting problem doesn’t occur. Its disadvantage is that the DA-tree is not height balanced which depends on the order of insertion. However, we suggest an *adaptive splitting* method to reduce the effect of root-split. The adaptive splitting splits a non-leaf node at an internal node of its ikd-tree instead of strictly at the root. The worst case of an unbalance tree is when all splits happen to be on the right most child of a kd-tree before a node becomes full. Then root-split builds a DA-tree as in Figure 4.4(a) while adaptive splitting builds as in Figure 4.4(b). If the fanout
of internal nodes is $f$ and the number of leaf nodes is $L$, then the number of internal nodes required for root-split is $N_1 = L - f + 1$ which is approximately the same as the leaf nodes. However, adaptive splitting may yield $N_2 = \frac{L}{f+1}$ blocks if always split from the right most child. So, in such a scenario, adaptive splitting can reduce the number of internal blocks created about a factor $\left(\frac{N_1}{N_2} \approx f\right)$ of its fanout over root-split’s block count.

As a result, splitting strategy affects number of nodes, space utilization, and the height of the tree. By reducing height of the tree, we increase search/query performance of a DA-tree.

4.7 Discussion

In this section, we will compare the DA-tree implementation to the TPIE’s kR-B-tree implementation if data aggregations were embedded.

To build the multiresolution data aggregations, aggregate data (min, max, sum, count, etc.) of each region have to be accumulated and stored. There are two alternative places to store the aggregate data. (1) Each region keeps its aggregation in the information field. (2) The parent node of a region in the hierarchy stores the aggregation information for each sub-region along with its child pointer. Each method has its pros and cons. The former reduces the fanout of non-leaf nodes and leaf nodes whereas the latter leaves the fanout (capacity) of leaf nodes unchanged. The fanout of non-leaf nodes suffers significantly more in the second one than the first one. However, in terms of search/query performance, storing aggregations in parent nodes works much better because the selection of a region based on any aggregation value is decided without issuing an I/O (read) operation to the
disk. Reduced I/O increases query performance greatly. Thus, if the fanout of non-leaf nodes is $f$, then $f$ disk block reads are avoided. We applied the second approach to the DA-tree and piggybacked the aggregations in leaf nodes of a ikd-tree. The TPIE's kdB-tree could adopt both modifications but the first one would be easier to code.

The TPIE's kdB-tree keeps redundant information in its internal nodes. Each element, $E$, of a non-leaf node has a bounding box for a subregion it represents. Thus, total space used to maintain split and region information is $space_1 = 2 \times d \times f$ where $d$ is the number of dimensions and $f$ is the fanout of an internal node. In a DA-tree, each NODE INFO has one bounding box and each ikd-tree maintains the split information in its internal nodes. Total space for bounding box and splitting information in a DA-tree is $space_2 = (2 \times d) + 2 \times (f - 1)$. So, our non-leaf node implementation with a ikd-tree gains $d \frac{space_1}{space_2} \approx d$ times more compact space utilization for bounding boxes than the TPIE's implementation.

Developing various non-cascade splitting strategies with a DA-tree is doable because of its internal structure.

Finally, a DA-tree intrinsically embodies intermediate subregions, thus its resolution levels or details for multiresolution data aggregation are high. At first, one might think that the data resolutions are the children of the internal nodes (leaf nodes of the ikd-tree), because data aggregations are stored in the leaves. However, an internal node of a ikd-tree also represents a resolution level whose bounding box is calculated top-down from the root and whose aggregations are accumulated from its children bottom-up. Thus, each internal node of a DA-tree adds an additional $fanout - 2$ resolutions. On the other hand, the generic kdB-tree tree, without ikd-tree used, only retains resolutions for a region that
a node represents, that is, the next resolution level will be its child. Thus, as illustrated in the Figure 4.7, while the kdB-tree node without ikd-tree has only 10 resolution for the region: 1 for the region and 9 for its children, the internal node of the DA-tree has an additional 7 regions as shown on the right side of the figure.

In conclusion, a DA-tree utilizes disk storage for internal index of a high dimensional data structure efficiently and in a very compact way when data aggregations are being backed.
CHAPTER 5

VISUALIZATION AND SOFTWARE ARCHITECTURE

5.1 Introduction

Multiresolution data aggregation represents a new format of data for data visualization. Because data aggregations are stored in multiple resolutions at internal nodes of a tree structure, we can visually explore high dimensional data by directly accessing internal blocks along a cut of the tree. Existing data visualization techniques need to be carefully evaluated for their effectiveness. An applicable visualization technique must be able to deal with high dimensionality, accept this new format of input data, and support user interactions such as zooming, picking, and brushing to preclude incompetence of large data visual analysis.

Figure 5.1 Client-server Architecture
In this chapter, we will introduce the architecture of a client-server software we developed to demonstrate the effectiveness and usability of the proposed density based methodology (Figure 5.1). On the server side, a simple multi-threaded data management system is created to manage high dimensional index structure of a DA-tree. We called it kdB-tree Management System (KDBMS). Details of KDBMS and SQ specifications that it supports to query multiresolution aggregations will be pointed out.

On the client side, we created a visualization framework called mVis, which stands for multiresolution visualizer, multi-dimensional or multiple visualization. mVis clients connect to the server over TCP/IP network to query KDBMS. In this tool, we have chosen scatterplot, 3D footprint splatting, parallel coordinates, and scatterplot matrices as our visualization techniques and integrated them with various interaction methods. We extended them to render multiresolution data aggregations. In addition to the visualization techniques, we also explicate extendability and simplicity of mVis for integrating new components to the framework.

5.2 The Server

The server of the architecture is composed of an input file, a metafont, TPIE library, and KDBMS. KDBMS accesses to the external disk using the Access Method Interface (AMI) of TPIE to create and query created tree structures. An input file and information about the data (metafont) are the inputs from the user to KDBMS to build a kdB-tree. Our kdB-tree implementation is a DA-tree as described before, so we use 'kdB-tree' and 'DA-tree' interchangeably unless otherwise specified through the text in this chapter.
5.2.1 Input File

An input file is one of the core elements to build the tree. It contains the $d$-dimensional data points or $d$-variable records. The structure of the input file is the TPIE’s stream format. TPIE comes with a data generator tool that converts ASCII files into an AMLSTREAM output or generates new data stream files with random points. A TPIE stream is defined as "an ordered collection of objects of a particular type, stored in external memory, and accessed in sequential order." It is similar to creating any C++ stream except the data placed in a TPIE stream file is stored on the disk (see TPIE manual, Chapter 4, for details).
5.2.2 Metafont

Information about the input data to build a kdB-tree is stored in a metafont. A .meta file contains information about variables (columns) of an input data and various specifications which affect building a kdB-tree and visualizations. In Table 5.1, the description of each field in a metafont is specified.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the variable</td>
</tr>
<tr>
<td>description</td>
<td>Description of the variable</td>
</tr>
<tr>
<td>data type</td>
<td>Data type of the variable in the input file. One of: text(T), integer (I), float (F), double (D)</td>
</tr>
<tr>
<td>precision</td>
<td>Number of digits after decimal point for numeric variables</td>
</tr>
<tr>
<td>scale factor</td>
<td>Input data can optionally be normalized into [0, scale Factor * 1] domain. This option is the range multiplier for the mapping. Variables with larger ranges have a high chance of splitting by longest span splitting. (default=1)</td>
</tr>
<tr>
<td>domain min</td>
<td>Minimum value of the variable (default=0)</td>
</tr>
<tr>
<td>domain max</td>
<td>Maximum value of the variable (default=1)</td>
</tr>
<tr>
<td>variable type</td>
<td>Type of the variable. One of the following: continuous: quantitative numeric variable. Usually measurements take on any value within some range. discrete: quantitative numeric variable. Usually measurements are integers. ordinal: qualitative variable. An ordering exists between the categories. nominal: qualitative variable. There is no natural order between the categories.</td>
</tr>
<tr>
<td>sorted flag</td>
<td>Indicates if the data is sorted on this variable or not (true/false) (default=false)</td>
</tr>
<tr>
<td>missing value</td>
<td>If there are missing values in the data, replace them with this value</td>
</tr>
<tr>
<td>binning interval</td>
<td>Length of intervals if binning is enabled (for discrete variables only).</td>
</tr>
<tr>
<td>VCL flag</td>
<td>Value-name-label pair indicator.</td>
</tr>
<tr>
<td>{value-name label pairs}</td>
<td>0 or more VCL pairs to define ordering for nominal and ordinal variables. Order of VCL pairs is the order for the variable.</td>
</tr>
</tbody>
</table>

Table 5.1 Format of a Metafont

Metafont is static in nature. Once a metafont is set, it shouldn't be modified after a tree
is built. As we explain later, optionally, KDBMS can build trees by normalizing coordinates of points (values of variables) from a domain range $[domainMin_i, domainMax_i]$ \( \forall i \) where \( z < d \) into \([0, scaleFactor_i + 1]\) range. Thus, insertion operation might change the domain range for earlier points inserted by changing the tree. This change invalidates normalization of existing insertions. Our solution to address this range problem, without modifying the minimum and maximum domain, is the assumption of new values out of the range are “outliers”. They will be represented by a fixed minimum or maximum value in the database. The fixed number is a buffer range from domain min and domain max of the variable. So that, these outliers will be normalized and visualized by the visualizers.

In Figure 5.3, data flow in building a tree are depicted to clarify the explanations above.
Data_gen is the synthetic data generator that creates AMLSTREAM files or transforms plain text files. A stream file and a metafont of the input data are input to KDBMS. All or a subset of points of the stream make up the tree based on the server settings and the metafont options.

5.2.3 kdB-tree Data Management System (KDBMS)

Overview

KDBMS is a data management system to build, update, query, and perform other operations over a kdB-tree built using TPIE library. It is a lightweight system and similar to a DBMS but lack a lot of basic functionalities i.e. buffer management, logging, concurrency control and locking, and transaction manager. It is a multi-threaded environment implemented in C++. The server itself is made up of templated classes like TPIE interface. The number of variables, data type, and the list of variables to apply aggregation are passed as template parameters to the TPIE interface in compile time. To use in run time, there are many options that the user can set while building the tree. Some of the parameters are to modify fan-out and splitting strategy for leaf nodes and internal nodes, logical disk block size, etc.

Before inserting individual points, KDBMS applies pre-processing algorithms in sequence. Currently, it applies a data cleaning and a transformation. First, if a record contains missing values for its variables, they are replaced by predetermined values from the metafont. Thus, preset values for replacement are important for data validity or uncovering missing values later.
Second, normalization as a transformation in data preparation is applied to each record before insertion if the server's parameter is set. On visualizing data, various transformations are applied to data in domain space to visualize it in screen space. In many cases, objects are modeled in \([0, 1]\) space. Thus, in visualization of large data, normalization adds-up to the transformations and computations on the client either in the main memory or in its graphics card's memory. In order to alleviate normalization process, KDBMS can optionally build a normalized tree. For this purpose, each point in \(d\)-dimensional domain space is normalized into \([0, 1]\) space before inserted to the tree. Optionally, variables can be mapped to a larger span by specifying a coefficient for the upper bound \([0, C_i]\) (where \(i < d\)). In the later case, only one division is needed to map the data into visualization's \([0, 1]\) coordinate system.

Other pre-processing algorithms such as data cleaning, dimension reduction (PICA [60], Isomer [92]), discrimination can be injected into pre-processing pipeline. A user interface for ordering and applying miscellaneous data preparation algorithms enables automating the process in KDBMS. The addition of such a dynamic feature is under consideration in a future edition.

KDBMS interacts with TPIE interface to build and query a kdB-tree. The tree is stored on the disk and TPIE provides read/write access to the disk. An input file as described above and its meta file (.meta) are input to the KDBMS to build the kdB-tree. Once a tree is created, KDBMS loads it during its startup and starts listening to a TCP port for client connections. An agent is dedicated for each connection. Each agent processes queries (range, point) from the clients and sends the results back to the requester over TCP/IP network. Figure 5.4 illustrates the internal design of KDBMS that shows the communication
between different interfaces and threads. In the implementation, POSIX threads (pthreads) are used. Threads require less overhead than "forking" or "spawning" a new process. That is a basic reason to choose threads over processes to implement KDBMS.

![Internal Design of KDBMS](image)

**Figure 5.4** Internal Design of KDBMS

**KDB Interface**

A level built on top of TPIE AMI interface to provide kdB-tree functionalities: built, read, write, query, get tree information, etc. This object is shared by the agent threads. Since our design allows querying only and no write (insert, delete) operations allowed to the clients at this time, the threads do not conflict with each other and no deadlocks occur. Future implementations will apply locking mechanism to allow concurrent updates in a
dynamic setting.

Connection Manager

Connection manager is the main process which creates other threads and builds loads the kdB-tree. There are many options that can be specified on the command line. Most of them are directly passed to the TPIE layer to build the tree. Some are required by the KDB interface hence we have a connection from the console to the interface as indicated in the figure above. Port number, metafont file name, maximum block fanout, leaf and non-leaf block factor, maximum memory size for the memory manager are some of the options. The programming environment is a templated environment. That is, we need to specify few options during the compile time. Such as, the number of variables (or number of dimensions), whether the input has a data field (satellite data) or not, if the input values will be normalized or not. Main process builds/load a kdB-tree using the command line options. The first user command (below) creates a new kdB-tree and then loads it while the second line loads an existing kdB-tree (\ldots \texttt{> denotes options to be provided}):

\texttt{/kdb -iu <inputStream> -ot <outputStream> -M <metafile>}

\texttt{/kdb -it <inputTree> -M <metafont>}

After loading a tree, KDBMS creates a signaler thread and a console thread (both discussed below). Then, the main process acts as a connection manager and starts listening to the specified TCP port (default is 6000) for connections. When a connection request arrives from a client, the connection manager creates an agent thread and continues listening for new connections. It is the agent thread's job to communicate with the child after the
connection is established.

Signal Manager

The signal manager is a thread created by the main process to catch all the signals. None of the threads after creating this thread receives any signal. Only one thread manages all the signals. This makes the multi-thread programming easier. The user signals from the keyboard (i.e. Ctrl+C), OS signals and other interrupts are caught. Depending on the type of signal caught, the signal manager initiates a shut-down procedure as well.

Console Manager

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>display this help</td>
</tr>
<tr>
<td>i</td>
<td>execute O/S system commands</td>
</tr>
<tr>
<td>h</td>
<td>display all server options including TPIE options</td>
</tr>
<tr>
<td>fill</td>
<td>update aggregate data in the tree</td>
</tr>
<tr>
<td>c</td>
<td>display current connections' information</td>
</tr>
<tr>
<td>d</td>
<td>delete the query result file or not (toggle)</td>
</tr>
<tr>
<td>i</td>
<td>display kdB-tree information</td>
</tr>
<tr>
<td>m &lt;limit&gt;</td>
<td>set max number of connections to the server</td>
</tr>
<tr>
<td>M</td>
<td>display metafont loaded</td>
</tr>
<tr>
<td>p</td>
<td>print the kdB-tree in DFS order</td>
</tr>
<tr>
<td>u &lt;file&gt;</td>
<td>perform queries from given ASCII file</td>
</tr>
<tr>
<td>s &lt;file&gt;</td>
<td>store built and tree information to the file</td>
</tr>
<tr>
<td>x</td>
<td>verbos/show more processing server details on the console</td>
</tr>
<tr>
<td>show &lt;bID&gt;</td>
<td>show information for the block with given block ID</td>
</tr>
<tr>
<td>cls</td>
<td>clear the console</td>
</tr>
<tr>
<td>b</td>
<td>buffer results to a file before writing to the socket (toggle)</td>
</tr>
<tr>
<td>x or q</td>
<td>shut-down the KD BMS Table 2List of Console Commands</td>
</tr>
</tbody>
</table>

Table 5.2 Commands List of the Console

The console manager is a text based command line processor to interact with the user. A separate thread, created by the main process, performs the tasks. The user can perform server-side activities through this interface such as changing the server options, collecting
and displaying kdB-tree information, executing SQL statements, and listing current connections. The current version of KDBMS is designed based on the assumption that the kdB-tree is static and there are no deletions after building the tree. Therefore, it provides limited functionalities. Hence, the list of commands that the console manager handles is minimal. Yet, it can be extended to have GUI, user management for security, performance tuning, and other DBMS functionalities in future releases.

This thread is responsible for collecting tree information through the KDB interface. Table 5.2 is the list of supported console commands:

Agent Threads

An agent thread is like a factor acting on behalf of a client connected to the server to execute the client's queries. In the KDBMS thread processing 5.4, the main process later becomes the connection manager that waits for connections from the clients. When a new connection comes in to the server, the manager creates a new agent thread. The agent thread communicates with the client over TCP protocol. The agent sends the payload to the client during connection establishment and waits for the queries to execute. After receiving the query, the agent calls for functions provided by the KDB interface. The results are written to the socket of the agent and sent to the client. Once the last record is sent to the client, the agent waits for another request. This process continues until the client closes the connection or the server shuts down. In the latter case, the agent sends a message to the client to close the connection. Thus, no open communication is left behind after a server shutdown. Details of the network messages and the SQL statements can be found in
sections below.

5.2.4 Data Generator

TPIE has a data generator/convertor to create stream files to be used with the library. It supports conversion of ASCII data files into TPIE streams. Also, it generates random points in $d$ dimension within specified domains (bounding boxes or regions). Distribution of random points can be uniform or diagonal. In addition to these methods, we added two clustered distribution techniques: Satellite clustering and random clustering. The first one creates a cluster at the center of the data space on all dimensions, and a specified number of smaller clusters are randomly placed to the intersection of the dimensions, so that it will create satellite clusters around the major cluster. The second algorithm is similar to the first one except there is no center cluster and the location of satellite clusters are randomly generated. The radius of the center cluster and the satellite clusters can be specified from the command line and can be different on each dimension. Also, some random points are scattered around to create noisy data in the data set. Furthermore, the number of points distributed to the clusters and noise data are specified on the command line as a percentage of the size of the data set. The default data type for variables is double float. However, the data types can be changed as long as it is the same for all of variables.

5.3 Network Messages and SQL Specification

5.3.1 Network Messages

Clients and the server of the architecture communicate over a TCP/IP network. In the application layer, defined custom message types guide the message interchange between
the agents and the clients. Network message types defined in the system are listed in Table 5.3.

<table>
<thead>
<tr>
<th>Message Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NW_CONNECT</td>
<td>open a connection</td>
</tr>
<tr>
<td>NW_DISCONNECT</td>
<td>close a connection</td>
</tr>
<tr>
<td>NW_QUERY</td>
<td>send a query with the type provided: only range (window) queries at this time</td>
</tr>
<tr>
<td>NW_GETMETA_DATA</td>
<td>request metadata</td>
</tr>
</tbody>
</table>

Table 5.3  List of Network Message Codes

5.3.2 Output Format

There are three different formats of query results to be sent by an agent to its client: points, hyper-rectangles (volumes), or mixture of these two. The structure of each result set is different and listed in Table 5.4:

<table>
<thead>
<tr>
<th>Type</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>point</td>
<td>$&lt;n-D \text{ Vector}&gt; &lt;\text{level}&gt;$</td>
</tr>
<tr>
<td>hyper-rectangle</td>
<td>$&lt;n-D \text{ lowVector}&gt; &lt;n-D \text{ highVector}&gt; &lt;n-D \text{ aggregate level}&gt;$</td>
</tr>
<tr>
<td>mixed</td>
<td>the same as hyper-rectangle except points have the same highVector as lowVector</td>
</tr>
</tbody>
</table>

Table 5.4  List of Query Result Formats

The point formatting is used when all records are fetched from the leaf nodes while the hyper-rectangle format is used when all records are fetched from the non-leaf nodes. Otherwise, the last format is used which represents records from leaf-nodes (points) as if they were hyper-rectangles whose high and low bounds are the same.

For a point query, leaf nodes are fetched from the disk and detail of individual records is returned. For a volume query, when a leaf node is accessed, its region information (minimum and maximum of all points for each dimension) as a hyper-rectangle is returned.
5.3.3 Resolution Levels

Internal nodes of a DA-tree constitute multiresolution levels. There are different ways to determine or distinguish resolution levels more meaningfully than tree depth level. For example, resolution of a data hyper-rectangle can be defined by its geometric measure or by its data aggregate measure. A DA-tree is conceptually a binary kd-tree. A resolution measure associates a function \( f(v) \) with each node \( v \) of the kd-tree. The only requirement is that \( f(v) \) is anti-monotone to the level of the tree, that is, \( f(v_i) \geq f(v_j) \) if \( v_i \) is an ancestor node of \( v_j \) in the tree. Given a particular resolution threshold \( r \), we define a cut \( C(r) \) of the data aggregation tree as

\[
C(r) = \{ v | f(v) \leq r \text{ and } f(parent(v)) > r \}
\]

\( C(r) \) can be intuitively envisioned as a horizontal cut across the data aggregation tree. In each path from the root to a leaf of the tree, there is exactly one node included in \( C(r) \). The union of hyper-rectangles represented by all nodes in \( C(r) \) is the whole volume. The position of \( C(r) \) changes smoothly as the value of \( r \) changes. The larger \( r \) is, the closer the cut is to the root of the tree; the smaller the \( r \) is, the closer the cut is to the leaves of the tree. To browse aggregated data at the resolution specified by \( r \), we need to access tree nodes in \( C(r) \). By changing the value of \( r \), we are able to explore a data set at different resolutions. Zooming and drill-down are supported by changing the value of \( r \). Because \( r \) is continuous, we envision that changing \( r \) will provide smooth transitions in data exploration. This process may continue until leaf pages are accessed and individual data records are retrieved, given the permission to access these individual records.
The following is the list of ways to define resolution levels.

a) *Levels of the tree:* This is the most natural way to identify resolutions. In this case, the maximum number of resolutions equals to the height of the tree.

b) *Data Aggregation Measure:* The number of data records, sum, minimum, or maximum of a dimension, and other aggregations stored can be candidates for determining resolutions. COUNT aggregates the number of points stored in a region. It can be used to determine the hierarchical levels of the resolution because it is a monotonic agglomerative information. The top most level is the highest in the hierarchy and is the root of the tree that contains the number of all points. Leaf nodes of the tree are at the lowest level storing only the count of inserted points at that node. Similarly, sum, minimum, and maximum of values of a variable within a region can be used as a resolution parameter.

c) *Geometric Measure:* This type of resolution considers monotonic decreasing or increasing geometric measures to select data, such as volume, minimum span or maximum span of a given variable.

d) *Derived Measure:* Other resolutions levels can be generated from the earlier resolution identifiers. Two examples are density and average. When data is accessed, values are computed and then the selection criteria is applied. Resolutions by density are decided by the density of the hyper-rectangles. The density of a hyper-rectangle is its volume divided by its count (the number of records) \((\text{density} = \frac{\text{volume}}{\text{count}})\).
density of the root is normalized to 1. Thus, densities of other regions are multiplied
by \( \frac{\text{volume of all space}}{\# \text{ of all records}} \). Resolution by average computes \( \text{average} = \frac{\text{sum}}{\text{count}} \) before
applying \( C(\text{average}) \) to each hyper-rectangle.

A range query starts at the root of a DA-tree. The query region is then compared
with the hyper-rectangle represented by each node of the tree. If they intersect, the search
goes recursively to the block pointed to by the node. The search continues until it reaches
nodes in \( C(r) \), where data aggregation values are retrieved. Since the DA-tree is not height
balanced, some leaf nodes will be reached earlier than other leaves on paths from the root
to leaf nodes during the fetching. In such a case, whether or not to output the leaf node
depends on the output type of the query. For volumes, the hyper-rectangle represented by
the leaf node is outputted. In this case, the limit of a cut (\( r \) of \( C(r) \)) denotes a lower bound
(threshold) for level, count, sum, etc., thus the union of regions selected is equal to whole
region and no region is left out. For point and mixed queries, records stored in the leaf
node are selected if they fall into the query region. Consequently, data fetched from hyper-
rectangles or individual points are not located at the same tree level, but it is an arbitrary
cut in the tree structure.

We suggest using the resolution level as another privacy measure for data access. Users
can be granted access to a resolution level and restricted beyond that limit for more de-
tail. Like so, only employees with appropriate clearance can view individual records while
others can see an overview to get an idea about the profile or distribution of the data.

5.3.4 SQL Specification

KDBMS supports the following SQL syntax for range and selection queries:
SELECT <fields> {TOP <recNumber>}
FROM <tableName>
LIMITBY <limitByOption>
WHERE (<fieldName> {IS} BETWEEN <minVal> AND <maxVal>
| <fieldName> {IS} = <value>
| LIMIT {IS} BETWEEN <minVal> AND <maxVal> )

In the SQL syntax, selection fields are the names of the variables of data as stored in meta files. The order of selected fields is protected in support of dimension re-ordering on the visualization client. Since current implementation of KDBMS supports single-table, using FROM clause is optional. LIMITBY statement specifies resolution level to fetch multi-resolution data or points and LIMIT statement sets the lower bound for fetching. WHERE statement defines a query window to restrict the selection. Thus, the former limits selection vertically in a kDB-tree, while the latter is horizontally.

<table>
<thead>
<tr>
<th>Token</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>all fields</td>
</tr>
<tr>
<td>SELECT</td>
<td>select fields</td>
</tr>
<tr>
<td>FROM</td>
<td>from table</td>
</tr>
<tr>
<td>WHERE</td>
<td>where condition</td>
</tr>
<tr>
<td>IS</td>
<td>equality operator</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>range condition</td>
</tr>
<tr>
<td>AND</td>
<td>conjunction</td>
</tr>
<tr>
<td>LIMIT</td>
<td>limit</td>
</tr>
<tr>
<td>COUNT</td>
<td>count points</td>
</tr>
<tr>
<td>VOLUME</td>
<td>volume points</td>
</tr>
<tr>
<td>MIN_SPAN</td>
<td>minimum span</td>
</tr>
<tr>
<td>MAX_SPAN</td>
<td>maximum span</td>
</tr>
<tr>
<td>SUM</td>
<td>sum</td>
</tr>
<tr>
<td>WHERE LIMIT</td>
<td>limit condition</td>
</tr>
<tr>
<td>&lt;dim&gt; &lt;d&gt;</td>
<td>dimension operator</td>
</tr>
<tr>
<td>one or more</td>
<td>one or more values</td>
</tr>
<tr>
<td>( )</td>
<td>one or zero (optional)</td>
</tr>
<tr>
<td>{ }</td>
<td>one or more values</td>
</tr>
<tr>
<td>*</td>
<td>select all fields</td>
</tr>
<tr>
<td>fields</td>
<td>select fields</td>
</tr>
<tr>
<td>recNumber</td>
<td>record number</td>
</tr>
<tr>
<td>meta</td>
<td>metafile</td>
</tr>
<tr>
<td>fieldName</td>
<td>field name</td>
</tr>
<tr>
<td>minValue</td>
<td>user limits</td>
</tr>
<tr>
<td>maxValue</td>
<td>user limits</td>
</tr>
<tr>
<td>value</td>
<td>user limits</td>
</tr>
</tbody>
</table>

Table 5.5 SQL Specification

Some SQL examples are as follows.

1. SELECT * FROM theServer WHERE elec BETWEEN 10 AND 120
2. SELECT elec, gas, water, hinc, FROM theServer
3. SELECT * FROM theServer
   LIMITBY COUNT
   WHERE LIMIT IS BETWEEN 1000 AND 9000
4. SELECT FLD3, FLD2, FLD5 TOP 1000 FROM theServer
   LIMIT BY SUM1
   WHERE LIMIT IS BETWEEN 1e4 AND 1e7

5.4 The Visualization Client
5.4.1 Introduction

In order to visualize and visually explore large relational data, we created a client visualizer tool to display multiresolution data aggregation. For the purpose of evaluating the usefulness of density-based input, we have extended existing visualization techniques. In this section we will discuss the detail of visualization of multiresolution data.

mVis reconciles various loadable components including data drivers and visualizers. Internal design and building block details of the framework are expounded in Chapter 6.

For visualizing large data by density-based methodology, we modified the following visualization techniques. First, we extended footprint splatting [103] with a two-step bootstrapping method to generate footprints of a high dimensional reconstruction kernel. A generated footprint reflects the projection of high-dimensional reconstruction kernel in the current view. Splatting footprints to the screen produces an image which gives a volume visualization of the data. Users can easily identify dense and sparse areas of the data in the visualization. Second, we extended traditional parallel coordinates [55] to density-based parallel coordinates where each aggregated hyper-rectangle is visualized as a translucent band across the parallel coordinates. All the translucent bands are then blended to visualize the data. These two extensions reduce visual clutter by displaying aggregated hyper-rectangles instead of individual data points. Third, we implemented scatter-plot matrices [8] to dis-
play the density-based hypercubes. This visualization mitigates visually understanding of data by displaying correlations of every pair of dimensions. All of the visualizers are concentric with their peer techniques which display individual data points. So, they seamlessly visualize point, volume and point-volume data.

5.4.2 Footprint Splatting

Volume rendering of data has been extensively studied and a few algorithms have been developed (see [72] for an evaluation). Footprint splatting [103] is an object-order volume rendering technique that directly renders 3D volumetric data. We have integrated volume rendering with 3D data projection [112] to visualize large relational data. Data hyper-rectangles are generated from the relational data by using data aggregation operations. The aggregated hyper-rectangles are organized by a high-dimensional indexing structure. In this paper, we extend our previous work to create footprints of high dimensional reconstruction kernel using a two-step bootstrapped method.

Footprint splatting works by projecting a high dimensional voxel to an image. The extend of its contribution to the final image depends on the projection of the corresponding reconstruction kernel. In footprint splatting, we usually use a Gaussian kernel because of its circular extend, smoothness, and quick attenuation in both spatial and frequency domains. A footprint is a projection of the kernel into the image buffer. The footprint splatting algorithm splats the Gaussian kernel of each data hyper-rectangle and blends colors and values at every pixel of the Gaussian kernel’s extend.

Traditional 3D footprint splatting works fine when the original volume data has three
dimensions. When we extend the algorithm to high dimensional splatting, however, how to compute the footprint of a high dimensional Gaussian reconstruction kernel becomes a problem. We have solved this problem by sampling the reconstruction kernel and then project each sample as a regular Gaussian distribution to a 2D image. In this way, we turn the process of generating a footprint into a simple reuse of the splatting algorithm. The final visualization of the data is a blending of footprints of hyper-rectangles, where each footprint is generated in a bootstrapping manner by sampling and projecting the reconstruction kernel. This technique is still applicable if the data are aggregated into unevenly partitioned hyper-rectangles.
Figure 5.5 shows the two-step bootstrapping process. The first step approximates a 2D footprint of a reconstruction kernel by sampling the kernel and applying the splatting algorithm to footprints of all samples. The second step visualizes data by applying the same splatting algorithm to blend footprints of all data hyper-rectangles. In the first step, we use a generic 2D Gaussian footprint (Figure 5.6(a)) which is symmetric in all directions. Footprint of a reconstruction kernel is computed by sampling the kernel and blending the footprints of all samples, each of which is appreciated by the generic 2D Gaussian footprint. Figure 5.6(b) gives three sample footprint images generated at different projections and axis alignments. In summary, footprint of a reconstruction kernel is generated by using the following steps:

1. Sample an n-D reconstruction kernel;

2. Project each sample to a location in a 2-D image buffer;

3. Center the generic Gaussian footprint (Figure 5.6(a)) to the projected location;

4. Augment the pixel values in the image with corresponding pixel values of the generic Gaussian footprint.

In the second step of bootstrapping, we apply the same splatting algorithm to each data hyper-rectangle using the footprint generated in the first step.

Each change of the view point requires the generation of a new footprint of the reconstruction kernel. Today, GPUs are fast and have much larger memory than a few years ago. Therefore, we can load the generated footprint into the texture memory and blend the splats.
of all data hyper-rectangles to generate the final image. In this way, we take advantage of the power of graphics hardware to reduce processing time by shifting some workload from the CPU to the GPU.

The final image generated is a density map of the data. To convey density information of the data, opacity of a footprint can be determined by the number of data records within a data hyper-rectangle. Specifically, the opacity of each footprint is a function of the weight of data records in a hyper-rectangle

$$\alpha = 1 - \exp(-\mu \cdot \text{density})$$  \hspace{1cm} (5.1)

where $\alpha$ is the opacity of the footprint, $\mu$ is a scale factor, and $\text{density}$ is the weight of data records in the hyper-rectangle. The weight is the aggregated value of data records in the hyper-rectangle. It may be the number of data records or the sum of the values of a particular variable of data records in the hyper-rectangle. The scale factor can be adjusted by the user. It can be changed so that the effect of sparse hyper-rectangles will diminish.
Footprint splatting is integrated with grand-tour to create animated projections to display different aspects of the data. This illusion is created by moving from a source projection to a destination projection along a geodesic path with interpolated transit visualizations. We have observed that creating a new footprint for each transit visualization improves the quality of the visualization slightly with a cost of 40% overhead. Therefore, we choose to use the footprint generated for the source projection during the tour. This is a compromise between the quality of the transit visualization and the rendering speed. Because our goal of applying grand-tour is to depict an overview of large data sets during the animation, we think the rendering speed is more important than the quality of transit visualizations.

Figure 5.7 illustrate the effects of footprint splatting with and without bootstrapping as taken from n23Tool [111], which is the ancestor of mViz. The image in Figure 5.7a is
generated by the following two-step bootstrapping method. It shows roughly the shapes of data hyper-rectangles in the projected space. The image in Figure 5.7(b) is blended with generic 2D Gaussian kernel which has a circular extent and is symmetric in all directions. The gain of bootstrapping is that it reveals information about the orientation of projection of hyper-rectangles. For example, cluster with blue color shows an orientation on one of the axis in (a) while the same cluster in the image (b) is symmetric at all dimensions.

For generating hyper-rectangles from a data set, a binning process is applied. In the process, each variable’s domain range is divided into smaller ranges to create similar smaller hyper-rectangles (bins). Then, each data point is assigned to a bin and data aggregations are accumulated for each hyper-cubical bins. During the process, the user limits are applied to each point whose aggregations are accumulated if selected. There is one drawback of this process that the number of hyper-rectangles created can be too many for interactive visualization. For example, bins with few data points can exist. In rendering pipeline, the center of the bins are projected using grand tour projection suit and splatting is applied by texture mapping the 2D footprints generated for the current projection matrix.

The scatterplot and the footprint splatting coincide and both use projections from the grand tour. When the current result set consists of points, the scatterplot is activated internally. This visualization is most appropriate when data are filtered or narrowed down to a subset without a slew of points. Optionally, volume data can be visualized with the scatterplot, in that case, the center of each hyper-rectangle is projected to 3D screen space. In mVis, detaching the scatterplot and the footprint splatting and executing them in separate visualizer windows are possible for side by side comparison. In such a setting, transform a-
Figure 5.8 Clustered Boston Housing Data [22]

(t_x, t_y, t_z), rotations (R_x, R_y, R_z), data selections, and color variable selection are linked and synchronized among the visualizers through the internal linking mechanism of the mVis framework (see Chapter 6). In Figure 5.8, the Boston Housing data [22], which is clustered by k-means clustering algorithm, visualized by the scatterplot and the footprint splatting. As seen from the figures, even when a small data is visualized, some points overlap in the scatterplot view (left image), while each point contributes to the final image in the footprint splitting (right image). If we closely inspect the images, we can identify this effect below the cluster4 (in magenta and in the top-left corner) where the footprint splatting image reveals existence of points from cluster3 (in blue).

5.4.3 Density-based Parallel Coordinates
Introduction and Related Work

In this section, we introduce the extensions to parallel coordinates in order to render multiresolution data aggregations. Parallel coordinates [54] provide an excellent metaphor to visualize data with high dimensionality by horizontally arranging vertical coordinates,
one for each dimension. A data record is displayed as a polyline that crosses each coordinate at a position corresponding to its value on that dimension. Parallel coordinates expose the following drawbacks and limitations when the data is large in number of records and dimensions. First, the number of dimensions that can be visualized is restricted only by the horizontal resolution of computer display, although too many coordinates may make the visualization difficult to understand. Second, a polyline is by no means an economical way to represent a data record. Third, large number of data records overlap on the screen and create clutters. Also, large number of dimensions causes dimensional clutter. Another disadvantage is that the crossing problem occurs when multiple polylines cross a vertical axis at the same point which are hard to follow. Therefore, parallel coordinates are inefficient in visualizing many individual records.

The parallel coordinate technique has been extensively studied in \cite{57, 58, 34, 93, 48, 109} to address the issues listed. To reduce the amount of clutter imposed by large number of records, interactive parallel coordinates frequency plot \cite{16} highlights clusters in a large data set by building a contingency table between every consecutive pair of parallel coordinates. XmdvTool uses hierarchical density-based parallel coordinates \cite{41} to visually explore hierarchical data clusters produced by an agglomerative data clustering algorithm such as Birch \cite{116}. Each data cluster is visualized as a horizontal zigzag opacity band. Agglomerative data clustering produces a hierarchy of data clusters. The user may specify a cut across the hierarchy. Data clusters along the cut are visualized in parallel coordinates. Agglomerative data clustering also imposes an order on all clusters. This enables proximity-based coloring in XmdvTool which assigns colors to each cluster by mapping
the cluster order into a linear colormap table.

Graham et al. [45] employed quadratical curves instead of straight lines, spreading out points on axes with small discrete values, and focus-context techniques to address the crossing problem. Peng et al. [76] applied dimension reordering to reduce clutters. In [109], a hierarchical organization of dimensions were proposed to lessen effect of dimension cluttering. Johansson et al. [58] applied a linear, square root, and logarithmic transfer functions to high-precision textures to reveal an overview, identify outliers, and emphasize low-density regions, respectively. 3D clustered parallel coordinates was introduced to find space relation of a selected dimension with many others [57]. In this representation, clustering data before visualizing is prohibitive for large data and there will be many combinations of 2 to examine relationships in a high dimensional data.

**Prominent Features**

The same as footprint splatting, input to density-based parallel coordinates is data hyper-rectangles. Compared with irregular shaped data clusters in agglomerative data clustering [41], regions represented by internal nodes of a DA-tree are always hyper-rectangles and can be ideally displayed as horizontal translucent bands. The location of a band across each coordinate reflects the position of the corresponding data hyper-rectangle in that dimension. The extend of a band represents the span of the data hyper-rectangle. The extend is determined by the minimum and maximum values of the corresponding hyper-rectangle.

The opacity of the band is determined by the same function as given in Equation (5.1) and is a function of the density of its corresponding data hyper-rectangle. The middle of
each band is encoded with the deepest opacity. A band fades gradually from the dense middle to fully transparent edges. The color of the band can be assigned to represent value in a dimension, cluster, closeness measure etc. Bands of all hyper-rectangles are blended to produce the final image. The regions with white color or close to high RGB values indicate dense regions.

We extended the traditional parallel coordinates to density-based parallel coordinates to convey the density information from multiresolution data aggregations. As an example, Figure 5.9 is a snapshot of visualization of 'PUMS' data with density-based parallel coordinates (DPC) in mVis, where the fast-access panel area is manually imprinted.

The gray band across the display in Figure 5.9 is called the data selector which serves for two purposes: (1) data selection in screen space for brushing and other operations, (2) range selection for window (range) queries. The center of the selector is highlighted with a darker color so that the user can adjust the selection conveniently. Clicking above the center line on an axis changes the high selection, while clicking below the line affects the low selection. The data selector represents a hyper-rectangle as a query region in the data space. The data selection band can be used to issue a range query to a DA-tree for the purpose of data fetching. This allows the user to select subsets of data in high dimensional space using the 2D screen.

Similar ideas mentioned in the related work can be applied to our DPC to overcome the limitations listed previously. In addressing the crossing problem, different coloring and translucency of bands naturally help to trace individual rectangular polyline in DPC. Additionally, when a selection is made on the screen over one dimension, selected lines
Figure 5.9  Density-based Parallel Coordinates for MDA

will be highlighted with a default color. If more than one line is selected, exact values of a line (active line) in the fetching order is displayed on the screen by tooltips. The user can easily alternate lines to display their textual detail on the screen.

In addressing the clutter problem, displaying a subset of the entire large data sets in the form of MDA from a DA-tree naturally reduces clutters greatly. Also, if the scale factor in Equation (5.1) is decreased, DPC brushes away some of the superfluous hyper-cubes with low density and reduces clutters. We have another mechanism to do the reverse selection to focus on low-density regions as well. Likewise, applying a dynamic extend to lessen
the extends of all bands by a factor cuts down the area mapped by each hyper-rectangular polyline. The extend function \( E_i(b) \) below finds the extend of a band on dimension \( i \) by multiplying the band width with the band multiplier \( b \).

\[
E_i(b) = b \cdot (highValue_i - lowValue_i)
\]

In addition to above clutter reductions, when any subset of current data visualized is selected and highlighted by the data selector, the user can opt to hide (brush) them and focus on analyzing remaining records. Thus, brushing helps reducing clutters on a over crowded screen. This process is not accumulative in applying the process consequently. That is, each brushing is applied to the current selection and each selection includes to the pool of all data points including previously brushed points.

Another feature of DPC is reversing the translucency to dim visible bands and enhance discernibility of barely visible or un-visible ones. When the opacity is applied to the current data set, some are not noticeable because their density is too low to be in human eye visibility range. In some cases, these unnoticeable points might be the ones that the user is interested in. In reversing the visibility, we pipeline \( \alpha = 1 - \alpha \) after application of Equation 5.1.

Last but not least, optionally, our backend database carries out normalization of data to mitigate visualization computation on the client. Various transformations are applied to data from data-space to screen space during the visualization process. Data in data-space will be mapped to screen-space usually in range \([0, 1]\) or \([-1, +1]\) and then into window coordinate system. The data-space to screen-space mapping is a linear mapping from source
\textit{valS} in source domain \([\text{minS}, \text{maxS}]\) into destination \textit{valD} in domain \([\text{minD}, \text{maxD}]\) as in Equation 5.2.

\[
\text{valD} = \text{minD} + \frac{(\text{maxD} - \text{minD})}{(\text{maxS} - \text{minS})} \cdot (\text{valS} - \text{minS}) \quad (5.2)
\]

Most of the variations of parallel coordinates apply a linear mapping to map data points into the same screen-coordinate for each variable. Generally, minimum and maximum of variables (dimensions) in data-space are mapped to the minimum and maximum of each axis, respectively, and length of axis allocated is the same for each dimension. This representation is suitable to analyze the distributions of values, however, it is not intuitive to make relative comparison of dimensions to each other. To reveal more information about variable-to-variable relations, \textit{MVVis} offers six different mapping options as summarized in Table 5.6. Direct, absolute, and relative mappings for data range and query range are identified as possible alternative mappings from the data space into the screen space. While one method discloses better information for a data set, it may not be good as good for another data set. In the table, second column is a descriptive name of the mappings. Third column indicates what minimum and maximum from the source (data-domain) are mapped to the axes.

To see the affect of the mappings, lets give an example of 4 variables whose domain minimum and maximums are depicted in Figure 5.10(a). In the figure, an empty circle and a solid circle indicate screen position to which minimum and maximum of a domain mapped, respectively. Implicitly, all values within domain range are mapped into these two points. We called it a direct mapping if all domains are mapped to the maximum allocated
axis length as in Figure 5.10(b). This is the mapping most of the parallel coordinates uses.

The absolute mapping, is the mapping where the minimum all dimensions maps to the lower end of axis and the maximum of all dimensions maps to the higher end of the axis.

As shown in Figure 5.10(c), values for each domain are mapped to their absolute position. An advantage of this mapping is that correlation between the values of different axis can be visually determined. A disadvantage is that while every pixel on the screen is an important asset, space above and below mapped maximum and minimum is wasted. The relative mapping is a relative mapping of dimensions according to another dimension. As plotted in Figure 5.10(d), we mapped the variable with the largest range (Var2) to whole span of screen space then the others are relatively mapped to the screen. We indicated center alignment (center-mapping) in the figure (dotted horizontal line), that is center of all domains are aligned. However, bottom and top alignments are straightforward to do by shifting center-mapped values up or down in screen space. The mappings for current query (2,4, and 6) is the same as domain mappings except minimum and maximum of dimensions
are lower and upper limits of a query window specified to fetch data.

When large number of points are translated from high dimensional space into screen, mapping contributes to slow-motion display of images and reduced interactions performance. One solution for this prohibitive consumption is to store normalized values in main memory to access directly instead of calculating at every refresh of the display. However, this is a trade-off between speed and memory. Our alternative solution for the problem is normalizing the values before building a kdB-tree. Thus, backend database not only stores multi-dimensional aggregation data, but also tail data for fast rendering.
5.4.4 Record-based Parallel Coordinates

Another visualization integrated to mVis is traditional parallel coordinates to visualize points as polyline across the vertical parallel axes. Our record-based parallel coordinates (RPC) coincides with DPC or acts as a stand-alone visualization technique. In the latter form, there are many attributes and variables shared with DPC through the linking mechanism of the mVis framework including current state of the data selector, zoom value (transformation in z-axis $T_z$), opacity. Figure 5.11 plots 704 points from 1%PUMS data after executing the query below. The data selector which generated the query from the
previous view is also intact. Although, it is not shown because of the cropping, the fast-access pain of RPC and DPC contain similar interface except the extend scaling which is unusable in RPC. As seen on the image, tooltips convey on-screen information as the pointing device moves around. This feature is particularly necessary for non-numeric variables i.e. textual ordinal and nominal variables, because non-numeric values are replaced with numeric indexes during construction of a B+ tree on the server side. Therefore, metadata carries mapping tables to the client and the client presents more meaningful textual data to the user instead of indexes. For example, the state with STATE code 22 is the state of Michigan as depicted in the image. Moreover, this numeric value is substituted automatically when the query is composed. In this view, the variable of STATE is assigned as the color variable and it is cyan for MICHIGAN.

```
SELECT TOP 10000 * FROM theTable
LIMIT POINT
WHERE LIMIT IS 0, STATE=22, YRBUILT BETWEEN 5 AND 9,
  ELEC BETWEEN 0 AND 2300, GAS BETWEEN 0 AND 1700,
  WATER BETWEEN 0 AND 1200, OCC BETWEEN 0 AND 1500,
  BEDRMS BETWEEN 2 AND 5, PERSONS BETWEEN 1 AND 6,
  EMPSTAT BETWEEN 1 AND 8
```

5.4.5 Scatterplot Matrices

We extended scatterplot matrices [8] to display multidimensional data aggregations similar to the implementation in XmdvTool [101]. This visualization reveals inter-dimensional relations for each possible pair. However, it deteriorates interactive visual analysis when the number of dimensions or the number of points is large, because each point is drawn as many as the number of cells in the matrix ($d^{ndimension}$). The density-based scatterplot matrix draws a rectangle (or a square for equilateral di-
dimensions) for each hyper-rectangle from a multiresolution data aggregation tree. The area consumed depends on the lower and higher bounds of the hyper-rectangle on each dimension. Each density-based input is assigned an opacity based on its density and each of the translucent rectangle is blended to create the final image. When point data is visualized, each point data can consume as small as a pixel for each cell of the grid.

Similar in function to the data selector of the density-based parallel coordinates, a rectangular data selector coexists within each grid cell to specify upper and lower limits for the crossing dimensions. The data selector is linked to a global range query array which is eventually used by a query fetcher. Thus, this visualizer shares the data selection attribute.
5.4.6 User Interaction

Exploration and analysis of large relational data require interaction with the data. When the size of data increases, real time user interaction becomes more and more challenging.

Kosara and Hauser [63] presented a survey of interactions in information visualization in general. Mehta et al. in [74] and thereof introduced brushing for multivariate data sets. We can display data and generate attractive images, however, lack of interactivity greatly reduces ability to analyze and extract information. Thus, complying and integrating interaction methods listed in aforementioned papers is crucial in large data visualization.

Brushing is an operation to select subsets of data for further operations like highlighting, masking, and moving. On-screen data selection does not work well with projection-oriented visualization techniques such as scatter plots and footprint splatting. For example, a hyper-rectangular brush specified in the screen space has only up to three dimensions. We need to define a high-dimensional brush in order to select a subset of the data in high dimensional space. Parallel coordinates offer a convenient way of specifying high dimensional brushes and selecting high-dimensional data. Thus, the selection of the data selector of the parallel coordinates in mVis is linked to other visualization techniques in which specifying a query window for high dimensional data space is either difficult or impossible.

Multi-window support is important to explore and compare side by side different subsets and resolutions of large data sets. In addition, different visualizations can be linked together so that information hidden in one view might be revealed in another. Linking is a
common way of sharing the user actions among the multi-view and multi-technique visualization systems. mVis framework is built on linking and sharing to make the visualization linking intuitive for users and developers.

For more convenient access to functionalities of visualizers, mVis framework provides a fast-access panel where each visualization technique can customize. The fast-access panel for the density-based parallel coordinates is marked on Figure 5.9. In the example, the following widgets are added: buttons for opening various GUIs, a spinner for changing opacity of the bands, a slider for extend scaling, a checkbox for brushing selected bands away, and a checkbox for reversing visibility of the bands.

mVis carries out various other operations for interactive exploration of large data sets.

Below is the list of some other interaction features:

- In the footprint splatting rendering visualizer, viewpoint can be changed by using mouse click and drag. Drag with the left mouse button pressed moves the scene. Middle button is used for zoom in/out, and right button for rotating the scene. All these operations are linked to the Scatterplot visualizer and available for shearing with new visualizers.

- The opacity of footprints and translucent bands can be adjusted to show sparse regions may fade away and dense regions show up.

- When applicable, extend sliders are integrated to change span of dimensions to reduce clutters.

- A grand-tour interface provides basic animation operations (start, rewind, pause etc.)
to control the selected tour which shows different projections of the data. Also, projection matrix at each step is available as a shared linkable object for other visualizers.

- Visualizers are linked so that same or different data can be visualized from different perspectives.
- Client-side selection on the screen-space is linked among visualizers so each visualizer identifies selected points.
- A subset of data can be selected by using the data selection band in the parallel coordinate display, then filter/hide the selected data to analyze the rest.
- The parallel coordinate display has a 2D display brush in addition to the selection band. Data points under the brush will be highlighted.
- The user can organize the order coordinates of the parallel coordinates to reduce intensity of clutters between adjacent axes.
- The order of variables can be specified when a range query is composed to push dimension reordering to the server side.
- History of range queries is preserved for back-tracking and navigation.
- Each visualization is augmented with a status bar, descriptive labels and numbers, or screen tips when the user moves the cursor over the visualization window
• Existing menu items on the visualizer manager window are accessible by both built-in visualizers and loaded dynamically later.

• The user can change color assignments, load and save colormaps for each data sets through the provided interface.

• Color variable and mechanism for point data and volume data are separated.

• A query manager and data fetcher are accessible from all visualizers for specifying dynamic queries.

5.4.7 Colormap Management

![Colormap GUI](image)

**Figure 5.13** Colormap GUI

Different color index tables are available to the user while user-defined color maps can be loaded as well. Figure 5.13 is cropped snapshot of the graphical interface. Existing
colormaps are generated with different color mechanisms: grey, red, green, blue scales; Hue-Saturation-Value (HSV) and Hue-Saturation-Lightness (HSL) [105]; and cyclically shifting HSL and converting to RGB. Each colormap constitutes of 256 color indexes. This number of distinct color is good enough for distinguishing close colors. A loaded colormap can mapped to current color variable as direct or scattered manner. In the direct assignment, first color is assigned to the first value of the variable, second color is assigned to the second value of the variable and so on. On the other hand, the scattered indexing assigns the first color to the first value of the variable, the last color to the last value of the variable, and colors at a calculated index distance are assigned to the other value.

5.4.8 Color Variable

![Figure 5.14 Color Variable Selection GUI](image-url)
Coloring visualization elements improves the analysis of final images incredibly. There are two questions to ask: (1) What is the color variable? (2) How to index the values?

There are many ways of color variable assignment. Most widely used one is to choose one of the variables as the color variable. Another preferred method is clustering data and using the cluster information as the color variable. We opted for the former method for point resultsets. Latter method needs clustering information stored at the database end, thus clustering large relational data organized into a high-dimensional tree index needs to be studied. Other method which is based on the closeness of the elements is ‘proximity-based coloring’ which is used in XmdVTool [101]. A problem is that it is not straightforward task to identify closeness or neighborhood of an internal node in a high-dimensional tree index. Consequently, our DA-tree doesn’t support this coloring at this time. However, it might be doable by using a multi-way neighborhood linking in the expense of loosing some disk page space.

For hyper-rectangle aggregation data, we used data aggregations as color variable in mVis. Number of points, average, or density of a region and low or high values of a dimension are some alternatives.

In addressing the second question, there various alternatives to map the color variable to a colormap table.

- **Direct Color Indexing:** Values of the color variable are within the colormap index domain i.e. [0, ColorMapSize] where ColorMapSize=256 or can be mapped to the index domain by subtracting domainMin or a name-value-label (VCL) pairs are de-
fined in the "meta" file. Each value directly indexes a colormap entry. This indexing is most suitable for nominal, ordinal, and discrete variables.

- **Order Indexing:** Indexes of values of the color variable are used. The order of the values is either the insertion order of each unique value during building the tree, or order of listing in the "meta" file. If the color variable has NVL pairs defined, the index of the value for each record of the result set will be the index to the colormap table. If there is no NVL and "Find unique values" option is selected, then the result set will be scanned ones to identify unique values and their index will be color index to the colormap. This mapping is applicable to ordinal and nominal variables. It can also index discrete variables if its range is less or equal to the size of a colormap table.

- **Linear Mapping:** Values of the color variable in domain \([domain_{\text{min}}, domain_{\text{max}}]\) will be mapped to the colormap index domain. Since continues variables and discrete variables with a large range of values cannot mapped one-to-one to colormap table, the domain of values are divided into range of values and each range is mapped to the color table.

If the direct or the order indexing is selected but neither is applicable to map to the color table, then the linear mapping is applied.
5.4.9 Other GUIs

In interacting and fetching data from a data aggregation tree, mili's framework provides dynamic query and data access tools. Figure 5.15(a) is the data fetcher to make up a query, set format of a result set, and select the resolution type. The data fetcher sends an SQL.
linking object with the visualizers. When the visualizers change its content i.e. through the 
data selector in RPC and DPC, changes are reflected to the data fetcher to be fetched by the 
user. Additionally, the fetcher keeps the history of executed SQL statements for re-fetching 
if needed. The Query builder (Figure 5.15(b)) presents an interface to the user to manually 
develop an SQL statement and updates the SQL object. The good thing about this view is 
that the user only deals with the names of values instead of the stored values.

As we mentioned before, the 3D footprint splatting and the Scatterplot implementation 
are integrated with the grand tour [28]. Figure 5.15(e) is the interface to manage the tour. 
Simple, cluster guided, random, and manual projection types are supported to create a 
movie of tours in 3D space.

The Link Manager is the management tool to arrange, link, unlink, and modify sharing 
of objects. As we discussed its detail in Chapter 6, event-driven framework utilizes a linking 
mechanism for inter-component communication, linking, and sharing of objects and events.

5.5 Conclusion

Representation of multiresolution data aggregation in the internal nodes of a high di-

dimensional indexing is a new representation for large data and augments additional aggrega-
tion information about the data. This pre-calculated information improves interactive 
visual analysis of large data, which otherwise would preclude visualization.

We described the internal detail of building and managing MGA with our backend 
database system in addition to exposing components and their roles in KDBMS. We en-
courage other researchers in the visualization and related fields to use KDBMS as their
We also showed extension of three existing techniques to render aggregated data. mVIs houses the extended visualizers and harmonizes them by highly coupling them. Various interactions are implemented and integrated into mVIs; nevertheless, more comprehensive mechanisms are needed for generating vivid outputs which present analytical information with ease.
CHAPTER 6

VISUALIZATION FRAMEWORK: mV is

6.1 Introduction

Major components of visualization frameworks, as mentioned in various taxonomies, are generally: abstract data, visualizers, interaction operators, views, the user, and the hardware. How all these modules or components are coupled or interacted are model specific. And each framework addresses some but not all of the issues. As a result of the design variations, the frameworks differ from each other based on which and how the components are integrated. The difference is also reflected to the extensibility and scalability of the frameworks. There are frameworks as ready-to-use packages, reusable code and modules, plug-in environments, or hybrid of later two. One common feature is that all of them embed different numbers of visualization techniques.

mV is is a visualization framework that follows the dataflow paradigm [24] as many other counterparts and descendent of n23Tool [111, 112]. In the design of our framework, we prioritize the simplicity and flexibility of both the development of visualizer techniques and the integration of external modules, libraries, algorithms, and other components. Our intention is to reduce the time spend in programming and integration while pointing out the need for compatibility and standardization of new components introduced to the visualization. The framework introduces a simple, event-driven link manager to link and share objects between the components of the architecture. Some other features of the toolbox are grand tour [11] functions, an interface for dynamic queries [1], and an easy-to-use
customized GUI development toolbox.

6.2 Related Work

There are various visualization frameworks with different taxonomies used. Shneiderman [87] presents seven data types and seven user tasks (task-by-data taxonomy) that the designers/developers of advanced graphical user interfaces should consider. The process of building a visualization has been discussed in a variety of ways by the research community [78, 91, 62, 87]. Also there are different issues in designing visualization architectures as discussed by Tang et al. [91].

Each visualizer framework identifies the system components and defines inter-component relations differently. For years, research in the field have focused on user interactions, data integration and transformation, view linking, attribute mapping, and integration of tasks and algorithms (sorting, classification, filtering) to built systems. Cui et al. [53] introduced reusable view and value operators (user interactions) to understand the interaction between data and view. They proposed a state model to identify and classify interactive and non-interactive operators. In [64], Kreuseler et al. proposed preprocessing pipeline to identify the structures from unstructured data using information objects which are grouped by a similarity measure and presented by self-organizing maps and dynamic hierarchy manipulations.

In re-View toolkit [113], the authors introduced two new interactions techniques to address the “Extract and Relates Tasks” of the mentioned taxonomy [87]: bookmarking and coordinated filtering. In their 3-tier architecture, a view coordinator and a meta-data repres-
itory interchange information about the bookmarked view (dataset and view parameters). XmdvTool [101, 74] and Spotfire [4] are ready-to-use, tightly linked, and interaction- and brushing-reach software. XmdvTool provides density-based visualizers based on cluster hierarchies, brushing and filtering. It is not extensible but open-source code.

Polaris of Stolte et al. [88], a data exploration system for large multidimensional relational databases, is based on Pivot Table interface and constructs visual specifications to describe table-based graphical views and generate relational queries. It is worth mentioning that its design trivially supports undo-redo operations. The InfoVis toolkit [37] is composed of existing visualizer implementations with variety of interactions: range-sliders, control panels, fisheye lenses, dynamic deformation techniques, dynamic labeling, and dynamic queries. It is for structured data and supports graphs, trees, and tables. VTK [84] is a comprehensive toolkit for 3D graphics based on the dataflow model [24]. It contains an interpreted application layer which complies with the Tang’s sixth issue [91]. Prefuse [50] provides set of reusable components to design/built dynamic 2D visualizations for data with discrete structures. The library contains layout algorithms, navigation and integration techniques, integrated search. A scalable filtering mechanism maps abstract data to visualizable units while actions manipulate data and set visual properties. Event logging and animation actions are features embedded.

There are other toolkits which are domain specific [83, 65, 85, 13, 69] and present nice features while addressing different issues. GUESS [2] is a graph exploration suit that provides a domain-specific embedded language and interactive interpreter for textual interactions.
These frameworks and others referenced in aforementioned papers present good abstractions to data and GUI interfaces with encapsulated code, and reusable programming units. Ideally, the visualization community, can generate and integrate analytical tools and libraries to fulfill the requirements and issues of the visualization and data analysis tasks by harnessing existing tools, code, and libraries. So, setting standards for compatibility to an extent is mandatory for developing effective visualization and analysis tools.

6.3 Components of mVis

mVis is a framework which is based on the dataflow model and is extensible by plugins. We identify the following issues in development of visualization techniques and frameworks in general:

1. Interface incompatibility of data, visualizers, and other components

2. Redundant code development

3. Irreconcilability because of the programming environments

4. Long learning and adaptation curve for developers.

These issues are closely related to each other that the visualization community should address them by setting meticulous standards and abstracts for interfaces, inter-operability, and reusability of code among different environments. We have centered these issues into our framework particularly to simplify integration of new interfaces to different data sources and development of visualization techniques. mVis has similarities to the frameworks described in [23, 113]. We have also focused on reducing the time spent in program-
ming while providing a meta-data in support of abstract data interface. Figure 6.1 shows the building blocks of the framework.

![Diagram of mVis Framework Components]

**Figure 6.1** mVis Framework Components

*mVis* is an object oriented software written in C++. OpenGL and Gtk+ libraries are base libraries for rendering and GUI development, respectively. The components of the framework are data accessible through data manager and drivers; linker objects, loadable modules (data drivers, visualizers, clustering drivers), visualizers, and interactions. To create reusable and dynamic components, we applied different design patterns to build the framework (see [104] for detail and further references). Some of the design patterns are factory method for the library manager to load and access to dynamic libraries; singleton model for clustering and data drivers to create a single instance of each; facade pattern for the visualizer manager to activate, refresh, and switch the visualizers; observer, which
is a behavioral pattern, for the linker manager's linker objects. There is a correspondence between our design patterns and some of patterns mentioned by Heer et al [49] such as the Reference Model pattern, the Renderer pattern, and the Dynamic Query Binding pattern.

6.3.1 Library Manager

This component loads and unloads external library files dynamically in run-time. It acts as a gateway between libraries and the other components of the system. A library is a module for a specific task that is derived from a C++ base class provided as a template. When the library manager loads a library, the loading library registers an entry point to a pre-defined factory to create an instance of the class derived. In mVis, there are various factories to host task-oriented libraries. In the current version of the framework, factories for data drivers, visualizers, clustering algorithms, and generic libraries are allocated. All factories are maintained and managed by the manager. It creates single or multiple objects of the registered classes based on settings of the manager. For example, only one instance of clustering algorithms is created while multiple copies of a visualizer class might be created. Components of the framework communicate with the manager to seize an instance of loaded libraries or a list of loaded libraries of a factory to make them available to the user.

6.3.2 Data Drivers

A data driver is one of the loadable components of mVis. Visualization systems are tightly coupled with data and database systems. In mVis, data drivers liaison between data sources and visualizer systems. mVis employs the Reference Model pattern of [49].
Two problems are prominent related to data—visualization tools. First, many visualization systems embed data access code directly into the visualization code. Thus, most of the time, modifications in data source require a recompilation of the framework. Second, since many of the visualization systems use their custom flat file formats or structures, there is no standard data format for the visualization systems. To address the decoupling of data and visualizer and these issues, we introduce the data driver level between visualizers and data sources. This layer is not to replace existing drivers for well-known database systems (JDBC, ODBC etc.). Nevertheless, our driver at this layer encompasses member functions and attributes for visualization purposes in addition to data access objects (connection, statement, recordset, metadata). For accessing to a legacy data source, implementation of a driver still uses commands from the specific drivers. By the introduction of the data driver interface, Vis also expands compatibility of different formats by applying the adapter design pattern. Thus, with our abstraction, a one-to-one relation is easily extended to a many-to-many relation. That is, a visualizer visualizes different file formats without compiling the working environment and special file formats can be harmonized by different visualization techniques.

Another advantage of our data interface is that it hosts class members for visualizer systems. For example, each driver lists what kind of data it fetches (i.e. volume, point, spatial data, image, graph) and presentation type (i.e. flat, tree, hierarchical). Thus, visualization techniques are enabled or disabled automatically for the input data. Also, the embedded metadata class standardizes collected information from the source. The metadata contains descriptive data for effective and explanatory visual representation, particularly for nominal
and ordinal variables. Since every database management systems stores meta data differently, detail of fetching of meta data is hidden from the rest of the framework within the drivers.

Furthermore, prefetching, buffering, or multi-thread data access operations are isolated from the visualization. This leads to a modular and manageable program development and smooth expansion of the system.

```c
mvis_statement* st;
mvis_resultSet* rst;
mvis_query q;
conn = Librarian->getConncetion("kdb.localhost:5000", ",", ");
st = conn->getStatement();
q.SQL = "SELECT FLD1, FLD2, FLD3 FROM theServer";
rst = st->executeQuery(q);
rst->first();
while( !rst->isEOF()){  
cout << rst->getValue(0) << endl;
rst->next();
}
rst->close();
st->close();
conn->close();
```

**Figure 6.2** Example SQL Code: Using kdb-driver

Data drivers for the mVis framework extends mvis_dataDriver class. Each driver specifies internally a driver identifier (DI). To create a connection via a driver, the library manager loads all the drivers and passes the user’s connection information to each driver until a suitable one is located. If a driver matching the given DI is found, a connection object is returned to the client.

To fetch data, a query statement whose syntax discussed in Section 5.3.4 is passed to
data driver. A base class `mvis_query` encompasses query parameters in addition to the SQL statement. Developers can customize the structure for more complicated query needs. `mvis_query` also passes output format of results (see Section 5.3.2) to the driver.

We have developed a data driver (kdb-driver) to access to KDBMS for visualization of multiresolution data aggregations. kdb-driver opens and closes connections, fetches data optionally by a single or multiple threads, and provides data as point, volume, and mixture of both. Multi-threaded data fetching might be necessary especially for large data. While the driver fetches data, fetched data can be visualized on the screen incrementally and the user interface is not freeze during that process. The visualization community can use the multiresolution data aggregation and extend our data interface by using this driver. The driver registers its entry point to the driver manager. To activate the driver, either the user selects it from a list or the library manager activates it based on the connection information. The code in Figure 6.2 shows the usage of the driver. Initially a connection from a kdb-driver is requested from the library manager with `kdb.<hostName>;<portNumber>` where `kdb` is the DI and connects to a KDBMS running at the specified host and port. No login credentials are passed in this example. Then a query is executed through the statement object of the current connection. After that, the first column of the record set is printed out.

Also, we have implemented a special driver (mdaFileDriver) to generate multiresolution data aggregations from disk files. By using a binning process, fixed size hyper-rectangles are created. However, creation of hierarchical organization for more detail requires the binning process to be repeated recursively. So, each query execution for a different level has to scan the file for the binning. Thus, this driver is suitable for small size files.
The DI for this driver is mda and can be specified as in mda\filePath\.

6.3.3 Data Manager

The data manager component pertains to data access, management, and processing. It is a gateway between the system and the loaded drivers. It is the interface to data manipulation tasks, as well. Some of its other duties are collection of login information, establishing and closing connections, fetching data, providing dynamic query interfaces, and checking SQL syntax. Although, it is not implemented, this is an alternative upper level place to apply a data pre-fetching and buffering. Another location for the same process would be data drivers.

6.3.4 Clustering and Cluster Manager

Another loadable modules are clustering libraries that are managed by the clustering manager. Most of the time, visualization systems apply built-in clustering algorithms as one of the common data operations. And often times, it is stated that “the technique under consideration works equally well with other clustering algorithms”. However, inclusion of new clustering algorithms requires understanding the code and recompilation if the source code is available. Keeping the code compatibility and standardization in mind, we have integrated clustering libraries to our framework as loadable modules. New clusters can be derived from mvis.clusterBase class provided. The library manager loads clustering libraries and returns a clustering object with getCluster(...) member function. Currently, mVis comes with k-means clustering and development of new methods are anticipated. The clustering manager communicates clustering services to the visualizer manager. It
provides a user interface with generic parameter settings to pass to clustering algorithms. All loaded clustering algorithms are available for selection to the user through the manager.

6.3.5 Visualizers and Renderers

A **renderer** is the running engine to draw, paint, and execute elements specified in a supported language. In other words, it is renderers' job to processing a model into images. A **visualizer** contains detail of a visualization technique and models. mVis framework practices the **Renderer design pattern** [49] to separate visualizations from their renderers. Each visualizer specifies what kind of renderer it needs. It uses the language that that renderer understands. An advantage of this pattern is that each component can be developed independent of each other and miscellaneous renderers can be fit into the same framework.

Support of multi-rendering engines is not a common practice in visualization frameworks yet. The initial version of mVis only supports OpenGL rendering by using *gluten* widget. However, we envisage to port new renderers such as a renderer bound to graphviz library [43] to augment drawing graphs when the data is small or large data is filtered to a manageable subset. In such a separation, a visualizer developer (professional or novice) writes the code using a graph, a node, an edge, etc. and the renderer provides the environment to display it (initialization, layout, etc.). Thus, mVis can span to visualization of other domains i.e. software engineering, network design, graph analysis.

All visualizers in the existing version of mVis are derived from `mvis_visualizer_base` class. It contains interface for setting references to data, clustering and color management, grand tour, application parameters, and linker. This class contains template member func-
tions for which the user can augment drawing primitives to run at specific renderer events (i.e. initialization, refresh, motion, zoom in and out, activate, de-activate) and grand tour’s tour events (tour start, tour step, tour end). Also, utility functionalities for projection, coordinate mapping, text display, tool tips, context sensitive menus, image etc. are available in facilitate and expedite visualization development. Derived visualizers set the kind of renderer the contained model is for in order to augment coexisting of multi-renderers in the same framework. Thus, the visualizer manager creates appropriate renderer widget before initializing a visualizer. Finally, the developed visualizer technique has to set types of data (point, volume, graph, etc.) supported to help the visualizer manager in enabling/disabling the visualization automatically as the connected data source changes.

6.3.6 Generic Libraries

MVis can extend its dynamic structure to pre- and post-data-processing steps by augmenting applicable generic algorithms to the system. mvis_libraryBase class is provided as a template to build new building blocks. New generic libraries have to overwrite the abstract (virtual) member function of execute(...). The library manager loads generic libraries into generic factory of algorithms and passes an object handle to applications with function mvis_libraryBase* getAlgorithm (<name>, ...).

6.3.7 Visualizer Manager

The visualizer manager is one of the major components of the system. It coordinates the other components and glues them together. Other managers such as the library manager, the color manager, the grand tour manager, clustering manager, linker manager, the data
manager etc. are all in interactions with each other. Some of its roles in keeping the system intact are the following. Provides the user access to other interfaces through main menu and context sensitive menus. It activates the managers for user interactions and reflects the changes to the system. It is the controller of the Model-View-Controller mode [104] to regulate and synchronize the renderers and the visualizers. The manager loads visualizers and their renderers. It also activates and deactivates visualizers in the system based on the data source type. For example, if a connection is established to a data source providing only volume data, with a fix synthetic structure, then it is the visualizer manager to deactivate the visualizers not supporting volume visualization. Most of actions from the system-wide graphical interaction interfaces are conveyed to the visualizers through the manager. Additionally, it provides means to apply generic algorithms pertaining to data post-processing before visualizing data.

6.3.8 Linker Manager

The linker manager is the controller of the inter-component communication in mVis. This component is not shown in Figure 6.1 because it is fully coupled with not all but most of the components as in Figure 6.3. The linker manager provides a mechanism to share and coordinate the components of the architecture. The mechanism to link multi-views is simpler than coordination model defined in [23]. The design patterns of its internal structure are the The Observer and Intercom [104]. Each component which participate to the linking extends the base class mvis_linker and called a container.

There are three main operations: registering a linker, joining to a linker group, and
Figure 6.3  Linking

synchronizing a group. Two types of linkers can be created: objects and events. An object linker is an intrinsic (int, long, float, double, string etc.) or user-defined type variable whose content will be modified dynamically as needed. To create a user-defined object linker, the user's class needs to inherit `mvis_linkingVariable` class and implement `void copy(const void* pRHS)` member function where the pRHS is the right hand side of source variable of the same type. The other linker type, event linker, is a notifier linker like a callback for changes or actions. A linking container with event linkers catches the updates for a linker by implementing abstract member function `void onLinkingVarChange(string pLinkerName)`.

Details of the linking process steps are below.

- An object or an event linker is registered by a component (a container) such as a visualizer, interface object, other loaded modules etc. This is process is called linking.
During the linking, the container specifies the type (object or event), status (readable, writable, or both), direction (ability to update other linkers or not), and visibility of the linker to the user at runtime for manual linking/unlinking.

- After linking, the linker will be subscribed or joined into one or more linker groups. Linkers and linker groups are identified by a name. If a registered linker tries to join a non-existing group, then a new group will be created automatically by the linker manager. A linker group can be private to only a few containers by making it invisible to the rest of the system. Only the containers who know the name of the linker group can participate to that group. Thus, components (such as visualizers) implemented by different developers can link their linkers silently. When the last linker is removed from a group, the group will be deleted by the manager.

- For updating or signaling linkers in a linker group, the container synchronizes whenever it modifies its linker. Then, the link manager updates content for object linkers and executes the callback for event linkers based on the restrictions and properties of the linkers and the groups under consideration.

The linking structure described has some advantages. First, its usage and implementation are straightforward with the aforementioned base classes. Second, it virtually shares any objects and variables without consuming memory resources. Third, integration of runtime loaded components through linking is effortless because of referencing linkers and linker groups by name. The link manager displays all of the linkers and linking groups created by the existing and previously loaded components through its GUI. Developers of
new loadable modules may join their linkers to the existing linker groups simply by using names of the groups. For example, a new visualization technique can share rotation angles \((R_x, R_y, R_z)\) from the existing scatter plot visualization in ways as well as it can register to the event linker of new data fetches in order to supply special data pre-processing. Another advantage is that through the linker manager, default visible linkers can be re-grouped, un-linked, or re-linked manually in run-time as needed. Such dynamic management capabilities are especially useful when the number of components is large i.e. many visualizers. Finally, the linking pattern helps particularly visualization techniques to share their new futures and attributes with other techniques to synchronize data visualization.

A drawback of the linking system is that it is backward compatible only. Components developed and integrated to the system cannot link to the linkers introduced later by other components. To use the new linkers, previous components' code needs to be updated tediously. Nevertheless, this limitation leaves out third party components incorporated to the system without available source code in updating and extending new features by linking.

6.3.9 Interactions

Look and feel metaphor is represented as the interaction module in mViDa framework. It is inclusive of graphical user interfaces, user-computer or user-visualization message exchange, input from the user and the response from the system, or events and actions of the participating components of the system. It provides data manipulation interfaces such as dynamic query interface with a built-in query builder, a navigator for SQL statements.

User interactions are important part of a visualization environment [63, 95]. Our ex-
tensions to visualization techniques are associated with effective interaction tools so that users can navigate within the large data set. Using zoom-in and zoom-out, a user can drill-through a visualization of data from a low resolution overview to the most detailed level where individual data points are displayed. Other interaction methods include linking, brushing, panning, data selection, dimension reordering, and animation of data projections by grand-tour [11]. In mVis framework, these interactions are easy to embed into developed components or are already presented to link from new components.

To speed access to major functionalities of a visualizer, the library manager allocates a space called fast-access-panel to locate an interface to snap in. A visualizer developer can easily implement a panel for his/her visualizer using the toolbox integrated to the system (Section 6.4). The visualizer manager affixes the panel of active visualizer as visualizers are switched. The main menu provided by the manager is accessible by the new components through an available event linker, which is registered by the manager. Thus, this is another way of fast access to provided interactions. As expected, keyboard short-cuts are in-place for common user actions.

To augment user-visualizer interactions and improve understandability and interpretation of visual representation, tooltips for the visual elements (widgets), descriptive status bar messages, pop-up menus for visualizers and individual widgets are provided. Particularly for visualizers, zoom-in and zoom-out through scroll wheel of a pointing device are integrated. Coding reaction of a visualizer to such events needs overriding a member function of the visualizer base class. Each visualizer produce rich interaction methods using various elements from the library such as a slider for adjusting transparency of visual
elements, mouse click-and-drag operations for data selection.

The linking mechanism is used in communication and propagation of interaction elements. For example, an on-screen selection for filtering or brushing of a view sets list of selected records, then other visualizers treat the selected records distinctively.

Other components and managers of the framework also provide GUI’s to the framework such as color management, grand tour management. Also, it is worth to mention that callbacks from widgets blended to system components are gathered to a member function (gui.Event(...)) to reduce wearisome function declarations and other wrappings in programming.

6.4 Toolbox Over Gtk+

For implementing GUI of mVis, we used Gtk+ toolkit. Since this is an event-driven library written in C, using it in C++ development needs wrappers. We built a higher level toolbox that contains object oriented widgets based on Gtk+. The toolbox contains only a subset of the Gtk+ widgets, including container, label, button (image and text), single and multi-line textbox, checkbox, listbox, GL renderer, slider objects. The base class for these objects is mvis_guiObject as depicted in the inheritance diagram of C.3. For implementing GUI’s using the objects from the toolbox, a programmer writes less number of lines of code compared to lines needed when using Gtk+ library directly. The syntax and event handling process are simplified so that even a novice developer can learn and use it in a short time. When objects are created an object reference is returned, however objects can be access by name or object id too. A facade design pattern is applied. The topmost object is the
container object. After creating a container, the programmer adds all other objects onto it.

To create a window, a base class (C.7) is available which inherits from the container class. For event handling, overriding guiEvent(...) member function of mVis guiEvents class is sufficient to code. Local variables of a container object can be attached to toolbox objects.

**In this way, no other calls is necessary to exchange values from/to a widget.**

Popup-menus, mouse, and keyboard events can be easily associated with the objects with a few lines of code. Otherwise, using Gtk+ library to do the same functionality would require quite a bit work. Thus, in GUI production, the simplicity and the efficiency are achieved in terms of the number of codes.

We demonstrate the simplicity of usage of the toolbox and the visualization template class in the code shown in Figure 6.4. To keep the code short, only two methods of the visualizer mVis are implemented (init() and guiEvent(...)) and one button is placed onto the access-panel. When the button (“Say Hello”) is pressed, it opens the window of helloWorld class which contains a label (“hello-world”) and a close button. When the close button is clicked, the class’s event handler (guiEvent(...)) executes to terminate the window.

**In conclusion, a visualizer developer can easily attach GUI’s to his/her visualizer to accomplish various tasks i.e. setting options, parameters or information collection.**
class helloWorld: public mvis_gtkWindow{
    helloWorld()
    this->addLabel("lblHello", "Hello World !!!");
    this->addButton("cmdClose", "Close");
    this->createGtkWindow();
}

bool guiEvent(string pObjName, int pEvents,
              int pValue, void* pUserData)
{    
    if(pObjName == "cmdClose")
    this->quitWindow();
    return true;
}

void myVis::init(){
    _container->addButton("btnHello", "Say Hello");
}

bool myVis::guiEvent(string pName, int pEvent,
                      int pVal, void* pUserData)
{    
    if(pName == "btnHello"){
        helloWorld hello;
        hello.open();
    }
    return true;
}

Figure 6.4  Hello World: GUI in mVis
CHAPTER 7

RESULTS

7.1 Introduction

In this chapter, we disseminate our empirical results conducted. We will categorize our findings for the server and the client of the visualization system separately.

7.2 KDBMS Performance Data

As we mentioned before, we organize the multiresolution data aggregations as a hierarchy of multiple resolution density regions into internal nodes of the DA tree. We have used a synthetic data set to test the performance of data aggregation tree for data insertion. The synthetic data set contains 12 continuous variables. Values of each variable are recorded as double float points and are randomly generated with uniform distribution within a range of values from 0.0 to 1.0. Block size of the DA-tree is 16K bytes (4 physical disk page). Fan-out factor of non-leaf blocks is 149. Fan-out factor of leaf blocks is 163. Up to 300 million data records (that is, more than 28G bytes of raw relational data) are inserted into the DA-tree. After insertion, the file containing the leaf nodes consumes about 42.5G bytes disk space. The space utilization reported for the DA-tree is about 69%.

Currently, KDBMS only builds DA-trees by insertion only. It is a long and tedious process. Figure 7.1 shows the timing in number of days. The main reason for such a big insertion time is the file organization on the disk. TPIE [9] library maintains two index files on the disk for internal nodes and leaf nodes. The size of leaf blocks file is quite big.
a large data. Also, considering the fact that the minimum occupancy of leaf nodes is not 100%, the index file is going to be larger than the original source file. TPIE allocates each new block always at the end of the file, therefore, one of the splits of a node goes always far from the splitting node. Since, splitting of nodes is unordered, when children of a node are accessed, the head of the disk scans the disk file back and forth as many as the number of split pairs residing in this node. The head movements over a single large file, i.e. 30G bytes, is very I/O inefficient and reduces the performance greatly.

We are currently working on improving this deficiency in file organization of the library. For reducing insertion time, we maintain blocks in smaller disk files to reduce the span of disk head movements. For increasing query performance further, we re-organize the tree such that blocks for sibling nodes reside close to each other in the file. We are considering applying different techniques for nodes organization. For example, depth-first and
breast-first organizations of internal nodes are two immediate solutions. However, more complicated organizations can be applied based on the statistical analysis of access patterns collected from the execution of queries. Since a re-organization is usually performed after building the tree and during database maintenance, its effects is not reflected to insertion performance. Consequently, an intermediate organization during building the tree could help to reduce the building time.

Table 7.1 shows how the tree size grows as new data records are inserted into the tree. Figure 7.2(a) illustrates the increase in the number of leaf and non-leaf pages as the size of the tree grows. As seen in the figure, the growth rate for both block types is similar whereas the number of leaf pages is much more than the number of non-leaf pages. The increment after insertion of $150 \times 10^6$ records is steady because the maximum height of DA-tree is 4. To observe a considerable change similar to the one shown for records between $25 \times 10^6$ and $150 \times 10^6$, the height needs to change which requires many records to be inserted.

Figure 7.2(b) shows the average number of block I/Os to insert a new data record into the DA-tree as the size of the tree grows. The number of I/Os includes both the I/Os
to insert the data record and the I/Os to update data aggregation information in non-leaf blocks. In the figure, we draw sum of reads and writes for leaf and non-leaf nodes, because the numbers of reads and writes are similar. Due to the caching, only a small difference observed.

Data aggregation tree allows two types of user interactions: (1) browsing of data aggregation information at a given resolution, (2) range query. A query issued by data exploration tool is usually a combination of these two types of user interactions. Browsing of data aggregation information at a given resolution is supported by visiting internal blocks of the data aggregation tree. There are multiple ways to define resolution. The simplest way is probably by the depth of the data aggregation tree. A data aggregation tree is conceptually a binary kd-tree where each non-leaf node at a depth level splits into two child nodes at the next depth level. Each leaf node of the tree contains a leaf block of data records.

To measure the performance of data browsing, we issue a range query at each depth level of the binary kd-tree that is the resolution threshold of the cut ($r$ of $O(r)$) is the level of the tree. The query region is set to the whole volume, that is $(0, 1]$ on all dimensions, so
that all hyper-rectangles at the depth level are selected. Also, the query is set to fetch both hyper-rectangles and points. Table 7.2 is the information collected after running 20 queries.

At each query we change the level information while keeping the range query large enough to select all regions.

Figure 7.3(a) shows the number of hyper-rectangles (including individual data records if leaf blocks are accessed) fetched against the tree depth level. Starting at level 17, leaf blocks are accessed and points are fetched. As we expect, we see a linear relationship until leaf blocks are accessed, which contain large numbers of individual data records.

Figure 7.3(b) shows the numbers of visited non-leaf and leaf blocks of the data aggregation tree. Please note that data aggregation tree is an unbalanced tree and a user may get

<table>
<thead>
<tr>
<th>Level</th>
<th># of Records</th>
<th>Wall time (seconds)</th>
<th># of Leaf Blocks</th>
<th># of Non-leaf Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.000</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0.000</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.000</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0.050</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>0.010</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td>0.004</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
<td>0.100</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>0.350</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>9</td>
<td>256</td>
<td>0.380</td>
<td>0</td>
<td>51</td>
</tr>
<tr>
<td>10</td>
<td>512</td>
<td>0.340</td>
<td>0</td>
<td>88</td>
</tr>
<tr>
<td>11</td>
<td>1,024</td>
<td>0.450</td>
<td>0</td>
<td>118</td>
</tr>
<tr>
<td>12</td>
<td>2,048</td>
<td>1.730</td>
<td>0</td>
<td>351</td>
</tr>
<tr>
<td>13</td>
<td>4,096</td>
<td>2.540</td>
<td>0</td>
<td>939</td>
</tr>
<tr>
<td>14</td>
<td>8,192</td>
<td>4.540</td>
<td>0</td>
<td>1,671</td>
</tr>
<tr>
<td>15</td>
<td>16,384</td>
<td>8.720</td>
<td>0</td>
<td>2,995</td>
</tr>
<tr>
<td>16</td>
<td>32,768</td>
<td>16.980</td>
<td>0</td>
<td>4,516</td>
</tr>
<tr>
<td>17</td>
<td>73,369</td>
<td>35.120</td>
<td>52</td>
<td>6,032</td>
</tr>
<tr>
<td>18</td>
<td>1,807,890</td>
<td>221.460</td>
<td>12,546</td>
<td>7,465</td>
</tr>
<tr>
<td>19</td>
<td>8,539,508</td>
<td>1,376.310</td>
<td>74,475</td>
<td>8,687</td>
</tr>
<tr>
<td>20</td>
<td>11,303,697</td>
<td>1,961.780</td>
<td>101,351</td>
<td>9,891</td>
</tr>
</tbody>
</table>

**Table 7.2** Results for Resolutions by Level Queries
query results of aggregated data mixed with individual data records. As seen in the figure, some queries are answered by fetching the same number of blocks because the information is collected from internal nodes of the internal tree of DA tree's non-leaf nodes. No leaf block is accessed for any depth level less than 17. As depth level increases from 17, the number of leaf block I/Os increases rapidly, where the resolution is high enough so that leaf blocks and individual data records are fetched. This is observed as a big jump from 0 number of leaf blocks fetched.

Figure 7.3(c) and (d) shows the wall time of query against the number of records fetched and the tree depth level, respectively. We can see that data browsing queries run pretty fast.
till a high resolution at which many individual records are fetched and compared with the query region. The comparison of many individual records with the query region takes a lot of CPU time.

![Figure 7.4](image.png)

(a) # of Records v.s. Count Limit  
(b) Time v.s. # of Blocks

Figure 7.4  Query by Count

We run another set of queries to time the query performance. In this case, we prepared 60 queries to fetch only hyper-rectangles by using the COUNT as the resolution threshold. When we increase the count limit, the query is answered fast because the hyper-rectangles with high number of points falling into are indexed close to the root. For example, when the count limit is 150 million, the number of hyper-rectangles is 1. Figure 7.4(a) depicts number of records fetched against COUNT limit. In Figure 7.4(b), we can see a dramatic increase in fetching time as we reduce the COUNT limit, because many blocks are fetched until reaching to levels where regions contain small number of points.

In conclusion, fetching aggregate data of large data sets, our data indexing structure performs well compared to scanning or recursively grouping to create similar output from a flat file or using a database management systems.
7.3 Visualization of Real Data

We have used data from Public Use Microdata Sample files (known as PUMS and available as 1% PUMS and 5% PUMS) made public by the US Census 2000. Major statistics of the data aggregation tree built from this data source are listed in the table, together with the statistics of the DA-trees. The original data files contain both housing unit records and a number of person records for surveyed people living in each housing unit. With a preprocessing, 12 variables have been chosen from the housing unit record data for this experiment. The 12 variables are STATE (state), YRBU1LT (year built), TAXAMT (property tax amount), EMPSTAT (employment status), PERSONS (number of persons living in the unit), BEDRMS (number of bedrooms), HINC (household income), FUEL (fuel type), and 4 utility usage variables including electricity, gas, water, and oil.

First data set to visualize is the 1% PUMS dataset which contains 1.25 million data records. The snapshots from the visualization of 1%PUMS data is appended to the end of this document (see Appendix B). Images are the output of visualizing the data with resolution by tree levels. Each image is from indicated level of the tree and shows the

### Table 7.3 Summary Information of PUMS Data Sets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>PUMS Samples</th>
<th>1% PUMS</th>
<th>5% PUMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Data Records</td>
<td>256 K</td>
<td>252K</td>
<td>62362K</td>
</tr>
<tr>
<td>Duplicate Records</td>
<td>2742</td>
<td>182K</td>
<td>934K</td>
</tr>
<tr>
<td>kdB-Tree Height</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>kd-Tree Levels</td>
<td>11</td>
<td>2</td>
<td>54</td>
</tr>
<tr>
<td>Non-Leaf Blocks</td>
<td>3</td>
<td>13</td>
<td>760</td>
</tr>
<tr>
<td>Leaf Blocks</td>
<td>211</td>
<td>11634</td>
<td>51516</td>
</tr>
<tr>
<td>input File Size</td>
<td>2.3 M</td>
<td>120M</td>
<td>630M</td>
</tr>
<tr>
<td>DA-tree Leaf Block Size</td>
<td>3.7 M</td>
<td>171M</td>
<td>872M</td>
</tr>
<tr>
<td>Space Utilization</td>
<td>68.10%</td>
<td>64.90%</td>
<td>71.80%</td>
</tr>
</tbody>
</table>

114
splitting points. The last image shows point and volume data records together at the level 11 where the first leaf node is accessed.

We have sampled 25,000 records from 1%PUMS data set to build a DA-tree which was used for the conducted user study. Figure 7.3 presents example screen snapshots of visualizing the data aggregation tree on the 25K samples. In the image on the left, an overview of data sample records in parallel coordinates is pictured. The visualization shows a gray data selection band across all coordinates. The data selection band specifies a query region which is used to retrieve a subset of data (individual data records in this example) that is visualized in the second snapshot. hyper-rectangles are shown and a selection is made to select records from state of Michigan excluding high utility consumption records. Consequently, a data fetch is applied to the selection and fetched records are displayed in higher resolution as shown Figure 7.3(b).

![Screen Snapshot of 25K PUMS Data](image)

(a) Overview of Data and Selection  
(b) Selected Data in Higher Resolution

**Figure 7.5** Screen Snapshot of 25K PUMS Data
The second test data set we use is the 5% Public Use Microdata Sample files (known as 5% PUMS) [1] made public by the US Census 2000. The 5% PUMS data set has the same
data format as the 1% PUMS data set. We have also chosen the same set of 12 variables from the housing unit records in this data set. Major statistics of the data aggregation tree built on the 5% PUMS data set is listed in the table. Figure 7.6(a) shows the initial query prepared to execute. Figure 7.6(b) presents an overview of all housing unit records in the 5% PUMS at tree level 3 as a result of the prepared query. The extend scale is reduced to improve visibility. All data is represented by four hyper-rectangles in this view and only 3 of them are visible. The density (#points/volume) of the forth one is too low to be visible at this view. This view shows a partition on the YR-BUILT and HINC. Figure 7.6(c) is the query set by the data selector on the previous image. Figure 7.6(d) highlights the low density hyper-rectangle. A new selection is made to filter the data to display houses built before 1970 with annual income less than 150K. Then we applied bedroom (3 or more) constraint at level 7 (not shown). In order to find houses with high fuel, electric, water and oil consumption (Figure 7.6(d)), we applied a new filter.

![Image](e) An Individual Data Record

Figure 7.7 Screen Snapshot of 5%PUMS Data - Continued
Figure 7.6(e) shows the detail of the identified data record, one house in California that has more than 3 bedrooms and built between 1960 and 1969. Its annual total income is $15K whereas the sum of electric, gas, water and oil bill ($700 + $4300 + $2700 + $3800, respectively) is $16500. The color in all these images denotes the number of records (count) in the hyper-rectangles.
CHAPTER 8

CONCLUSION AND FUTURE WORK

In this work, we tried to tackle the problem of interactive visual exploration of large data against the known limitations. We proposed to exploit density representation of large relational data to visualize instead of individual points. Therefore, a new representation between databases and visualization tools was presented as a mechanism to transmit data. In support of analytical exploration of and mining information from large data, we extended the new data format for overview-and-drill-down operations to the visualization techniques as a new data input.

We perceived records of large relational data as points in a high dimensional space, therefore we split the space into smaller subspaces, where each of them is a hyper-cubical space (hyper-rectangle) that denotes a volumetric data cube. We aggregated data points within the smallest hyper-rectangles into larger ones and continued the process for the other surrounding regions to build a hierarchical organization of multiple resolutions. Since one of the attributes of a hyper-rectangle is density, we discriminated this representation of large relational data records as an element of overview-and-drill-down access pattern.

To organize the data aggregated in multiple resolutions, we used a high dimensional tree index. We explained the detail of such a tree and called it a data aggregation tree (DA-tree). We chose a kdB-tree, which is a point access method, as the fundamental data structure for a DA-tree, because its structure and properties are in accord with our needs of density organization. We piggybacked the aggregated data onto internal nodes of the
DA-tree while storing individual data points at its leaf nodes. As a result, visual operations i.e. zooming and panning, are performed by index-only queries that access aggregated data in the internal nodes of the tree. For example, the hyper-rectangle accumulated at the root of the tree represents the whole data, thus visualizing it delineates the top most overview of the data set.

We listed advantages of multiresolution data aggregation as intermediate representation to flow data from a backend data source to final views. Its density-based characteristic conveys better presentation to human visual perception. Also, its smaller size makes it more manageable than the relational data and provides less data traffic over communication networks. Additionally, visualization becomes scalable to the size of the data, because density of hyper-rectangles (i.e. number of points) depends on the resolution levels. Over-and above, density is inherently efficacious for client side filtering such that thin hyper-rectangles below a threshold are brushed away or with a reverse-filtering are made more noticeable. Last but not least, we introduced resolutions as a new privacy preservation option.

We presented the internal layout of DA-trees. In our implementation, the node splitting problem was addressed by the root-split and the adaptive splitting techniques such that nodes are split precisely at a previous split. In spite of adding aggregation data to the internal nodes, we kept the fan-out high with our compact storage of the structure onto disk pages. Also, for the leaf nodes, we introduced the weighted longest span splitting to achieve better disk utilization to lessen the minimum occupancy degradation because of the un-balanced tree.
To show the performance and effectiveness of visualizing large data, we developed a client-server architecture. We presented our multi-threaded KDBMS system which hosts a DA-tree and delivers density-based multiresolution data aggregations. We depicted our performance results for reasonably large synthetic data. Although, the DA-tree doesn’t guarantee the minimum occupancy, it turned out to be about 60% for 300 million records with 12 dimensions which we consider an acceptable attainment. Actually, this may consolidate the DA-tree to a dynamic tree for insertions.

mVis, the visualization client of the architecture, is a framework in which we extended existing visualizations to accept multiresolution data aggregations. We implemented the scatterplot and 3D footprint splatting with grand tour, density-based and records-based parallel coordinates, and density-based scatterplot matrices. We showed effectiveness of density-based representation in visual analysis of large data. To demonstrate the interactivities, we used various mechanisms to reduce cluttering further on the clients in addition to a reduced number of records coming in from the server. Each visualizer in mVis shares its well-known attributes by linking, so that these futures become pervasive and able to be synched with the existing and future components.

We organized and visualized 1% and 5% PUMS data. The results are promising for interactive visualization and encouraging to improve and tailor the high-dimensional partitioning on the server. We invite fellow researchers and developers in visualization and related fields to visualize the new data format in their visual tools.

Even though, this research didn’t intend contemplating a public domain dynamic framework, in time, mVis became a modular construction which supports tasks and data types.
from Shneiderman's taxonomy [87]. We presented the link management system as a highway of inter-component communications. Thus, linking various visualizers and other pre- and post-processing modules happens on-the-fly. Moreover, it supports dynamic loading and integration of components to alleviate tightly-coupled and the static nature of similar tools. As part of mVis, an archetype is provided to abate the development and incorporation effort of new visualization methods and interactions. By simplifying the visualization development, we intended to facilitate teaching and learning of graphics programming by getting down to the nitty-gritty of geometry of visual technique and the rendering language (i.e. OpenGL). Finally, we introduced our toolbox which is an object layer on top of GTE + for creating object oriented GUI designs productively.

We conclude this paper by itemizing anticipated future enhancements and thoughts we bear in mind.

- Incorporation of new techniques to better split high-dimensional space into subspaces to improve visualizations are necessary.

- Splitting of a DA-tree depending on the order of insertion creates an un-balanced tree. To keep the tree height balanced, we are investigating ad hoc techniques such as re-balancing the tree during regular maintenance.

- Re-organization of disk blocks improves I/O performance for queries; therefore, efficient storage is necessary.

- To improve query performance and reduce I/O delays further, mechanisms such as
caching, prefetching etc. are indispensable. So, we plan to implement these features into our model.

- **Building a DA-tree by insertion** is much too slow for large data. Although, **bulk loading** is a prominent alternative solution, it could be a problem for high-dimensional index of large data sets. So far, there is no mechanism as elegant as B-tree bulk loading to rapidly build a high dimensional tree index.

- **Dynamic data operations altering the tree index** is another problem. For example, in insert operation, cascade splitting of nodes in the subtree causes an update of all aggregated information of nodes. Overflow nodes from insertion node to the root require a lot of aggregated data update. In delete operation, restructuring of nodes triggers update of aggregate data again. We are looking into techniques to address these problems by using statistical information of the data or by re-balancing the tree.

- Clustering large relational data is an open research issue. We want to investigate more on this subject and devise the DA-structure to support dynamic clustering as data is inserted.

- Consolidating mVis with advanced algorithms and models from other disciplines is crucial for sophisticated visual analysis.

- Basics for privacy preservation by resolution are available within a DA-tree, but its implementation and support by the server need to be done.

- Supporting panning by linking internal nodes of the high-dimensional index is a chal-
lenging issue that we want to shed some light on. Defining a “neighborhood” for a node is hard because the hyper-rectangle represented by the node has different neighbors to choose from.

To help the user command and comprehend components and visual views of images conveniently, affixing sound synthesizers and interactive interpreters is under consideration. Additionally, these components guide the user for revealing unseen and hidden information effectively and precisely. However, this idea requires an understanding of Artificial Intelligence, Machine Learning, and related fields.

Our final note is the announcement of the availability of source code for both the server and the client for public use. We hope and encourage others to modify it according their requisites.
APPENDIX A

Brief Information About the Trees Investigated*

*Some of the definitions are from Black's online dictionary [21].
**BV-tree** is a conceptual framework to generalize B-trees to higher dimensions. It is not a concrete PAM but a conceptual framework applicable to various access methods. BV-trees are not balanced and searching may require backtracking. **Promotion technique** moves lower-level intervals up closer to the root and original level number is stored in promoted region called a **guard**.

**Buddy-tree** is a flexible partitioning and reorganization schema. It doesn't partition empty space. Directory pages contain minimum bounding rectangles. Sequence of insertions is not important and performance in dynamic environment is high because of the buddy system adapted. It is not height balanced.

**LSD-tree (Local Split Decision tree)** is SAM for point and non-point objects. It has an internal memory tree, external directory pages and data buckets. The universe of data is partitioned into disjoint cells. The internal memory tree is a kd-tree. If the tree doesn't fit into the memory, selected subtrees are paged out to the external directory with a paging algorithm. Its external tree is a heap of fixed height. Split position is local optimum, that is independent of other cell boundaries and previous split decisions. Split dimension and position are kept in each node. Local split makes the structure behave well with different data distributions.

**hB-tree (Holey Brick tree)** is related to kdB-tree by organizing its internal nodes with a kd-tree. Node splits in kd-tree creates holey bricks (brick, which smaller bricks extracted from). Several leaves of kd-tree in an hB-tree may refer to one hB tree node. hB-tree achieves good insert by only splitting nodes on one root-to-leaf path.

**CRB-tree (Compressed Range B-tree)** is an indexing scheme for 2D range-aggregated
queries. It consists of two B-tree: one is constructed on the x-coordinate of points, with each node has a special structure, and the other is constructed on the y-coordinate of points. The internal structure of the first tree has two arrays which stores index and count information of the subregions. In the internal representations, bit-wise model is used.

**Bkd-tree** is an I/O efficient high dimensional indexing tree based on kd-tree. It has a kd-tree in main memory and forest of kd-trees on external storage. If in-memory tree has size of $M$, then external memory tree $T_k$ has size of $2^k \times M$ where $k$ is $0 \leq k$. Once in-memory tree is full, first empty external memory tree $T_k$ is found. Then all points in trees $T_i (0 \leq i \leq k)$ including in-memory are dumped and bulk loaded into tree $T_k$. In this structure multiple trees has to be searched in order to answer a range query.

**kdB-tree** is basically a hard-disk implementation of kd-tree, which is a height balanced binary tree where each internal node partitions the hyperrectangle represented by the node along a particular dimension. This structure created in anticipation of performing the multidimensional search efficiency of kd-trees and I/O efficiency of B-trees. kdB-tree tries to improve I/O efficiency by storing multiple nodes in one disk block. kdB-tree is height balanced. Unlike B-tree, however, kdB-tree does not guarantee the minimum page occupancy.

We have evaluated the structures discussed above. CRB is a complex data structure and supports only 2D. LSD's paging algorithm impedes its efficiency. Also, its height and minimum occupancy are unpredictable. BV trees need backtracking in searches and yet it is complex to support multiresolution data aggregations. It tree consumes extra space for extracted bricks to split the nodes. Bkd-tree needs multi tree search and this leads higher
I/O operations. We opted to implement our DA-tree using a kdB-tree as our selected multivariate data structure because it is easily extendable to support multi-resolution aggregated data. Also, there is already a few other kdB-tree variants available that our proposed DA-tree can be implemented with. Additionally, its structure is more suitable and simpler to implement as an external memory algorithm than the others. Finally, TPIE [9] had a sample implementation ready to start with. Therefore, to implement the density-based methodology, kdB-tree chosen as the base high dimensional index.

<table>
<thead>
<tr>
<th>PAMS</th>
<th>Rectangular</th>
<th>Height-Balanced</th>
<th>Guaranteed Minimum Occupancy</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>kdB-tree (1981)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>A basic PAM on secondary storage. Tradeoff between height-balancing and load distribution</td>
</tr>
<tr>
<td>LSD-tree (1989)</td>
<td>✓</td>
<td></td>
<td></td>
<td>A PAM for nested, A list is in memory linked to external heaps.</td>
</tr>
</tbody>
</table>

Table A.1 Multidimensional Point Access Methods
APPENDIX B

Images from Visualization of 1% PUMS by Levels
Figure B.1  Visualizing UMS by Level.
Figure B.2 Visualizing 1%PUMS by Levels–Continued

(e) Level 5

(f) Level 6

(g) Level 11
APPENDIX C

Inheritance Diagrams
Figure C.3  Inheritance Diagram for Toolbox Objects

Figure C.4  Inheritance Diagram for Multi-line Textbox Object
Figure C.5   Inheritance Diagram for Scale Object

Figure C.6   Inheritance Diagram for Visualizer Base Class
Figure C.7 Inheritance Diagram for Window Classes
BIBLIOGRAPHY


13. BAUDEL, T. From information visualization to direct manipulation: extending a
generic visualization framework for the interactive editing of large datasets. In *Proc.
19th Annual ACM Symposium on User Interface Software and Technology (UIST 06)*

14. BAYER, R. The universal B-tree for multidimensional indexing: General concepts.
In *Proc. Inter. Conf. Worldwide Computing and Its Applications WWCA’97* (London,
UK, 1997), Springer-Verlag, pp. 198–209.

15. BAYER, R. AND MCCREIGHT, E. M. Organization and maintenance of large or-

16. BECKMANN, N., KRIEGEL, H.-P., SCHNEIDER, R., AND SEGERER, B. The R*-Tree:
An efficient and robust access method for points and rectangles. In *Proc. ACM

17. BRENTLEY, J. L. Multidimensional binary search trees used for associative search-

18. BERCHTOLD, S., BÖHM, C., AND KRIEGEL, H.-P. Improving the query per-
formance of high-dimensional index structures by bulk-load operations. In *Proc.
6th Inter. Conf. Extending Database Technology (EDBT 98)* (London, UK, 1998),

19. BERCHTOLD, S., KEIM, D. A., AND KRIEGEL, H.-P. The X Tree: An index-
structure for high-dimensional data. In *Proc. Inter. Conf. Very Large Databases

20. BISWAS, G., JAIN, A. K., AND DUBES, R. C. Evaluation of projection algo-
rithms. *IEEE Trans. Pattern Analysis and Machine Intelligence (PAMI)* 6, 6 (Nov.

21. BLACK, P. E. Dictionary of algorithms and data structures [online]. *U.S. Na-
tional Institute of Standards and Technology* (December 2004).

22. BLACK, C. L., AND MERZ, C. J. UCI repository of machine learning databases,

23. BOUKHELIFA, N., AND RODGERS, P. J. A model and software system for coor-
dinated and multiple views in exploratory visualization. *Information Visualization* 2,
4 (2003), 258–269.

24. CHI, E. H. Expressiveness of the data flow and data state models in visualization


32. **Doshi, P. R., Rundensteiner, E. A., and Ward, M. O.** Prefetching for visual data exploration. In *Eighth International Conference on Database Systems for Advanced Applications (DaSFAA'03)* (March 2003).


34. **Fanee, E., Carpendale, S., and Isenberg, T.** An interactive 3D integration of parallel coordinates and star glyphs. pp 149–156.


53. INSELBERG, A. The plane with parallel coordinates, special issue on Computational geometry. The Visual Computer 1, 2 (1985) 59–91.


101. WARD, M. O., RUNDENSTEINER, E. A., YANG, J., Doshi, P. R., and Rosario, G. Interactive poster: Xmdvtool: Interactive visual data exploration system for high-dimensional data sets. pp. 52–53.


