INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

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 bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI 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.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Visualization and Performance of Dynamic Multipaging Algorithms

by

Gautam Jain, B.E.

A Thesis Submitted to
the Faculty of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

September 1997

© Copyright

1998, Gautam Jain

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-32387-0

Canada

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,
Visualization and Performance of Dynamic Multipaging Algorithms
submitted by
Gautam Jain, B.E.
in partial fulfillment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University
March 20, 1998

Abstract

Multidimensional storage structures provide a useful way of organizing data in a multidimensional fashion. Using good indexing data structures, data stored in a multidimensional fashion is accessed faster. Moreover, multidimensional data storage suits very well those applications that are characterized by multidimensional data, such as GIS (Geographic Information Systems), CAD (Computer Aided Design), VLSI (Very large scale integration) etc.

This thesis attempts to simplify the understanding of the complexity of multidimensional storage structures by means of visualization. Dynamic Multipaging is our multidimensional storage structure of choice in illustrating the process of Insertion, Deletion and querying.

The main contributions are 1) development of a visualization tool for the insertion, deletion and querying (partial match and range) algorithms of dynamic multipaging 2) a generalized N dimensional implementation of dynamic multipaging, and 3) performance study of dynamic multipaging. The results indicate that Dynamic Multipaging has high storage efficiency and processes different classes of database queries with reasonable efficiency for uniform data distribution.

Acknowledgments

I would like to thank my supervisor, Ekow J. Otoo for his generosity, guidance, especially his patience while editing the thesis drafts, which made all things possible for me.

I would also like to acknowledge the financial support from the School of Computer Science, Carleton University.

Contents

Abstract	iii
Acknowledgments	iv
List Of Tables	viii
List Of Figures	x
1 Introduction	1
2 Background and Related Work	6
2.1 Hashing Structured Methods for Multidimensional Data Organizations	6
2.2 Tree Structured methods for Multidimensional Data Organizations . .	9
2.3 Visualization	11
2.4 Visualization of Multidimensional Data	13
2.5 Visualization of Large Datasets	14
2.6 Static Multipaging Structure	15
2.7 Dynamic Multipaging Structure	17
2.7.1 Dynamic Multipaging for Multidimensional Data Organization	17
2.7.2 Concepts of Dynamic Multipaging Structure	18
2.7.3 K-Dimensional Extendible Array (KDEA)	20
2.7.4 Page Address Calculation	22

3	Dynamic Multipaging Structure Design	24
3.1	Basic Data Structures	24
3.1.1	Scales	24
3.1.2	Directory Organization	27
3.1.3	Data Pages	28
3.2	Page Address Calculation	29
3.3	Splitting and Merging Policies	32
3.3.1	Splitting Policy 1	33
3.3.2	Splitting Policy 2	33
3.3.3	Splitting Policy 3	34
3.3.4	Merging Policy	35
3.3.5	Summary	36
4	Visualization	37
4.1	Visualization Model	37
4.2	Visualization Technique	38
4.3	Implementational Details	40
4.4	Visualization Set Up	43
4.5	Two (2) Dimensional Visualization	48
4.5.1	Example of Directory Expansion	48
4.5.2	Example of Directory Contraction	51
4.6	Extension to Visualization of Large Datasets	54
4.7	Visualization in Higher Dimensions	55

5	Simulation Experiments	58
5.1	Objectives of the Simulation	59
5.2	Simulation Model	60
5.3	Simulation Results	62
5.3.1	Utilization of data pages	62
5.3.2	Directory Fan Out	67
5.3.3	Probe Factor	70
5.3.4	Disk Accesses For Record Insertion	72
5.3.5	Query Processing Strategy and Performance	74
5.4	Performance Comparison	79
5.5	Summary	80
6	Conclusion	81
	References	83

List of Tables

3.1	Multipaging Directory	28
5.1	Performance Of Splitting Policy 1 for Data not Distributed Uniformly in 2 Dimensions	68
5.2	Performance Of Splitting Policy 1 for Data not Distributed Uniformly in 3 Dimensions	68
5.3	Query Performance Results for 10,000 records in 2 Dimensions with Uniform Data Distribution	77
5.4	Query Performance Results for 10,000 records in 2 Dimensions with Data not Distributed Uniformly	77
5.5	Query Performance Results for 10,000 records in 3 Dimensions and d=1 for Partial Match and Partial Range Queries with Uniform Data Distribution	77
5.6	Query Performance Results for 10,000 records in 3 Dimensions and d=1 for Partial Match and Partial Range Queries with Data not Dis- tributed Uniformly	78
5.7	Query Performance Results for 10,000 records in 3 Dimensions and d=2 for Partial Match and Partial Range Queries with Uniform Data Distribution	78

5.8	Query Performance Results for 10,000 records in 3 Dimensions and d=2 for Partial Match and Partial Range Queries with Data not Dis- tributed Uniformly	78
5.9	Performance Comparison for Uniform Data Distribution	79
5.10	Performance Comparison of Data not Distributed Uniformly	79

List of Figures

2.1	Visualization Model	12
2.2	Schematic Layout of a 2DEA $A[0:3, 0:3]$	20
2.3	Array $A[0:3,0:3]$ extended to $A[0:3,0:4]$	21
3.1	Binary Tree Structured Scales	25
3.2	Binary Tree Structured Scales	26
3.3	Buddy Relationship	27
3.4	Extendible Array with Axial Directories	30
3.5	Page Address Calculation Procedure	31
4.1	Visualization Model	38
4.2	Code Organization	41
4.3	Application Structure	42
4.4	Graphical User Interface	45
4.5	Query User Interface	46
4.6	Visualization Interface	47
4.7	Initial Situation of the Structure	49
4.8	Result of Directory Expansion	50
4.9	Initial Situation of the Structure	52
4.10	Result of Directory Contraction	53
4.11	Directory Structure for a Large Dataset	54
4.12	Example of Visualization of Large Dataset in 2 Dimension	56

5.1	Data Page Utilization for uniform data in 2 and 3 Dimensions	64
5.2	Data Page Utilization for uniform data in 2 and 3 Dimensions for Real Data	65
5.3	Total number of pages for 2 and 3 Dimensions	66
5.4	Total number of pages for 2 and 3 Dimensions for Real Data	67
5.5	Fan out for 2 and 3 dimensions	69
5.6	Fan out for 2 and 3 dimensions for Real Data	69
5.7	Probe factor for 2 and 3 Dimensions	71
5.8	Probe factor for 2 and 3 Dimensions for Real Data	71
5.9	Number of overflow records for 2 and 3 Dimensions	72
5.10	Number of overflow records for 2 and 3 Dimensions for Real Data . . .	73
5.11	Average number of disk accesses for 2 and 3 Dimensions	73
5.12	Average number of disk accesses for 2 and 3 Dimensions for Real Data	74

Chapter 1

Introduction

Present database management systems allow queries to be expressed on any combination of attributes of the requested records or tuples; however traditional databases files are organized on a single key. There are a number of database applications where it is very common to perform searches using the values of several attributes. Examples of such applications are Very Large Scale Integration (VLSI) design, Computer Aided Design (CAD), Geographical Information Systems (GIS), medical applications representing the arrangement of chains of protein molecules in 3-D space etc. These applications require data organizations that provide multikey access to records. Accessing data on multiple attributes of a file is referred to as associative searching. Thus attention has been drawn towards efficient organization of databases suitable for associative searching. Critical to the performance of a database is the storage organization and the technique of accessing desired tuples in response to a query which in turn requires good search structures.

A lot of work has been done on the effectiveness and usefulness of multidimensional storage structures suitable for fast associative searching. Multidimensional data organizations are characterized by the following properties.

1. Each record is identified by multiple independent keys and the domain of each key is large and linearly ordered.

2. All keys are treated symmetrically, i.e., the multidimensional organization does not favor the primary key at the expense of secondary keys. Therefore queries that involve different keys are processed with equal efficiency.

An important component of a multidimensional storage structure is its usage as an indexing structure. A good indexing structure should possess the following properties.

1. Good storage utilization of indexes and data pages.
2. Fast exact match search.
3. Good clustering on all keys for reasonably fast range searches.
4. Large index fan-out, resulting in a small index and a small number of disk accesses.
5. Easy incremental reorganization as the file representing the database grows.

Several multidimensional storage structures (MDS) that are suitable for both associative searches and provide a satisfactory response to the different classes of queries (partial match, range and partial range) have been proposed. These include the Hashing Methods: Interpolation-based Index Maintenance [5], Multi-dimensional Linear Hashing [7], Multi-dimensional Linear Hashing with Partial Expansion [8], Dynamic Multipaging [9], Grid File [10], Multi-dimensional Extendible Hashing [11], Multi-dimensional Extendible Hash Tree [12], Multi-dimensional Linear Dynamic Hashing [13] and the Tree Structured Methods: Multi-dimensional Binary Search Trees [15], Quad Tree [16], Multi-dimensional B-Tree [17]. But all of these have either logarithmic access paths to the data pages or are not dynamic in nature. Dynamic direct access schemes for single attribute key searches are Extensible Hashing [19], Linear Hashing [20], and [14]. A dynamic multidimensional direct access scheme has been proposed, Extensible Cell [21], but can use arbitrarily large amounts of memory.

Despite these advantages they have not been popular or readily employed in the industry. The reason is attributed to the fact that they are a little difficult to conceptualize. The complexity of understanding multidimensional storage structure is probably the main cause for its underutilization. Our approach to the above problem is to simplify the understanding of the complexity of multidimensional storage structures by means of visualization.

This thesis proposes to facilitate the understanding of multi-dimensional storage schemes by providing a visualization tool that demonstrates through visual representation the organization of data in a multi-dimensional storage structure. The process of splitting and merging and their effect on the storage structure in response to insertion and deletions in the database.

For our implementation purpose we need to select a multidimensional storage structure and an indexing structure keeping in mind the desirable features of both as mentioned above. For our purpose we have selected Dynamic Multipaging [9] as our Multidimensional storage structure because Dynamic Multipaging is superior to any known multidimensional data organization for uniformly distributed data. Dynamic Multipaging uses a dynamic order preserving multiattribute key-to-address function to address its data space.

Multidimensional storage structures provide means for organizing data in multidimensional data space with each axis representing an attribute. The data space is partitioned into subspaces or regions such that each subspace or region stores tuples that fall in that subspace or region range. The multidimensional space is indexed by means of a directory structure. The directory structure constitutes data structures that provide access to the data stored in the storage structure. These data structures include linear arrays, binary trees, Balanced trees, R trees etc. Binary trees are efficient search structures and their structure very well represents the history of the partitions of a multidimensional space. A binary tree is a collection of components arranged in a hierarchical fashion. A component at each level can be used to logically divide data space into two distinct subspaces or regions. Additional information can be stored in these subdividing components representing the history

of the partition at that point. Binary tree can be easily extended to index very large datasets. Performance benefits of using binary trees as accessing structures have been reported in Ying Lin's thesis [25]. Thus our approach is to use binary tree structured (BTS)-scales to index data organized using Dynamic Multipaging. Simulation studies are carried out to establish the performance benefits of dynamic multipaging.

Thus the advantages of multidimensional storage structures are 1) they provide a useful way of organizing data in a multidimensional fashion 2) they suit several applications mentioned above very well and 3) they facilitate clustering of data thereby providing faster access to the data they store.

There exist several visualization tools such as the VisDB system [33], Navigational View Builder [36], Tioga-2 [37] and visualization techniques Parallel Axis Technique [39], Use of cons or glyphs [40], linked display [45], pixel-per-value [33, 34] that the above visualization tools use. Using existing visualization tool(s) for visualizing dynamic multipaging poses two main problems: 1) it is not trivial to apply a visualization tool to applications for which it has not been built, i.e., it may not be easy to integrate an existing visualization system with a user developed application. The system has to be tightly integrated with the application whose data is to be visualized. Tight integration is necessary if the visualization tool modifies the data retrieved for visualization. The integration problem may arise due to the difference in design methodologies used for the visualization system and the application, difference in platform requirements for running the visualization system and the application etc. and 2) our visualization requires support for animation, such as animating the traversals of two binary trees simultaneously. The latter feature would require for instance creating two threads and handling traversal of each binary tree in a separate thread of control.

We identified Java as the most optimal solution for our visualization. The programming language constructs, support for multi-thread programming and a class library for simple graph drawing seemed ideal for our dynamic multipaging structure implementation and visualization. The use of Java also simplifies the problem of in-

tegration of the visualization system and the application as they are both developed in Java.

We thus identify two goals in this thesis. One is to measure performance of dynamic multipaging to show it is a good candidate from a performance point of view and the other is to develop a visualization tool for visualizing the working of a multidimensional storage structure.

The main results achieved in the thesis are

- implementation of dynamic multipaging algorithms (insertion, deletion and query).
- development of a visualization tool that visualizes the insertion and deletion algorithms of dynamic multipaging 2 Dimensions. The tool simplifies the understanding of the complexity of multidimensional storage structures.
- simulation results establishing the performance benefits of employing dynamic multipaging for uniform data distribution in 2 and 3 dimensions

The outline of the rest of the thesis is as follows. In chapter 2 we present the background work related to multidimensional storage structures and visualization of data structures. In chapter 3, we present our design of the Dynamic Multipaging structure with the various splitting and merging policies that are implemented. For better understanding of multidimensional storage structures we demonstrate the visualization of dynamic multipaging algorithms (insertion and deletion) in 2-dimensions in chapter 4. The visualization shows how the splitting and merging policies of dynamic multipaging affect the growth and shrinkage of the structure. Splitting policy is visualized by showing how the directory structure is partitioned in response to an insertion and merging policy demonstrates how the directory structure shrinks in response to a deletion. The visualization is implemented in Java. Performance of our implementations is presented in Chapter 5 and we present our conclusions and scope for further research in Chapter 6.

Chapter 2

Background and Related Work

Data structures for representing points in a k -dimensional space have a much longer tradition than spatial database system. This is because a tuple consisting of n attributes; $t = (x_1, \dots, x_k)$, can be viewed as a point in k dimensions, and, therefore, such data structures can be used to support multi-attribute retrieval. On the other hand, they can also store points with a geometrical representation. Such data structures can be classified in two main categories namely hashing methods and tree structured methods.

2.1 Hashing Structured Methods for Multidimensional Data Organizations

Multidimensional hashing methods are all based on one or the other of two 1-dimensional dynamic hashing, namely: Extensible Hashing [19] and Linear Hashing [20]. The characteristics of Extensible Hashing are 1) no overflow records are organized in a special overflow area and 2) a record retrieval is guaranteed in at most two disk accesses. The disadvantage of Extensible Hashing is that the directory size can grow exponentially in the number of record insertions unless the keys are uniformly distributed over the directory space. Therefore, Extensible Hashing may generate a bad index page utilization and high *fan out* ¹. A number of tech-

¹The number of indices emanating from a node

niques have been proposed in [22] to overcome this problem and also to achieve an order preserving hashing scheme to support range queries. In Linear Hashing, a dynamically changing function is used to compute the address of the data pages. The main advantages of the Linear Hashing function are 1) the size of the data file grows and shrinks gracefully; 2) there is no index and 3) the utilization of allocated storage can be controlled. One major disadvantage of this scheme is that it requires the use of a good collision resolution method in order to avoid poor record retrieval times. There is a trade-off between the data page utilization and retrieval times.

The above two hashing techniques, however, emphasize only single-key cases. These techniques must be extended to the design of multidimensional data organization. Interpolation Based Index Maintenance [5] and Storage Mapping for Multidimensional Linear Dynamic Hashing [13] are multidimensional file structures which are derived from Linear Hashing. In a manner similar to Linear Hashing, these methods assign regions to data pages by means of mapping functions and use collision resolution by separate chaining to handle overflow. Both methods are sensitive to non-uniform distribution because the guarantee of a predefined minimum load factor results in overflow chains that may be long and therefore reduce performance. Their solutions have been proposed in [7, 8]. In order to completely avoid the overflow problem, Multidimensional Extendible Hashing [11] and the Extendible Cell Method [21] were proposed. These are based on Extensible Hashing. As in Extensible Hashing, these two methods are related by the fact that they both maintain directories whose entries point to data pages of corresponding regions of the partitioned attribute space. Collision resolution is not necessary as no overflow can occur. However, non-uniform data distribution may cause the directory to grow at an exponential rate in the number of records.

The grid file [10] is a good search structure for uniformly distributed multi-attribute keys. However, if the distribution of the attributes of the keys are skewed, the index can use large amounts of space. The initial implementations of the grid file by Hinrichs [24] with linear scales suffers from large directory growth. This problem was overcome in interpolation based grid file [13]. It uses a representation of the

data space in which only page regions are stored in the grid directory. As a result, the number of directory entries is linearly proportional to the number of data pages. However, when data in a page region overflows, there is no choice in the subsequent splitting sequence, which may create empty data pages. Thereby reducing storage utilization and the directory would be filled with entries with sparsely populated pages. Arbitrary deletions sometimes generate deadlocked empty regions that are impossible to eliminate as they occur. These limitations were resolved in Ying Lin's thesis [25].

This thesis presents one level extendible grid file to avoid reorganizing the directory at each expansion step and a multilevel grid file to offset the exponential growth of the directory. The deadlock problem is resolved by applying a merging policy of reversing the splitting operation. To improve the storage requirement of the interpolation based grid file, so called BANG file [27] was proposed. The interpolation based grid file splits a data page region into a number of k -dimensional rectangular page regions of possibly different splitting levels and therefore of unequal size. But in a BANG file, it is no longer necessary to impose the restriction that all page regions must be rectangular boxes. This induces a new kind of page regions which are obtained from a k -dimensional rectangular area from which, possibly some k -dimensional rectangular holes are carved out.

The procedure of splitting an overflow page region is recursive. If one of these two page regions into which the overflow page region has been divided contains more records than the other, the page region with a large population is progressively halved until the balance of the number of records in the two pages is reserved. During this process, the best balance is chosen. The balancing technique guarantees that there are never empty data pages. Another advantage associated with the BANG file is that the merge algorithm is more flexible in responding to changes in data distribution. If the population of a page region sinks below some predefined minimum, an attempt is made to merge it with one of the page regions which it immediately encloses, starting with the smallest. If it fails, final attempt is made to merge it with its immediately enclosing region which must exist.

2.2 Tree Structured methods for Multidimensional Data Organizations

Amongst the earliest and popular tree structured methods for accessing rectangles is the R-tree [26]. R-tree is a height-balance tree similar to a B-tree with index records in its leaf nodes containing pointers to data objects. Nodes correspond to disk pages if the index is disk resident, and the index is designed such that a spatial search requires visiting only a small number of nodes. The index is completely dynamic and no periodic reorganization is required in response to insertion and deletion. The search algorithm descends the tree from the root in a manner similar to a B-tree.

An R-tree stores multidimensional rectangles as complete objects without clipping them to higher dimensional points before. A non-leaf node contains entries of the form $(cp, Rectangle)$ where cp is the address of a child node in the R-tree and $Rectangle$ is the minimum bounding rectangle of all rectangles which are entries in that child node. A leaf node contains entries of the form $Oid, Rectangle$ where Oid refers to a record in the database, describing a spatial object and $Rectangle$ is the enclosing rectangle of that spatial object. The main problem in R-trees is the following: For an arbitrary set of rectangles, dynamically build up bounding boxes from subsets of between m and M (M is the maximum number of entries that will fit in one node and m is a parameter specifying the minimum number of entries in a node ($2 \leq m \leq \frac{M}{2}$)) rectangles, in a way that arbitrary retrieval operations with query rectangles of arbitrary size are supported efficiently. The known parameters of good retrieval performance affect each other in a very complex way, such that it is impossible to optimize one of them without influencing the other parameters which may cause a deterioration of the overall performance. Moreover, since the data rectangles may have very different size and shape and the directory rectangles grow and shrink dynamically, the success of methods which will optimize one parameter is very uncertain.

R-tree is essentially based on a heuristic optimization. The optimization criterion which it pursues is to minimize the area of each enclosing rectangle in the inner

nodes. This criterion is taken for granted and not shown to be the best possible. An R^* -tree incorporates a combined optimization of area, margin and overlap of each enclosing rectangle in the directory. Performance comparison of R-tree and R^* -tree show that the R^* -tree outperforms R-tree. The advantages with R^* -tree is that it can be used efficiently as an access method in database systems organizing, both multidimensional points and spatial data.

In the tree structured methods, the partitioning boundary values of the space serve as discriminating values for the internal nodes of an index whose terminal nodes are the pointers to the data pages holding the records. Such design methods are exemplified by the heterogeneous K-D tree [15] and Quad tree [16]. Quad tree is the most straightforward d dimensional generalization of binary search tree, but it is impractical because the tree nodes become large and contain many nil pointers. These problems are avoided in K-D trees and binary TRIEs [22], both of which can be thought of as an efficient implementation of the d - dimensional generalization of quad trees. K-D trees share many properties with binary search trees. The K-D-B-tree of Robinson works analogously to a B-tree, instead of nodes containing search values in disjoint intervals of a one-dimensional space, each nodes "covers" a brick-like region of a k -dimensional space. This K-D-B-trees inherit from B-trees the balanced free property, that is, all paths to leaves of the tree are equal in length. Further, like B^+ -trees, all data is stored in leaf nodes, internal nodes containing only index entries direct the search.

The extension to the K-D-B-tree of Robinson is the hB-tree [29]. The two distinct features of hB-trees are: 1) index nodes are organized as K-D-trees and 2) splitting of a node may require the participation of more than one attribute as opposed to K-D-B-tree and grid file that split nodes by single attribute. The result is that nodes no longer correspond to bricks in k -dimensional space, but rather to "holey" bricks or bricks from which smaller bricks have been removed. Because of this, hB-trees can avoid cascading splits. The major disadvantage with hB-trees is the use of K-D-trees as the structures within hB-tree nodes and K-D-trees consume more space than the usual vector-oriented binary search organization. This problem is overcome

in the extension to hB-trees called hB^π -trees [30] proposed in 1997. This is because the size of the K-D-trees that reside in the hB^π -tree nodes does not depend on the number of indexing attributes. hB^π -tree is a multi-attribute index that would be appropriate for inclusion in a general purpose DBMS.

2.3 Visualization

Visualization is one of the most important activities in modern exploratory data analysis. General purpose visualization systems have proven to be valuable tools for scientific research in various disciplines. As scientific data grow in size and complexity, it becomes apparent that data management capability should be incorporated into visualization systems. Many commercial visualization systems have already been equipped with data management tools. However most of them are *ad hoc* implementations tuned for specific visualization systems. The other trend is to integrate visualization systems with existing general purpose database management systems. The challenge of this approach lies in developing an effective and streamlined interface between the visualization system and database system.

The general objective of visualization is to explore hidden relationships, patterns or characteristics of data with the aid of human visual perception. From a wider perspective, visualization only plays a part - a very important one, nonetheless - in modern exploratory data analysis. As far as interactive data exploratory system is concerned, the following figure shows the system components.

The data visualization component enables a user to visualize, among other things, the database structure, base data in the database tables, database queries and data analyses and obtain new insight into the data by using different visualizations of the data. Thus it is a crucial component to support navigations through data space and the system components. Visualization allows the user to configure displays and control selection, analysis and display parameters by providing consistent and intuitive visual metaphors. It guides the exploration and serves as the glue that holds the environment together. A visualization system based on the above model

Figure 2.1: Visualization Model

is the VisDB system [33]. Another approach is to develop an interface between the visualization system and the database as in the Exbase system [35]. However, this approach has the impedance mismatch problem between the database and the visualization.

Other than the architecture shown in figure above there are several other requirements for building an information visualization tool. An experimental system called Navigational View Builder (NVB) [36] was built to develop visualizations of the information space of hypermedia systems. It has been used to form overview diagrams for a section of the most popular hypermedia system, the World-Wide-Web. NVB also supports multiple simultaneous visualizations. During the course of developing NVB, several requirements for building an effective information visualization tool were identified.

Yet another visualization system that applies database visualization to the World-Wide-Web is the *Hy⁺* [38]. Another visualization tool developed at Berkeley is the Tioga-2 [37] database visualization environment. Tioga-2 is based on a small set of primitive operations for transforming data and its visualization.

2.4 Visualization of Multidimensional Data

The need for visualizing multidimensional data is due to the increased number of applications that are characterized by multidimensional data, and the size of the data that these applications maintain. For instance, financial data, census data, survey data of all types, socio-economic data and medical records are intrinsically multidimensional and can and do involve tens or even hundreds of millions of data points, and are of interest to a wide audience of scientists and non-scientists alike. However, a satisfactory method of graphically representing data involving more than three variables is not easy and has in fact occupied several investigators over the last several decades. A range of techniques involving projections of the data onto two variable subspaces has evolved. Non-projective techniques have also been invented, such as, the parallel axis technique [39] and the use of icons or glyphs [40]. These techniques are however good for certain types of multivariate data and do not support visual analysis of data in real time. A technique for plotting large multivariate data sets was proposed by Timlin [41]. The technique used involves the mapping of n independent variable dimensions to a single hierarchical horizontal axis with a single dependent variable being plotted on the vertical axis. However, this technique is difficult to generalize to higher dimensions. In addition to iconographic displays, graphical methods for exploring multidimensional spaces are becoming more and more popular. A manual method is to produce several linked (potentially orthogonal) plots of a data space, select data in one display using the cursor and observe the data in a linked display [45]. In this way, multidimensional clusters and other relationships can be determined. Similar, automated processes include rotating between data spaces [44] has also been used. [45] extends his earlier work to incorporate high dimensional data visualization. Some of these visualization techniques have also been employed for practical purposes. Examples of these are 3-D visualization of large databases by [42], the use of cone trees [28] for animated 3-D visualizations of hierarchical information.

New technologies such as, VRML [50] have been identified as useful tools for

visualization of 3 dimensional data. VRML is a scripting language that provides constructs for performing several useful operations on 2 and higher dimensional objects. These operations include rotating an object, providing different views of an object and handling animated images to provide animation effects. The advantage of using VRML is that it can easily be integrated with high level programming languages such as Java [31, 32] and thereby use the powerful programming facilities of the language, such as, Java threads [51] to perform animations of the graphical data. The technique of integrating Java and VRML thereby using all the powerful programming language constructs combined with the powerful visualization constructs of VRML are being used for multimedia applications, such as representation of multimedia objects including text documents and graphical images.

2.5 Visualization of Large Datasets

In traditional data base management systems there is no possibility to visualize the result of a query or the relevance of the data items with respect to the query. New techniques directly presenting huge amounts of data to the user are needed if data visualization shall be used to support exploring large databases. An example of such a technique is the pixel-per-value developed in the VisDB project [33, 34]. The basic idea of pixel-per-value technique is to map each data value to a colored pixel and present the data values belonging to each dimension in a separate window. This permits visualization of up to a million values. The pixel oriented techniques use different arrangement of pixels for different purpose. The VisDB system is a sophisticated tool for visualizing and analyzing large databases and the interactive graphical user interface allows a direct interaction with the data. Users may for example, interactively change the query ranges, highlight corresponding pixels in different windows, use a projection of the visual representation to specific color ranges, or access the attribute values that correspond to some specific pixels. The VisDB systems has been used in a molecular biology project to find possible docking regions by identifying sets of surface points with distinct characteristics. Other

data sets being explored include large databases of geographical data, a large environmental database and a NASA earth observation database. One problem of the query-dependent pixel-oriented visualization technique is that their local clustering properties are insufficient. Keim [46] generalized the pixel-oriented techniques and proposed a new variant which retains the overall arrangement but enhance the clustering properties by using screen-filling curves locally. Mukherjea [47] discusses a technique for visualizing large data sets bases on multiple hierarchical views.

Our visualization model is based on a reference model for visualization of multidimensional data [40]. The model consists of three components. Namely, the user, the visualization interface and the application holding the data to be visualized. The model meets the requirements for our visualization system. It provides a means for integrating and interfacing the three components of our visualization system. The graphical display of the application data in our visualization interface exploits the features of the pixel-oriented technique [33]. This technique uses the screen pixels to represent the records of the database (application data) as a point in a 2-dimensional space. The graphical display of application data (records) as points in a 2-dimensional space is accomplished by using the Abstract Windowing Toolkit (AWT) classes of the Java [31, 32] programming language. We have used the multi-threaded programming features of Java to accomplish the animation in our visualization system. In summary, the reference model, the pixel-oriented technique and the Java programming language constructs have been used to implement our visualization system.

2.6 Static Multipaging Structure

Multidimensional paging is a storage structure for physical organization of multidimensional data. The structure is comprised of page regions for storing the data and a relatively small directory and a page addressing function for computing the address of data pages. The page addressing function addresses the primary page in which a tuple is stored and not the exact location within the page. The term

Multipaging refers to the static multipaging model described in [3].

The structure assumes a paging environment and organizes tuples into clusters, with tuples of the same cluster being assigned to the same page. Tuple access is done by first accessing the candidate primary page and then searching this page for the target tuple.

Given a relation $R(A_1, A_2, \dots, A_m)$. Queries may be expressed on a subset $Y = (A_1, A_2, \dots, A_k), k \leq m$. These k attributes are called the key-attributes and normally should uniquely identify a tuple. The k selected attributes define a k -dimensional space where each attribute corresponds to an axis. Tuples occupy unique points defined by the k -coordinate attribute values in some order. The attribute values of the axes $(A_1, A_2, \dots, A_k)$, are divided into $(f_1, f_2, \dots, f_k)$ intervals respectively, of possibly varying sizes. The attribute space is thus partitioned rectilinearly into p subspaces or regions, where $p = \prod_{i=1}^k f_i$. A cluster then is made up of tuples whose point images fall in the same region and are assigned to the same page. Multipaging maintains a directory for each axis-attribute. The directory of axis-attribute i , denoted as Dir_i has f_i entries; a total of only $\prod_{i=1}^k f_i$ directory entries for all axes. The j^{th} directory element of axis-attribute i is denoted by Dir_{ij} . A given value of attribute A_i determines an index number J_i from the directory of the i^{th} axis. To determine the primary page address, the k index numbers $(J_1, J_2, \dots, J_k)$ are located and then the page addressing function $H_s(J_1, J_2, \dots, J_k)$ is used to calculate the actual page address.

Given a page capacity C , a relation of cardinality N , the factors $f_1, f_2, \dots, f_k$ may be determined by a heuristic discussed in [3]; namely that $\frac{V_i}{f_i}$ be approximately constant, where V_i is the number of different attribute values on axis-attribute i . The manner in which the attribute space is partitioned may cause the tuples allocated to some pages to exceed the capacity and hence cause overflow. Overflow tuples are allocated to under-filled pages which are chained to the primary page. Two measures of performance determine how good a multipaged relation is. These are the probe factor π and the load factor α . The probe factor is defined as

$$\pi = 1.0 + \frac{N_0}{N} \quad (2.1)$$

where N_0 is the number of overflow tuples. π is an approximate measure of the average number of page accesses required for an exact-match search. The load factor is defined as

$$\alpha = \frac{N}{P \times C} \quad (2.2)$$

where P and C are the number of pages and the page capacity respectively. This measures the storage utilization of the multipaged relation.

The multipaging directory of the axis-attribute is usually small enough to be held in RAM. For a hypercubical attribute space, where the axis-attributes have the equal number of partitions, the directory size is $O(kp^{\frac{1}{k}})$. It is not difficult to appreciate that, for a good partition selection, the expected number of page accesses for exact-match query is π if an overflow tuple is retrieved with one extra page access. For partial-match query, where s of the k attributes are specified, the expected number of page accesses is $O(p^{1-\frac{s}{k}})$.

Details of the implementation procedure for static multipaging are given in [3]. The major drawback of the multipaging structure is that it does not admit insertions and deletions of tuples without occasional reorganization of the whole relation. To extend static multipaging into a dynamic structure, we observe the similarity of the page addressing function with the element addressing technique of multidimensional arrays. The technique of finding a satisfactory storage allocation scheme for extendible arrays discussed in next section is exploited to realize Dynamic Multipaging.

2.7 Dynamic Multipaging Structure

2.7.1 Dynamic Multipaging for Multidimensional Data Organization

Dynamic Multipaging [9] is superior to any known multidimensional data organization for uniformly distributed data. It is a hashing method for organizing multidimensional data. It uses a dynamic order preserving multi-attribute key-to-address

function to address its data pages. The key-to-address function is computed with the aid of a directory of size $O(kp^{\frac{1}{k}})$, where p is the number of data pages in the Dynamic Multipaging structure and k is the number of dimensions. The quantity $k p^{\frac{1}{k}}$ corresponds to the sum of linear dimensions of a k -dimensional hyper-rectangle (whose volume in this case is the number of data pages) and is usually small enough that the entire directory can fit into RAM. The quantity $p^{1-\frac{1}{k}}$ also appears as a term in the cost analysis of this paper. This corresponds to a surface plane of a hyper-rectangle of volume p . The Dynamic Multipaging scheme is a sequel to an earlier proposed structure called Multidimensional paging [3], in which an algorithm to partition data into a static multipaging structure is presented. The realization of multidimensional paging poses the same problem as the design of storage allocation scheme for extendible multidimensional array. A satisfactory storage scheme for such extendible arrays along with the page addressing technique is presented in [4]. An improvement to the directory structure of dynamic multipaging was proposed by [2] in which the overall storage requirement is reduced and the addressing technique they proposed is used in our design.

The use of linear scales does not explicitly represent the space partitioning and the very nature of linear scales making searching very inefficient. Thereby we replace the linear scales by binary tree structured scales. Binary tree structured scales explicitly represent the history of the partitioning and provide faster access to the data. Their use also reduces the overall directory size considerably. Simulations studies are carried out for measuring the performance of Dynamic Multipaging with binary tree structured scales and uniform data distribution.

2.7.2 Concepts of Dynamic Multipaging Structure

Dynamic Multipaging has been made possible by introducing additional fields in the entries of the static multipaging directory, and adopting a page addressing function H_d analogous to the element addressing function of the extensible array scheme described in [1]. The structure grows from an initial allocation of one page, page 0. Tuples get inserted until further insertion causes an overflow. Consequently, the

tuple is stored in an overflow page or a segment of the primary data pages is split so that the tuple may be stored in the structure but within some acceptable limits of either the load factor or the probe factor. The set of pages with a common range of attribute values on an axis is called a segment. A page then is a member of k segments since each page is uniquely defined by k attribute intervals, one from each attribute axis. The page which causes a splitting process to be initiated is called the *pivotal* page. Segment interval splitting is done by adding an additional segment and distributing the tuples of the segment corresponding to the pivotal page into the pivotal segment and the new segment created.

In the reverse process, deletion of tuples from pages may cause the count of the tuples in a segment to be well below what can be contained in it. In such a situation the segment may either be deleted if it belongs to the attribute which was most recently extended, or merged with a more dense adjacent segment and split again to distribute tuples evenly between them. For a hypercubical organization, since a segment holds $O(p^{1-\frac{1}{k}})$ pages, the file holding the relation expands and contracts by amount of $O(p^{1-\frac{1}{k}})$ pages.

As in static multipaging, measures of the performance of DMP are the probe factor π and the load factor α both of which we try to keep to 1.0. Obviously, splitting a segment whenever a page overflows leads to a low load factor, i.e., poor storage utilization. To control the rate of segment splitting, segment splitting is deferred by storing overflow tuples in a set of independently organized overflow pages. The collision resolution method is such that the presence or absence of a tuple is detected with the minimum number of overflow page accesses and further that the overflow area is as full as can be achieved. For instance if a page has z overflow tuples then these can be found in at most $\frac{z}{C'}$ pages where C' is the overflow page capacity. Since the number of overflow tuples are restrained and kept low by reassignment to the primary area after segment splits, overflow tuple retrieval can be achieved with one extra page access. Since overflow tuples are not resident on the primary data pages, the load factor as defined earlier may exceed 1.0 in this

case. The true storage utilization is defined for dynamic multipaging as

$$\mu = \frac{N}{P \times C + P' \times C'} \quad (2.3)$$

where P' is the number of overflow pages and C' is the overflow page capacity. We keep $C = C'$ in our design.

2.7.3 K-Dimensional Extendible Array (KDEA)

To appreciate the page addressing mechanism in Dynamic Multipaging we describe briefly a technique, called the Index-Array scheme, for storing a multidimensional array in linear consecutive storage locations. For simplicity we will describe a two-dimensional extendible array, $A[0 : U_1, 0 : U_2]$, where the upper bounds U_1 and U_2 are allowed to expand indefinitely. The method is easily generalized to any number of dimensions. The Index-Array scheme (IXA) uses "index arrays", B_1 , and B_2 to help calculate addresses of array $A[0 : U_1, 0 : U_2]$.

Figure 2.2: Schematic Layout of a 2DEA $A[0:3, 0:3]$

Figure 2.3: Array $A[0:3,0:3]$ extended to $A[0:3,0:4]$

Figure 2.1 shows the schematic layout of storage for the extendible array $A[0 : 3, 0 : 3]$. There are two index arrays, $B_1[0 : 3]$ and $B_2[0 : 3]$. Consider the addressing problem involved in Figure 2.2 when we add a column to the 4×4 2DEA. The elements of the existing portion of the array occupy $4 \times 4 = 16$ locations, addressed from 0 to 15. The new column must occupy 4 locations starting from address 16 since 4, the number of columns in the initial array, is implicitly given as the column index 4 of the new (fifth) column, we need only record the fact that there were 3 rows present when the column was added. This is done by storing the number 3, the number of rows of the existing array, in $B_2(4)$, as in figure 2.2. This gives a 4×5 array.

To calculate the address of an arbitrary element, say $A(1, 4)$, it is necessary first to determine whether this is a part of column 4 or part of row 1. This is easy to resolve by noting that the element always belongs to the row or column which gives the larger starting address value; i.e., column 4 with starting address 16. The actual

element location is found by adding the displacement, using row index 1, to this starting address. The element $A(1,4)$ then has the address $4 \times 4 + 1 = 21$.

The above technique generalizes immediately to k -dimensional arrays, except that instead of adding a row or a column, we add a sub-array of $k-1$ dimension. Entries in the arrays $B_i, i = 1, \dots, k$ must contain $k-1$ values of each other dimension in order to specify fully the size of the sub-array that is adjoined whenever an axis is expanded.

2.7.4 Page Address Calculation

The page addressing function in DMP is identical to element addressing in the IXA scheme for KDEA. To facilitate page address calculation and the detection of conditions necessitating segment interval splitting or the merging of two adjacent segments, the entries in the Dynamic Multipaging directories include other fields besides those of the entries in the static multipaging directories.

The i^{th} attribute directory Dir_i corresponds to the B_i array of the IXA scheme. Unlike the IXA, which has integer values, the directories have attribute values which are maintained in a natural ascending order. Associated with each attribute value in the directory is an integer index number assigned at the time the value is admitted. In terms of the order of the attribute values, the index numbers for the axis-attribute i , are a permutation of the numbers $0, 1, \dots, f_i - 1$. The fields of the entries in the attribute directory for axis i are:

$Dir_i(j).b$ The attribute value defining the upper bound of j^{th} segment interval in dimension i . Every attribute value is less than the high value denoted as ∞ .

$Dir_i(j).J$ The corresponding index number of the value b .

$Dir_i(j).n$ The number of tuples in the segment, (used during deletion)

$Dir_i(j).v$ The number of different values in the interval range.

$Dir_i(j).f_q, q = 1, \dots, i-1, i+1, \dots, k$ The number of segments on the other axis-attributes.

Given a tuple, how does the primary page address get computed ?. We note that the axis-attribute are given in a predefined order. Using the axis-attribute values, the corresponding index numbers are obtained and these numbers may now be perceived as the subscript of an element in a KDEA. The page address function is computed exactly as in the IXA scheme. Assume the interval indices of the attribute values are $j_1, \dots, j_k$ and $Dir_i, 1 \leq i \leq k$, are the directories. If we define $Dir_i(j_i).V$ as the field V of the entry $Dir_i(j_i)$ then the addressing function may be computed as shown below.

$$PGA = MAX + \sum_{r=1, r \neq t}^k (a_r \times Dir_r(j_r).J) \quad (2.4)$$

where $a_r = 1$ if $r = k$ else

$$a_r = \prod_{1 \leq q \leq r, q \neq t}^k Dir_t(j_t).f_q \quad (2.5)$$

and, $t =$ the index i which gives the maximum value of the product defined as

$$Dir_i(j_i).J \times \prod_{q=1, q \neq i}^k (Dir_i(j_i).f_q), 1 \leq i \leq k \quad (2.6)$$

and

$$MAX = Dir_t(j_t).J \times \prod_{q=1, q \neq t}^k (Dir_t(j_t).f_q) \quad (2.7)$$

Chapter 3

Dynamic Multipaging Structure Design

The growth of the structure differs in the implementation of the splitting policies. We begin with the discussion of the basic data structures followed by the page address calculation scheme and finally the three different splitting and merging policies used for physical database organization.

3.1 Basic Data Structures

3.1.1 Scales

The scales define a partition of the domain of each dimension. They enable access to the data pages by aiding the computation of their addresses based on the values in each dimension. In the original multipaging structure, d -independent one dimensional arrays called linear scales were used to implement the scale structures. The linear scales do not explicitly represent the history of the partitions, which is important to some operations such as merging of two data pages. Also the operations such as inserting and deleting and querying become inefficient as the database grows large in size.

Therefore, we use the so called Binary Tree Structured scales (BTS-scales). This technique was previously adopted in Ying Lin's [25] thesis. The BTS-scale consists of two types of nodes, an internal node and a leaf node. Each internal node stores a partitioning point representing a $(d-1)$ dimensional hyperplane that

cuts the space into two rectangular shaped regions. Each leaf node is associated with a d -dimensional slice $S(i_k, j)$, $0 \leq k < d$ of the space which is bounded by two neighboring partitioning hyperplanes, $0 \leq i_k \leq m_k$, where m_k is the number of slices in dimension k . The slice $S(i_k, j)$ is addressed by the index i_k stored in the corresponding leaf node.

In a 2-dimensional case, the BTS-scales could be shown as in Figure 3.1.

Figure 3.1: Binary Tree Structured Scales

The splitting policies (discussed later) determine when a slice $S(i_k, j)$ is split. The split results in replacing the leaf node i_k by a new internal node representing the partitioning point and then attaching two leaf nodes as children of the internal node. Additional modifications may be needed according to the different implementations of the splitting policies.

In the reverse action to splitting, when the directory (to be explained subsequently) needs to be merged, the internal node corresponding to the merged hyperplane is located according to the merging policy and its two children are deleted to create a single leaf node.

The two leaf nodes of an internal node represent two sides of a hyperplane.

Figure 3.2: Binary Tree Structured Scales

During split, the left child retains all properties of the original slice. The right child is created with some properties determined by different storage mapping functions. During merging, the two leaf nodes of the internal node are deleted to create a single leaf node with the properties of its left child.

There are significant advantages of using BTS-scales over linear scales proposed in the original paper [1].

1. Tree structures allow faster data search
2. BTS-scales explicitly represent the history of the structures growth
3. BTS-scales facilitate directory merge by maintaining a buddy relationship between page regions of two sides of a hyperplane.

Definition 3.1 A buddy of a data page B in dimension k ($0 \leq k < d$) is a data page C such that the corresponding page regions of these two data pages B and C were obtained by splitting a page region on dimension k in the course of partitioning the data page.

Example 3.1.1 In Figure 3.3, three data pages correspond to three pages b , b_1 and b_2 respectively. Page region b is represented by nodes (n_1, n_2) , page region b_1 is represented by nodes (n_0, n_2) and b_2 is represented by nodes (n_2, n_0) . The data pages corresponding to page regions b_1 and b have the buddy relations in dimension k_0 since their corresponding nodes n_2 and n_0 have the same parent. Similarly, the data pages corresponding to page regions b and b_2 have the buddy relation in dimension k_1 .

Figure 3.3: Buddy Relationship

Thus for a given data page B , k possible buddies can be located by finding siblings having the same parents as the k nodes in the k BTS-scales. If some sibling is an internal node the buddy in that dimension does not exist. Therefore, by using the BTS-scales, it is easy to find the buddies of a data page.

3.1.2 Directory Organization

The directory for dynamic multipaging structure is organized in the form of k binary trees. One for each dimension (axis). The binary tree for dimension i is represented by its root node $Root_i$. The directory for axis i has f_i entries, i.e., it consists of

p	k			
	2	5	10	20
10^2	20	13	16	25
10^5	632	50	32	36
10^8	2×10^4	199	63	50

Table 3.1: Multipaging Directory

$f_i - 1$ leaf nodes, index of which may vary from 0 to $f_i - 1$. A given value of an attribute corresponding to dimension i determines the leaf node where the attribute value falls and the index I_i in that leaf node. A page address is thus calculated by locating the k index numbers $I_1, I_2, \dots, I_k$ for dimensions $0, 1, \dots, k - 1$ respectively. Then a page addressing function $H_s(I_1, I_2, \dots, I_k)$ is used to calculate the actual page address. The directory thus maintains a dynamic correspondence between the page regions in the data space and the physical data pages stored on disk. The entire directory (d -binary trees) is small enough to be kept in the main memory. Thus any reorganization of the directory during split or merge is performed in the main memory. This is very effective as memory operations are fast compared to disk operations. The main memory capacity is the only limitation.

For a hypercubical attribute space, where the axis-attributes have equal number of partitions, the directory size is $O(kp^{\frac{1}{k}})$. Where k is the number of dimensions and p is the number of data pages. Table 3.1 gives some values of the directory size as a function of p and k .

3.1.3 Data Pages

The data pages are used to store the records of the database. We use sequential allocation of records within a page and records are inserted into a page in no specific order.

Since records stored in a page will later be retrieved during query operations, we

need to store a small amount of status information on the page, such as whether the page is empty, if not what is the occupancy number. This is perhaps the smallest amount of information that can be stored and that's what is exactly done in our design, consequently requiring lesser disk space than previous implementations of multi-dimensional storage structures. The set of pages with a common range of attribute values on an axis is called a segment since each page is uniquely defined by k attribute intervals, one from each attribute axis.

3.2 Page Address Calculation

We discuss the page addressing technique [1] used to calculate the page address in our extended dynamic multipaging design. The basic idea of page address calculation has already been discussed in the last chapter, however, we explain it here by means of an example.

In dynamic multipaging, access is through the spatial axes, which is possible because the pages are bounded by hyperplanes orthogonal to the axes. Thus we must split all pages bounded by a hyperplane if we wish to split one. So dynamic multipaging allows a file of p pages to grow by adding a segment of $O(p^{1-\frac{1}{k}})$ new pages, where d is the number of dimensions. Since we create a new segment all at once, its pages can have addresses which are numbered sequentially, and the segments can be identified by its lowest page address. The axial directory must hold, at least, this lowest page address and an appropriate value (say the lowest or the highest) of the corresponding attribute, for each segment orthogonal to the axis. Thus, the axial directory is normally used to return the address of a segment containing a given value of the attribute.

With this structure, the segment addresses need not be in the same order as the attribute values: the axial directory can permute the segments from their ordering, in the data space, by the values of the attribute. Thus, when we come to split segments in the middle of the data space, we can always consider that the new segment is created on the outer face of the hyper-rectangle, that is, at the face

which is furthest along the axis from the origin.

The multipaged hyper-rectangle thus grows dynamically always at its outer faces, and the problem of how the data space grows is equivalent to the problem of how to extend a d -dimensional array. For this, it is sufficient to have axial directories which record the lowest page address of each segment as it is created. Each segment is a $(d - 1)$ -dimensional array, and, in order to address pages within the segment, we can use the usual row-major or column-major addressing formulas. The segment dimensions are deduced from the lowest page addresses of the segments that preceded it chronologically in the growth of the file.

As an example, the following figure shows a 3-dimensional directory structure.

Figure 3.4: Extendible Array with Axial Directories

We illustrate the page address calculation of the page corresponding to the index $(i, j, k) = (1, 2, 1)$. The extendible array is assumed to have been built up as follows

1. The segment containing page 0 was added 1st giving a $1 \times 1 \times 1$ array
2. The segment containing page 1 was added 2nd giving a $1 \times 2 \times 1$ array
3. The segment containing pages 2,3 was added 3rd giving a $2 \times 2 \times 1$ array
4. The segment containing page 4,5 was added 4th giving a $3 \times 2 \times 1$ array

AXDIR (Table 1)

AXIS	INDEX	BASE ADDRESS
1	0	0
1	1	2
1	2	4
1	3	18
2	0	0
2	1	1
2	2	12
3	0	0
3	1	6

SEARCH (Table 2)

AXIS	INDEX
1	1
2	2
3	1

RELATION A

AXIS	LARGEST BASE ADDRESS
2	12

RELATIO B

AXIS FOR LARGEST BASE ADDRESS	AXIS	INDEX	SUBMAX	DIM
2	1	2	4	3
2	2	2	12	1
2	3	1	6	2

RELATION C

AXIS	DIM	CO	COEF	COEFF
1	3	3	3	1
3	2	6	1	3

Figure 3.5: Page Address Calculation Procedure

5. The segment containing page 6–11 was added 5th giving a $3 \times 2 \times 2$ array
6. The segment containing page 12–17 was added 6th giving a $3 \times 3 \times 2$ array
7. The segment containing page 18–23 was added 7th giving a $4 \times 3 \times 2$ array

The final result has three axial directories of lengths 4, 3 and 2 respectively, containing the base addresses of the seven segments (the first segment has base address 0, which is stored as the first element of all the three axial directories).

The relations AXDIR corresponding to the axial directories of the above figure are as shown in table 1. Table 2 shows search for the element $(i, j, k) = (1, 2, 1)$.

Tables A, B and C are generated from the steps used for page address calculation. The method of page address calculation corresponding to index $(i, j, k) = (1, 2, 1)$ proceeds as follows.

Step 1 is to compute the base address of the segment containing the page at position $(1, 2, 1)$. This is the largest of the three axial directory at $i = 1$, $j = 2$ and $k=3$, respectively. This is 12.

Step 2 is to calculate what the dimensions of the extendible array were when the segment at $j = 2$ (the 5th segment was formed. Say these dimensions are $d_i \times d_j \times d_k$). The dimensions of the segment are then $d_i \times 1 \times d_k$. The computation is done by finding, for each dimension, the largest entry less than or equal to 12 (found in the last step). This gives values of 4 for $i = 2$, 6 for $k = 1$ and, of course, 12 for $j = 2$. Since the indices are numbered from 0 onwards, the actual dimensions of the segments are 1 for j and 1 + the indices found for i and k : $3 \times 1 \times 2$.

Step 3 evaluates the multinomial which combines the dimensions and the searched indices to calculate the offset address within the segment. CO is the cumulative product of the dimensions, COEF is the same as the CO except for the last axis (k), where it is 1, and COEFF is the same as COEF, but cyclically permuted among the axes. Finally, offset is the multinomial $INDEX \times COEFF$ for all axes other than the one with maximum base address, i.e., only i and k and not j .

Step 4 finally adds the largest base address (12) corresponding to dimension j to the offset (4) to compute the page address as 16 (offset 4 within the segment based at page 12.)

3.3 Splitting and Merging Policies

We discuss three different splitting policies proposed in [1]. The splitting policies are designed so that the storage utilization is maximized and the probe factor is kept as low as possible. We compare the performance of the three splitting policies in chapter 5.

3.3.1 Splitting Policy 1

This is the simplest splitting policy that organizes the database fully in primary data pages. In other words, no overflow pages are maintained. Tuple insertion continues until the page where the new tuple is inserted overflows. On overflow, the segment containing the target page is split. The tuples in the segment to be split are equally distributed amongst the segment and the new segment to be added. The actual splitting of a segment involves selecting one of the dimensions to be split as the pivotal page is a member of k segments, one in each dimension. Since querying the database on one dimension is not at the cost of the other, we would like to maintain approximately equal size directories on each dimension. For this it is important to select different dimensions during subsequent splitting operation. We also have to ensure that the selected dimension to split also maximizes the load factor. Thus given a pivotal page, there are k possible attribute intervals, $I_i(j)$ denotes the j^{th} segment interval of the i^{th} axis-attribute. We choose and split a segment interval $I_i(j)$, so that $\frac{V_i}{f_i} \geq \frac{V_j}{f_j}$ for all $i = 1, 2, \dots, k$. The choice of dimension is justified in the following proposition. If more than one t value is found, we select one that maximizes the load factor. This proposition is the same as proposed in [1]. A detailed proof of the above proposition can be found in [1].

This splitting policy is simple and has a probe factor of 1, however suffers from low data page utilization. This stems from the fact that splitting a segment whenever a page overflows results in creation of several empty data pages, thereby reducing average data page utilization.

3.3.2 Splitting Policy 2

In this splitting policy, we do not split a segment whenever a page overflows. To control the rate of splitting, we defer splitting by storing overflow tuples in a set of independently organized overflow pages. Since overflow tuples are not resident on the primary data pages, the load factor may exceed 1.0 in this case. The true

storage utilization is

$$\eta = \frac{N}{P \times C + P' \times C'} \quad (3.1)$$

where P' is the number of overflow pages and C' is the overflow page capacity. We keep $C = C'$ in our design.

Every time a page overflows, we verify the resulting storage utilization. In case, the resulting storage utilization exceeds the required storage utilization, we split the segment. Else, we store the tuple in the corresponding overflow page. If the overflow page is already full, then splitting is carried out. The direct consequence of maintaining overflow pages is increased probe factor as an absence of a tuple in primary data page requires access to its corresponding overflow page. Also storage utilization increases as splitting occurs less frequently compared to splitting policy 1. For large database sizes, the average number of disk accesses for record insertion increases as the cost of a split increases with increase in database sizes.

3.3.3 Splitting Policy 3

Splitting policy 2 results in increased data page utilization but increases the probe factor, which is undesirable. This results from increased number of overflow tuples and so our objective is to minimize the number of overflow tuples. This is achieved by segment balancing whenever need for a split arises. i.e., before carrying out a split we explore the possibility of segment balancing but only if the resulting storage utilization is below the required storage utilization. If the resulting storage utilization is greater then we proceed with split.

Segment balancing is carried out as follows: Given a pivotal page, for each dimension $1, \dots, k$ we find the segment containing this page. We select the least loaded segment of these k segments. The next step is to locate the buddy (discussed in chapter 2) segment of the least loaded segment identified in previous step. We attempt to balance the tuples in these two segments. If balancing is successful, we attempt to insert the new tuple in a primary data page. Thus segment balancing results in deferred splitting and attempts to get the primary data pages as full

as possible by moving overflow tuples to primary data pages. Consequently the data page utilization increases with reduced probe factor. The only disadvantage of segment balancing is whenever segment balancing is not successful. Such a case occurs when either the number of tuples to be transferred to a page exceed twice its capacity or after segment balancing the new tuple cannot be accommodated in a primary data page and the overflow page. Thus the number of disk accesses used to read the buddy segment are wasted as split must to be carried out. These wasted disk accesses prove costly for large database sizes. For example, for a large database a buddy segment could contain as many as 50–100 pages. This accounts for 50–100 disk accesses wasted each time segment balancing fails. And also, segment balancing is no doubt costlier compared to a segment split because twice as many pages have to be read for segment balancing. However, we study the performance of the three splitting policies (splitting policy 1, splitting policy2 and splitting policy 3 discussed earlier) in chapter 5 and comment on the suitability of each.

3.3.4 Merging Policy

The counterpart of splitting is merging. Merging in general arises when a page becomes empty or the page utilization falls below the threshold due to tuple deletion. Merging is influenced by two factors. First, we must decide which segment (containing the page from which tuple was deleted) pairs are candidates for merging. This decision is based on a buddy system. In a *buddy system*, each segment, say X , can be merged with exactly one segment in any of the d dimensions, provided the resulting page region retains the shape of a multidimensional rectangular box. Ideally, the chosen segment, say B_j , should have the property that at some earlier point it was split to yield pages X and B_j . This buddy is called the *tree buddy*.

The second factor that influences the merging policy is the data page utilization threshold. A high data page utilization (close or greater than the threshold) does not favor merging as splitting would have to occur very soon again.

Given a page that initiated a merge operation. We evaluate the storage utilization that would result if we deleted a segment. If the resulting storage utilization is

close or more than the threshold, we delete the segment containing the page in the last dimension. Because that's the dimension to which the last segment was added and this also avoids the spatial deadlock problem as discussed above. In case the page utilization falls below the threshold we delete a segment as discussed in sequel.

We identify all the segments containing the page that initiated the merge operation and select the one that is the most heavily loaded. If this segment has a utilization less than the threshold we merge it with its buddy and split again into two thereby equally distributing the tuples between them.

3.3.5 Summary

Splitting and merging operations do not cause an entire directory reorganization in our design, thereby improving insertions and deletions performance. In most applications insertions and deletions are not as frequent as querying the database, and the use of BTS-scales greatly improves the performance as search operations are faster on BTS-scales than in linear scales. The structure keeps the principle of two disk accesses for processing exact match search. Different classes of queries are also processed with reasonable efficiency. Splitting policy 2 and 3 reduce the directory *fan out* by controlling the rate of structure expansion. This also increases the average data page utilization. Moreover, memory utilization is optimal as the BTS-scales representing the directory are significantly small and thus can always be kept in memory. This significantly speeds up the search operation and in turn improves the query response time.

Chapter 4

Visualization

4.1 Visualization Model

The goal in this chapter is to develop a visualization tool that allows the user to create a dynamic multipaging structure and visualize the insertion, deletion and query algorithms of dynamic multipaging. Our visualization model is based on a reference model for visualization of multi-dimensional data [40]. Figure 4.1 shows the high level structure of the model. The visualization interface and the dynamic multipaging structure are tightly coupled as the visualization interface permits retrieval and manipulation of the data in the structure.

The visualization system is the interface between the user and the database. All visualization related activities are propagated from the user to the visualization system through a user interface. The request is accomplished by accessing the necessary database classes and the resulting structure is visualized in a window which is a part of the visualization system.

The goal of the visualization can be categorized as:

- How the dynamic multipaging structure grows and shrinks in response to inserting and deleting records ?
- How the directory structure assists in locating pages where the records to be inserted or deleted are located ?

Figure 4.1: Visualization Model

- How the splitting policy and merging policy that create the structure work ?

The above three categories of visualization also cover the visualization for processing an exact match query. Essentially, servicing an exact match query request is the same as locating a record that satisfies the exact match query condition. This behavior can be seen during record insertion and record deletion, as the directory structure (binary trees in all dimensions) has to be traversed to locate the page where the record is to be inserted or from where the record is to be deleted.

4.2 Visualization Technique

There exist several visualization tools: VisDB system [33], Navigational View Builder [36], Tioga-2 [37] and visualization techniques used by these tools: Parallel Axis Technique [39], Use of cons or glyphs [40], linked display [45], pixel-per-value [33, 34]

All these tools have to be integrated with the user developed applications to visualize the application's data. Most visualization tools expect, directly as input, the data that is to be visualized. They do not handle visualization of external data structures that are used to index the data. Visualization of our application data

requires not only graphical representation of the data but also the visualization of the data structure (binary trees) used as scales. One of the major requirements of our visualization process is the support for simultaneous visualizations, such as, visualizing the traversals of the two binary trees representing the scales in two dimensions. This requires the tight integration of the visualization system with the application that maintains the directory structure of the dynamic multipaging structure and the binary tree structure scales used to index directory structure. Using any existing visualization tool would require us to integrate the application with the visualization tool to take care of in particular the visualization of the binary tree structure scales. Some visualization tools are static in terms of operations. That is, they support naive graph drawing capabilities such as, graphical representation of a set of data provided as input. For instance, representing the records of a database as points in 2 or more dimensions. Some have enhanced capabilities, such as support for drawing 2 and 3 dimensional objects and performing certain operations such as rotation, providing different views of an object from different angles etc. Some of these tools are being used for visualizing data of real world applications. They do not suffice for our visualization process. Some tools provide facilities that are too naive while others are too sophisticated to be used for our visualization process. Moreover, none of the tools allow customization for visualization of a user developed storage structure (dynamic multipaging in our case) for the physical organization of a database.

Having explored the features of existing visualization tools and the requirements for visualizing our application, we decided to use Java [32] for implementing our visualization system. The reason for using Java are multi-fold. Java is an unified programming environment providing 1) the conventional programming constructs 2) support for sophisticated graph drawing and 3) support for multi-threaded programming. The use of threads greatly simplifies the process of simultaneous visualization. For example, we can instantiate multiple threads, each handling visualization of the traversal of a binary tree. The availability of conventional programming constructs enables us to implement our dynamic multipaging structure also in Java. This sim-

plifies the integration of the visualization system with the structure being visualized. For our visualization we have used the pixel-oriented technique for representing the database in a two dimensional space. The pixel oriented technique [33] is based on the idea of representing a record as a point in a two dimensional space. Each record is uniquely identified by its two attributes on the two dimensions and uses the screen pixels to represent a record as a point.

4.3 Implementational Details

To implement our visualization system and the storage structure, we outline the implementational details of the visualization system and the storage structure.

The two major components of our application are: 1) dynamic multipaging structure and, 2) visualization system. The user interacts with the visualization system through the graphical user interface developed using Java Workshop [48, 49]. Java Workshop is a software developed in Java for developing Java applications. Amongst the several resources Java workshop provides, we use it's visual tool [49] for developing the user interface. The user interface is divided into four sections. One section, publishes an interface for allowing the user to perform insertion and deletion on the storage structure. During insertion and deletion the process of locating the binary tree leaf nodes corresponding to the record being searched is visualized in two windows displaying the binary tree scales. The visualization includes traversal of binary tree in each dimension. The results of insertion and deletion are visualized in another section of a window displaying the storage structure. The components of the user interface such as, text fields, buttons and graph drawing area are developed using the drag and drop feature for building a user interface. The (AWT) Abstract Windowing Toolkit [31] classes are used for drawing the directory structure and the binary trees representing the scales.

Figure 4.2 shows the coding files hierarchy for the entire application and figure 4.3 shows the different components of the application and how they interact for accomplishing the visualization. The files containing the code for each component

DynamicMultipagingApplication	[Parent Package]
DynamicMultipagingMain.java	[Parent class for running the Application]
DynamicMultipaging.java	[Representing the Multipaging Structure]
DynamicMultipaging.gui	[Code for the User Interface]
DrawStructure.java	[Thread class for Performing Animation]
DynamicMultipagingRoot.java	Gui Code
DatabasePackage	[Database Package]
Database.java	[Code for database growth and shrinkage]
Page.java	[Code for handling pages on disk]
Record.java	[Representing a Record structure]
Constants.java	[Code for database growth and shrinkage]
BTSPackage	[Package for handling the BTS-scales]
Binary_tree.java	[Representing a Binary Tree]
Binary_tree_node.java	[Represents a Binary Tree Node]
InternalNode.java	[Represents an Internal Node structure]
LeafNode.java	[Represents a LeafNode structure]

Figure 4.2: Code Organization

namely, the structure, visualization and binary tree scales are also shown.

The entire application is developed as a Package [32] in Java. Each *.java* files shown above is compiled into a platform-independent executable form called *.class* file. The *.class* file corresponding to file is "DynamicMultipagingMain.java" is the entry point for the java application. DynamicMultipagingApplication is the parent package containing two sub packages called DatabasePackage and BTSPackage (Binary Tree Structured) that contain the structure and binary tree structure scales files respectively. A special file called "DrawStructure.java" that subclasses the Thread class of Java is used for performing simultaneous visualization of the traversals of the binary tree during search of a record for accomplishing insertion and deletion. The directory structure is implemented in the file "DynamicMultipaging.java". This file implements the three splitting policies and merging policy for controlling how the structure grows and shrinks in response to insertion and deletions. The user interface code is contained in file "DynamicMultipaging.gui". All requests generated by the user through the user interface are handled by "DynamicMultipaging.java". The

Figure 4.3: Application Structure

DatabasePackage is implemented using the three files "Database.java", "Page.java" and "Record.java". The database is a collection of pages and a page is a collection of records. The "Page.java" file provides methods for reading and writing records on to the physical disk. The "Record.java" file represents the attributes of the record under consideration. And finally the BTSPackage contains all the files necessary for creating and maintaining binary tree structure scales in each dimension. The "Binary_tree.java" file contains code for generating a binary tree using the code in files "InternalNode.java" and "LeafNode.java" for generating the internal and leaf nodes of a binary tree. The structures for internal node and leaf node are defined in the respective files. All these files are grouped together under the parent package "DynamicMultipagingApplication". The file "DynamicMultipagingMain.java" contains the code for starting the application as an applet or an application. To start the application one has to type "java DynamicMultipagingMain". Any java application can use this application by merely importing (equivalent of #include in C) the parent package "DynamicMultipagingApplication".

4.4 Visualization Set Up

This section represents snap shots of the graphical user interface that has been developed using Java Workshop [48]. Figure 4.4 shows the graphical user interface that allows a user to specify all the necessary parameters for creating a dynamic multipaging structure. The user can specify the number of records for which he or she wants to generate the structure. The Dimensions field is used for the number of attributes the records of the database has. The distribution (uniform and non-uniform) parameter enables the user to specify the distribution of the data to be generated for the attributes of the records. A user can supply a datafile if the data is to be taken from a file. The Generate button is to start the generation of the structure. The three fields AVG. DISK READS, AVG. DISK WRITES and AVG. DATA PAGE UTILIZATION get filled with the respective values once the generation of the structure for the specified number of records is complete. The field

NUMBER OF PAGES gets filled with the number of data pages generated for the structure.

Figure 4.5 shows a snapshot for querying the database. The different parameters a user could specify for querying a database is the class of query which could be an exact match, partial match, range or a partial range. The above structure specifies a RANGE query. The number of dimensions to use for the class of query specified. The interface volume allows a user to specify whether to generate a query with low, medium or high range. This is applicable only for range and partial range queries. The field HOW MANY QUERIES lets a user specify over how many queries to take the average result for. The only result being calculated here is the the average efficiency of the query specified.

The final section represents the interface for visualization. As shown in figure 4.6 the interface consists of a region for displaying the structure of the directory storing the records. Each point in 2-dimensional space represents a record. The binary tree structure scales for each axis are represented too. The Database interface allows a user to manually create a structure by specifying the attributes of the record. However, this is a sample user interface for viewing data only in a 2 dimension, even though the third attribute corresponding to FEMALES is stored in the structure, only the first two attributes are used for the visualization purpose.

Figure 4.4: Graphical User Interface

A screenshot of a "Query User Interface" form. The form is titled "Query User Interface" and contains several input fields and buttons. The fields are labeled "Query Name", "Query Type", "Query Status", "Query Date", "Query Time", "Query Location", "Query Category", "Query Subcategory", "Query Description", "Query Keywords", "Query Tags", "Query Author", "Query Editor", "Query Reviewer", "Query Approver", "Query Status", "Query Date", "Query Time", "Query Location", "Query Category", "Query Subcategory", "Query Description", "Query Keywords", "Query Tags", "Query Author", "Query Editor", "Query Reviewer", "Query Approver". The "Query Name" field contains the text "Query Name". The "Query Type" field contains the text "Query Type". The "Query Status" field contains the text "Query Status". The "Query Date" field contains the text "Query Date". The "Query Time" field contains the text "Query Time". The "Query Location" field contains the text "Query Location". The "Query Category" field contains the text "Query Category". The "Query Subcategory" field contains the text "Query Subcategory". The "Query Description" field contains the text "Query Description". The "Query Keywords" field contains the text "Query Keywords". The "Query Tags" field contains the text "Query Tags". The "Query Author" field contains the text "Query Author". The "Query Editor" field contains the text "Query Editor". The "Query Reviewer" field contains the text "Query Reviewer". The "Query Approver" field contains the text "Query Approver". There are two buttons: "Query" and "Cancel".

Query Name	Query Type	Query Status	Query Date	Query Time	Query Location	Query Category	Query Subcategory	Query Description	Query Keywords	Query Tags	Query Author	Query Editor	Query Reviewer	Query Approver
Query Name	Query Type	Query Status	Query Date	Query Time	Query Location	Query Category	Query Subcategory	Query Description	Query Keywords	Query Tags	Query Author	Query Editor	Query Reviewer	Query Approver

Figure 4.5: Query User Interface

Figure 4.6: Visualization Interface

4.5 Two (2) Dimensional Visualization

For GUI design, Java provides a set of classes [31] which are a part of the Java package [32] called Abstract Windowing Toolkit (AWT). The classes provide the basic constructs for developing sophisticated user interfaces and drawing. The user interface consists of four components, one, for allowing the user to enter the necessary information for developing the structure, another, for the actual display of the dynamic multipaging structure and the remaining two for displaying the directory structure (binary trees) one for each dimension.

The two binary trees constitute the directory structure. Each internal node (circular) represents a boundary of a hyper plane, partitioning the structure in two distinct data space regions. The leaf nodes represent a segment and store all the necessary information for indexing the segment. Each grid cell in the structure represents a page where actual records are stored. A point in a grid cell represents a record. A page is thus a member of two segments, i.e., a page is identified by the intersection of two segments, one in each dimension.

4.5.1 Example of Directory Expansion

As discussed before, directory expansion occurs when we try to insert a record into the structure and the target page is full. This initiates a segment split and the page that caused this split is called the pivotal page. Segment split causes directory expansion. Assuming the initial situation of the structure is shown in figure 4.7 with a page capacity of 3. An insertion is attempted in the page identified in the right most corner of the directory structure.

Figure 4.8 shows the resulting structure after record insertion. The segment containing the target page is identified by the two binary trees in each dimension. The tuple count in the target page is increased by 1 and the number of records represented by 2 points. In this case the directory growth is just the increase in the number of tuples in the leaf node corresponding to the segment of the target page in dimension 1 (horizontal dimension).

Figure 4.7: Initial Situation of the Structure

Figure 4.8: Result of Directory Expansion

4.5.2 Example of Directory Contraction

Directory contraction occurs when we try to delete a record from the structure and the target page becomes empty due to that record deletion. Consequently, we attempt to merge the segment containing this page with a buddy segment (discussed in chapter 2), if one exists. Assuming figure 4.9 is the initial situation of the structure. And we attempt to delete a record from the topmost page on the left.

Figure 4.10 shows the resulting structure and the modified directory structure. This page contains only a single record so when we delete a record from it, it underflows, causing a merge. The merge dimension selected (based on merging policy) is y-axis. The siblings of the internal node containing the value 65 are neighboring segments and all pages identified by these two siblings are merged.

Figure 4.9: Initial Situation of the Structure

Figure 4.10: Result of Directory Contraction

Figure 4.11: Directory Structure for a Large Dataset

4.6 Extension to Visualization of Large Datasets

The major limitation of visualization system implemented is that it does not handle visualization of large datasets. In other words a user can visualize the working of dynamic multipaging algorithms for a fairly small database. From a practical viewpoint what is needed is a mechanism for visualizing large databases providing a bigger picture of the potential of the the visualization system. One of the main problem of visualizing large dataset is the size of the screen used to display the structure and the binary tree structure scales. As the structure grows in size so does the width and height of the binary tree structure scales. The size of the directory structure and binary trees are limited by the screen size. We propose a multi layered window architecture for visualizing large datasets. One window can be reserved for displaying the entire directory structure without showing the binary tree structure scales. The reason why the binary tree structure scales cannot be shown is the maximum depth and width of the binary tree for a large dataset will be too big to fit on the screen. For instance, the directory structure could be as densely populated as shown in figure 4.11 for say 1000 records.

Representing the entire binary tree structure scales corresponding to x and y axis

is impossible on any screen size. Thus even though the directory structure represents the entire database in a single window, the structure is incomplete because of the absence of the binary tree structure scales. The absence of binary tree structure scales does not convey the history of the partitions and the data represented is a mere graphical representation. The approach we propose is to allow a user to select a smaller rectangular region from the entire directory structure. The selected rectangular region can then be magnified to fit in the screen along with the binary tree structure scales for each dimension. The user can select a different region which can also be magnified and displayed to the user with its directory structure and the binary tree structure scales. The maximum number of such windows that can be displayed is limited by the screen size. However, windows representing data of different regions can overlap and the user can select the window he or she wishes to see and that window can be brought to the front of the screen and thus become the active window. The advantage of having a single window representing the entire directory structure is that the results of different classes of queries can be visualized by highlighting the pixels that correspond to the records matching any query. The pixel-oriented technique for displaying the database records is useful for displaying the results of different classes of queries. The maximum size of a database (number of records, each represented as a point) is limited only by the number of pixels that can be displayed in a window of specific size. Figure 4.12 is an example of using the technique discussed above for visualization of large datasets.

4.7 Visualization in Higher Dimensions

The visualization system implemented and the extensions discussed are primarily for visualization of dynamic multipaging algorithms in a 2 dimensional space. Higher dimensional visualizations would certainly be helpful in further understanding of multidimensional storage structures but require additional programming efforts. 3 dimensional techniques for visualization require drawing of three dimensional objects and tools for performing operations such as rotation, magnification and viewing

Figure 4.12: Example of Visualization of Large Dataset in 2 Dimension

objects with different perspectives or angles. Accomplishing all these operations in Java is difficult from a programming point of view because currently Java does not provide functional constructs for handling sophisticated operations on 3 dimensional objects. It supports simple 3 dimensional graph drawing capabilities. The use of VRML [50] has gained a lot of attention and importance in the visualization of different applications. VRML provides powerful constructs for drawing three dimensional objects and performing fairly sophisticated operations on them. The other benefit of using VRML for applications developed in Java is integration of Java applications and VRML is not difficult. VRML provides simple programming language constructs that can handle java applications.

Chapter 5

Simulation Experiments

A second objective of this thesis is to measure some performance characteristics of dynamic multipaging. The performance of a file system is determined by two criteria: response time to access requests and storage utilization. Response time consists of the processing time in memory and disk access time. Since disk access time is much greater than the processing time in memory, in most applications, the major components of this response time is the time spent in accessing disk. The maximal amount of data transferred in one disk access is fixed (a disk block or page). We will therefore measure the request efficiency in terms of the number of disk accesses per request. The storage utilization consists of the utilization of the datafile associated with the data pages.

A theoretical analysis of dynamic multipaging is difficult for two reasons: many of the techniques developed for analyzing single-key data structures do not directly generalize to their multidimensional counterparts, and dynamic multipaging has parameters that are complicated to capture in a mathematical model (e.g., the different splitting and merging policies). For these reasons we resort to simulation to study the behavior of our implementation.

5.1 Objectives of the Simulation

The objective of simulation is to study the performance characteristics of multidimensional storage structures such as storage utilization and the efficiency with which different classes of queries are processed. In our experimental set-up we consider only the case of a growing file, i.e., we consider the above parameters during the insertion of a record. However, using "inverse of splitting" as our merging policy, during deletions also the storage utilization does not fall too low. When creating the structure from an initial empty file, we observe the following parameters:

U_p the average utilization of a data page

f_{out} fan out of the directory

d_{access} average number of disk accesses required for a record insertion

p_{fact} the number of disk accesses required to process an exact match query

efficiency the efficiency with which the structure processes partial match, range and partial range queries

The average storage utilization of a data page is defined by $U_p = \frac{N}{P \times C}$, where N is the total number of records in the file and P and C are the total number of pages and the page capacity respectively.

The *fan out* of the directory is defined as the number of entries required to address the entire data page region. We measure *fan out* as the number of leaf nodes (L) required in all dimensions to address the entire data page region of (P) pages.

The number of disk accesses for a record insertion consists of two parts: one for searching the data page where the record is supposed to be inserted. The other for actually inserting the record into the data page. The former part is equivalent to an exact match search and requires at the most two disk accesses (if overflow pages are maintained) and the latter part depends on the splitting policy. Typically, the number of disk accesses depends on the following aspects of the splitting policy.

- When an overflow occurs we attempt to insert the overflow tuple in an overflow page. However, if a page overflows, then splitting would occur and incur twice as many reads as write operation because a split requires reading of a segment and writing of two segments. The records to be split are read from a segment containing the overflowed page in one of the dimensions selected by the corresponding splitting policy. These records are then distributed among the segment just read and the new segment created.
- In the case of an overflow page, tuples insertion is attempted into the overflow page. However, if overflow page overflows or the storage utilization increases beyond the threshold, we attempt to split as above. A segment balancing could occur e.g., as in splitting policy 3. Segment balancing incurs a huge number of disk accesses as two segments have to be read as well as written.

The simulation runs have the following objectives: to measure the efficiency of exact match, partial match, range and partial range queries. We measure the number of disk accesses for an exact match and term the measurement as probe factor (p_{fact}). For range, partial match and partial range, we measure the efficiency defined as the ratio of the number of records found N_f satisfying the query and the total number of records N , actually retrieved. That is, $efficiency = \frac{N_f}{N}$.

5.2 Simulation Model

Here we study the performance of three splitting policies for dynamic multipaging in 2 and 3 dimensions with uniform data distribution. We represent the dimensions by d and page capacity by c and study performance of following two models.

$c=18, d=2$: with keys uniformly distributed in a 2-dimensional data space;

$c=18, d=3$: with keys uniformly distributed in a 3-dimensional data space;

$c=18, d=2$: with keys non-uniformly distributed in a 2-dimensional data space;

$c=18, d=3$: with keys non-uniformly distributed in a 3-dimensional data space;

The purpose of selecting a page size of 18 records is based on the disk block size. The idea is to keep the page size (*number of records* $\times$ *record size*) close to the disk block size. Thereby enabling reading or writing an entire page to and from disk in one disk access during an insertion or deletion of a record. A page capacity of 18 records for a record of size approximately 24 bytes is ideal for a disk block size of 512 bytes. The total requirement is thus $18 \times 24 = 432$ bytes. We also maintain some additional information in each page such as count of duplicate tuples, existence of an overflow page etc. Taking into account the additional space required for storing such information a page capacity of 18 records is ideal.

Query performance experiments are also carried out for different classes of queries, namely, exact-match, partial match, range and partial range queries. The query performance models are as shown:

EQ : 2000 exact match queries;

PMQ : 200 partial match queries for $d=1$ and $d=2$;

RQ : 20 range queries with low, medium and high volume;

PRQ : 20 partial range queries with low, medium and high volume for $d=1$ and $d=2$;

The exact match query is performed by randomly generating 2000 records in an indicated data space. We perform the partial match query by generating random data only for the selected dimensions.

Range query is performed by randomly generating 20 queries in a given ranges. Similarly for partial range queries, we randomly generate 20 queries for the given ranges but only for a subset of the dimensions of the structure.

The value of 2000 selected for exact match query is high because we want to be able to average the results of query performance parameters. Since exact match queries are generated randomly the probability of finding a record satisfying this

query is low. Taking performance results by generating high number of exact match queries and taking the average guarantees accurate performance results. The corresponding value 200 for a partial match query is lower because large number of queries would be satisfied and since partial match queries are more time consuming than an exact match query we average the results over a smaller set of queries. A low value 20 for range and partial range queries is justified by the fact that these classes of queries are very likely to return large number of records satisfying the query and since these queries are time consuming we average the performance results over a very small set of queries.

5.3 Simulation Results

5.3.1 Utilization of data pages

A factor that controls the rate of splitting is the page capacity. The higher the page capacity the slower the splitting. However, very high page capacity results in higher disk accesses for record insertion or retrieval. This is because in one disk access only a single disk block can be read which is physically limited in size. An ideal situation will be to keep the page capacity close to the size of a disk block. That is,

$$diskblocksize(inbytes) = pagecapacity * recordsize(inbytes) \quad (5.1)$$

Since we know the size of a record in our implementation, we calculate the page capacity using the above formula and keep the page capacity as 18. This guarantees that at no time more than one disk access is required to read either a primary or an overflow page.

A high data page utilization is desirable but very high data page utilization is difficult to achieve. The splitting policies 1, 2 and 3 show increase in data page utilization in that order. We keep a threshold of around 0.75 and are able to achieve 0.70 in splitting policy 3. These results are, for uniform data distribution. Dynamic multipaging is not suitable for non-uniform data distribution. This was established by running test cases on data generated from a real world application. The data

is certainly not distributed uniformly. In the worst case it is non-uniform. Test cases were run to measure the storage utilization of dynamic multipaging with this set of data in 2 and 3 dimensions. The reason we selected to measure the data page utilization is because data page utilization is the most important parameter from performance point of view. Several parameters such as number of total pages generated for a fixed number of records and the average number of disk accesses required for a record insertion and query processing efficiency depends on the data page utilization. A good storage utilization is necessary for performance of other parameters. Figure 5.2 shows the data page utilization for the real world application data. We observe that the storage utilization is very low with splitting policy 1 and increases with splitting policy 2 and 3 but is still very low compared to the results obtained for uniform data distribution. The fact is attributed to how dynamic multipaging structure adapts to the data distribution. The splitting of the structure is more frequent with the real world application data and hence the structure grows much faster in terms of number of pages. Increased splitting leads to increase data pages for same number of records and hence fairly low data page utilization. The low storage utilization with the real world application data that is in the worst case non-uniformly distributed suggests that dynamic multipaging is not suitable for non-uniform data distribution. Conducting further experiments for measuring performance of other parameters for the above set of data would only be to support the low storage utilization result.

The main idea behind increasing data page utilization is to prevent frequent splitting due to record insertion as discussed in chapter 3. Splitting policy 3 defers splitting by performing segment balancing. However, the results indicate that the gain in terms of data page utilization is not very significant between splitting policy 2 and splitting policy 3. We attempted to identify the cause for the same. It seems that segment balancing is not always successful. Since segment balancing requires that a pair of data pages that are to be merged must always have tuples less than or equal to the capacity of a single data page. This must hold for all data page pairs of the two segments that are to be merged. Even if one pair of data pages does not

Figure 5.1: Data Page Utilization for uniform data in 2 and 3 Dimensions

satisfy this condition, segment balancing is not possible.

The results of data page utilization for all the three splitting policies for 2 and 3 dimension are as shown in figure 5.1. Figure 5.2 shows data page utilization of all 3 policies in both 2 and 3 dimensions. The results clearly indicate the splitting policy 3 in dimension 3 has the highest data page utilization.

As discussed before, the data page utilization is defined as

$$U_p = \frac{N}{P \times C} \quad (5.2)$$

for splitting policy 1 and

$$U_p = \frac{N}{(P + P') \times C} \quad (5.3)$$

where P' is the number of overflow pages and C is the page capacity for primary as well as overflow pages.

For a fixed number of records and page capacity the only parameter that actually decides the data page utilization is the total number of pages (P). The data page utilization in splitting policy 2 and 3 are higher compared to splitting policy 1 because of less frequent splitting. This is indicated by the fact that splitting policy 2 & 3 generate less number of pages when compared to splitting policy 1. This is

Figure 5.2: Data Page Utilization for uniform data in 2 and 3 Dimensions for Real Data

true because in splitting policy 2, when an overflow occurs, we do not immediately split the segment containing the target page. We instead store the overflow tuple in an overflow page. There exists only one overflow page or a chain of overflow pages. Consequently only an extra page is added in response to an overflow as compared to addition of an entire segment in case of splitting policy 1. This technique essentially accounts for deferred splitting thereby allowing more records to be added in comparison to splitting policy 1 before a splitting of a segment occurs. The direct consequence is increased data page utilization. That is, the total number of primary and overflow pages generated in splitting policy 2 and 3 are less than the total number of primary pages generated in splitting policy 1. The results shown in figure 5.3 and figure 5.4 establish this fact. The result for splitting policy 3 is better than splitting policy 2 because in splitting policy 3 splitting is deferred further. This is achieved by so called segment balancing whenever the need for splitting arises. In splitting policy 2 when an overflow page overflows we have to split. In splitting policy 3 we defer splitting by balancing the target segment and its neighboring segment. The idea is to accommodate the new tuple by creating space for it in the

Figure 5.3: Total number of pages for 2 and 3 Dimensions

overflow page that had over-flown due to it. Segment balancing would create space for the new tuple thereby avoiding need for splitting. The good storage utilization is explained by the fact that the increase in the number of pages is linear with the increase in the database size.

Table 5.1 shows the results for real data using splitting policy 1. The results indicate that splitting policy is not suitable for real data. The data page utilization is extremely low and splitting policy 1 incurs high number of disk operations even for a very small database. The results have been taken only for the first 1000 records and since splitting policy 1 performs very badly even for small database sizes we take performance results for larger database sizes using splitting policy 2 and 3.

Figure 5.4: Total number of pages for 2 and 3 Dimensions for Real Data

5.3.2 Directory Fan Out

The directory fan out refers to the amount of information required to address the actual data space. The directory is organized in the form of d binary trees. The fan out is defined as the number of indices required in all dimensions to address the data page region. The indices are basically the leaf nodes of the binary trees in all the dimensions. Figures 5.5 and 5.6 show the number of indices required for a fixed number of records in each of the three splitting policies. Splitting policy 3 gives the lowest fan out which is desirable, however, splitting policy 2 also renders close results. The low fanout with splitting policy 2 and 3 is a direct consequence of deferred splitting. Fan out is low for real data in both dimension 2 and 3 and the reason is there may be several duplicate records for which no storage is used, thereby no extra pages need to be added to the directory structure. We maintain duplicate records by just incrementing their count.

No. of Records	Avg. Disk Reads	Avg. Disk Writes	Data Page Utilization	Total No. of Pages
100	5.9	10.8	0.01	49
200	21.9	42.8	0.03	418
300	25.4	49.8	0.02	732
400	19.3	37.6	0.03	732
500	15.64	30.3	0.04	732
1000	11.44	21.8	0.05	1043

Table 5.1: Performance Of Splitting Policy 1 for Data not Distributed Uniformly in 2 Dimensions

No. of Records	Avg. Disk Reads	Avg. Disk Writes	Data Page Utilization	Total No. of Pages
100	10.8	20.6	0.06	98
200	74.51	148.02	0.008	1470
300	50.0	99.0	0.01	1380
400	37.76	74.5	0.015	1470
500	28.14	55.28	0.02	1357

Table 5.2: Performance Of Splitting Policy 1 for Data not Distributed Uniformly in 3 Dimensions

(a) 2D

(b) 3D

Figure 5.5: Fan out for 2 and 3 dimensions

(a) 2D

(b) 3D

Figure 5.6: Fan out for 2 and 3 dimensions for Real Data

5.3.3 Probe Factor

The probe factor is defined as the average number of page accesses required for retrieving a record satisfying an exact match query. Since every record is uniquely identified by its d attributes. The d attributes identify the page address where the record is supposed to be located. Thereby requiring one page access in case of splitting policy 1. Since we also maintain overflow pages in splitting policy 2 and 3, at the most one extra page access may be required in order to search an overflow page. The advantage of maintaining overflow pages is increase in the data page utilization, however, at the cost of increased probe factor. We attempt to reduce the probe factor by reducing the number of overflow tuple count in splitting policy 3 by performing segment balancing whenever possible. This stems from the fact that during record insertion there are situations where some segments are overloaded and others lightly loaded. We attempt the balance of two such segments by redistributing tuples between them and moving overflow tuples of a page to its primary page. The idea is to get the primary data page area as full as possible before storing tuples in overflow pages. Segment balancing can be considered as extended deferred splitting. Consequently, the data page utilization in splitting policy 3 also increases. The following figure shows how the probe factor varies between the three different splitting policies. The results clearly indicate that splitting policy 3 gives lower probe factor than splitting policy 2.

Figures 5.7 and 5.8 indicate that probe factor is low for splitting policy 3 compared to splitting policy 2. As discussed above this is due to reduced overflow tuple count with splitting policy 3. Figure 5.9 and 5.10 clearly show the reduced overflow tuple count for splitting policy 3.

(a) 2D

(b) 3D

Figure 5.7: Probe factor for 2 and 3 Dimensions

(a) 2D

(b) 3D

Figure 5.8: Probe factor for 2 and 3 Dimensions for Real Data

Figure 5.9: Number of overflow records for 2 and 3 Dimensions

5.3.4 Disk Accesses For Record Insertion

One of the main factors that greatly affect the performance of a multidimensional storage structure is the average number of disk accesses required for a record insertion. The greater the number of disk records the greater the cost of creating the structure. Basically it is the splitting policy and the merging policy that are responsible for the number of disk accesses for record insertion and deletion respectively. We study the effect of the three splitting policies on the creation of dynamic multipaging structure.

Figure 5.11 and 5.12 show how the average number of disk accesses for record insertion vary from splitting policy 1 to splitting policy 3. The observation we made for data page utilization that segment balancing is not always successful in splitting policy 3 is observed in this result too. The curve for splitting policy 3 rises almost linearly for record insertion but with an insignificant increase in data page utilization. For the same reason as discussed for data page utilization, that, the number of disk access required for performing segment balancing are wasted whenever segment balancing is not successful.

However, for splitting policy 1 and 2 the number of disk accesses are very close

(a) 2D

(b) 3D

Figure 5.10: Number of overflow records for 2 and 3 Dimensions for Real Data

(a) 2D

(b) 3D

Figure 5.11: Average number of disk accesses for 2 and 3 Dimensions

Figure 5.12: Average number of disk accesses for 2 and 3 Dimensions for Real Data

and almost constant for 2 as well as 3 dimensions. Even though, splitting is more frequent in splitting policy 1 than in 2, but splitting policy 2 incurs additional disk accesses during splits as overflow pages have also to be read and written. In summary, splitting policy 3 is very costly for creating higher dimensional dynamic multipaging structure. This is obvious from the 3-dimensional results, where the average disk access for record insertion is around 25 for 10,000 records. Since disk access cost a lot in time, we propose the use of splitting policy 2. However, we are yet to study the query performance of the three policies. For real data splitting policy 3 incurs high number of disk accesses because the data is not uniformly distributed causing splitting and merging to be relatively costly.

5.3.5 Query Processing Strategy and Performance

A query can be expressed by specifying a region R called the query region in a multidimensional space. It is convenient to think of a region as a cross product of intervals $[l_i, u_i]$ ($0 \leq i < d$). An exact-match query requests the retrieval of a point whose coordinates k_i ($0 \leq i < d$) match some d specified values x_i . A partial-

match query specifies $x_i (i \in Q, Q \subset 0, 1, \dots, d-1)$ and requests the retrieval of all points that match the specified values in the specified axes in Q . A range query requests the retrieval of all points whose coordinate values lie in the intervals $[l_i, u_i]$ ($0 \leq i < d$). A partial range query specifies $x_i (i \in Q, Q \subset 0, 1, \dots, d-1)$ and requests the retrieval of all points whose coordinate values lie in the intervals $[l_i, u_i]$ ($i \in Q, Q \subset 0, 1, \dots, d-1$).

Given uniform data distribution, it is easy to see that the proposed data organization for efficient query processing essentially partition the attribute space rectilinearly into rectangular regions or cells. The number of points in a rectangular cell is no more than b , where b is a predefined page capacity.

Performance of any database system is identified by its query processing speed or efficiency. The performance of a multidimensional structure for physical database organization is measured in terms of how efficiently the different types of queries (exact match, range query, partial match and partial range queries) are processed. We discuss here the query processing strategies for different classes of queries.

The basic operation in satisfying different types of queries is the search operation, which in our design is assisted by a directory represented by binary tree structured scales as discussed before.

For an exact match search, searching for a record corresponds to binary tree traversal to locate d leaf nodes which store the d indices, one for each attribute of the record.

A range query is defined by a set of intervals for each dimension (attribute). Each interval is represented by a lower and upper bound. By searching the BTS-scales for lower bounds and upper bounds in the intervals, a set of leaf nodes is located. These leaf nodes have the property that in each dimension the leaf node corresponding to the upper bound of the range query must lie at the right of or in the same leaf node corresponding to the low bound of the query range. Therefore, a range query corresponds to locating all leaf nodes lying between the leaf nodes corresponding to the lower and upper bound in each dimension. All the pages pointed at by a set of d different leaf nodes, one in each dimension, are candidates for searching. All

these pages are read in memory and the tuples satisfying the query are retrieved. Typically, the candidate pages for a range query form a d dimensional rectangular region.

In a partial match query, an attribute corresponding to a subset of d dimensions are specified. The leaf nodes corresponding to these attribute are located by searching the respective dimensions. All leaf nodes of other unspecified dimensions are candidates for searching. Similar to range query, the set of intersecting pages are read in memory and searched for the records satisfying the query.

Partial range query is similar to a range query except that a set of intervals are defined only for a subset of d dimensions. For each specified dimensions, all leaf nodes located between the lower and upper bound are searched. For all other unspecified dimensions, all leaf nodes are candidates for searching. Now all the intersecting pages are read in memory and searched for the records that satisfy the query.

We measure the performance of different classes of queries in terms of the efficiency of the query. Query efficiency is defined as the ratio of the number of records satisfying the query to the actual number of records traversed.

The different classes of queries for which we measure the efficiency are namely, partial match, range and partial range queries. The results are summarized in the table below. The results are for 10,000 records. The results are based on the following models.

- For exact match queries, we take the average number of disk accesses on 2000 randomly generated queries.
- For partial match we take the average efficiency by randomly generating 200 queries.
- For range and partial range queries we average the efficiency over a set of 20 randomly generated intervals.

The results clearly indicate that query processing efficiency is better for uniform data distribution.

NO. OF RECORDS	PARTIAL MATCH	RANGE			PARTIAL RANGE		
		LOW	MID	HIGH	LOW	MID	HIGH
SP1	0.002	0.27	0.72	0.81	0.55	0.87	0.91
SP2	0.001	0.22	0.55	0.70	0.44	0.80	0.87
SP3	0.0006	0.13	0.51	0.65	0.39	0.69	0.81

Table 5.3: Query Performance Results for 10,000 records in 2 Dimensions with Uniform Data Distribution

NO. OF RECORDS	PARTIAL MATCH	RANGE			PARTIAL RANGE		
		LOW	MID	HIGH	LOW	MID	HIGH
SP1	0.0007	0.18	0.45	0.64	0.35	0.63	0.71
SP2	0.0008	0.12	0.35	0.52	0.23	0.55	0.53
SP3	0.0002	0.09	0.30	0.47	0.18	0.37	0.59

Table 5.4: Query Performance Results for 10,000 records in 2 Dimensions with Data not Distributed Uniformly

NO. OF RECORDS	PARTIAL MATCH	RANGE			PARTIAL RANGE		
		LOW	MID	HIGH	LOW	MID	HIGH
SP1	0.0007	0.03	0.28	0.46	0.35	0.64	0.74
SP2	0.0003	0.008	0.16	0.23	0.20	0.50	0.63
SP3	0.0005	0.01	0.22	0.28	0.25	0.59	0.71

Table 5.5: Query Performance Results for 10,000 records in 3 Dimensions and $d=1$ for Partial Match and Partial Range Queries with Uniform Data Distribution

NO. OF RECORDS	PARTIAL MATCH	RANGE			PARTIAL RANGE		
		LOW	MID	HIGH	LOW	MID	HIGH
SP1	0.0003	0.01	0.16	0.30	0.15	0.35	0.41
SP2	0.0002	0.005	0.12	0.12	0.12	0.14	0.26
SP3	0.0001	0.005	0.10	0.15	0.10	0.18	0.20

Table 5.6: Query Performance Results for 10,000 records in 3 Dimensions and $d=1$ for Partial Match and Partial Range Queries with Data not Distributed Uniformly

NO. OF RECORDS	PARTIAL MATCH	RANGE			PARTIAL RANGE		
		LOW	MID	HIGH	LOW	MID	HIGH
SP1	0.0004	0.04	0.25	0.43	0.11	0.43	0.58
SP2	0.0002	0.004	0.13	0.29	0.03	0.24	0.35
SP3	0.0001	0.006	0.19	0.35	0.05	0.33	0.46

Table 5.7: Query Performance Results for 10,000 records in 3 Dimensions and $d=2$ for Partial Match and Partial Range Queries with Uniform Data Distribution

NO. OF RECORDS	PARTIAL MATCH	RANGE			PARTIAL RANGE		
		LOW	MID	HIGH	LOW	MID	HIGH
SP1	0.0002	0.02	0.15	0.25	0.08	0.19	0.29
SP2	0.0001	0.001	0.10	0.18	0.008	0.10	0.13
SP3	0.00008	0.001	0.12	0.20	0.01	0.15	0.12

Table 5.8: Query Performance Results for 10,000 records in 3 Dimensions and $d=2$ for Partial Match and Partial Range Queries with Data not Distributed Uniformly

Splitting Policy	Data Page Utilization	I/O Efficiency	Fan Out	Query Processing Efficiency
SP1	poor	good	poor	very high
SP2	good	good	good	high
SP3	good	poor	good	high

Table 5.9: Performance Comparison for Uniform Data Distribution

Splitting Policy	Data Page Utilization	I/O Efficiency	Fan Out	Query Processing Efficiency
SP1	poor	average	poor	average
SP2	average	average	good	average
SP3	average	average	average	average

Table 5.10: Performance Comparison of Data not Distributed Uniformly

5.4 Performance Comparison

Table 5.9 and Table 5.10 summarizes the performance comparison of the three splitting policies of Dynamic Multipaging for data uniformly and not uniformly distributed in 2 and 3 dimensions.

Table 5.9 and 5.10 summarize the performance comparison of the three splitting policies of Dynamic Multipaging for data uniformly and not uniformly distributed respectively in 2 and 3 dimensions. The actual distribution of the data is not known as the data is generated from a real world application. In the worst case the data could be non-uniformly distributed.

Results of Table 5.9 and Table 5.10 indicate that splitting policy 2 (SP2) has the best performance amongst the three policies. None of the policies perform well for data that is not distributed uniformly. Thus dynamic multipaging is suitable for multidimensional data that is distributed uniformly.

5.5 Summary

The overall results indicate that splitting policy 2 and 3 perform equally well except for the average number of disk accesses for record insertion. Splitting policy 3 performs badly in creating the dynamic multipaging structure. However, splitting policy 3 has higher query performance, that is it has higher efficiency in processing different classes of queries and the same should hold for higher dimensions. Splitting policy 1 has the best query performance for all classes of queries, however, it is not suitable for practical purposes as the data page utilization is too low. The performance of splitting policy 2 and 3 is close for all other parameters viz. probe factor, fan out and data page utilization. The results favor the use of splitting policy 2 for most of the applications that multi-dimensional storage structures are useful for. These applications include Geographical Information Systems (GIS) and Computer Aided Design (CAD). These applications are characterized by complicated geometric objects which need to be stored and retrieved with reasonable efficiency. The databases storing these geometrical objects are manipulated very frequently and so the structure must ensure that the data is read and written with reasonable efficiency. This is the reason why splitting policy 3 may not be suitable for these kind of applications, as it incurs huge number of disk accesses for record insertion thereby proving costly in creating the multipaging structure.

Chapter 6

Conclusion

We have visualized the working of insertion, deletion and query algorithms of multidimensional storage structures. Visualization enhances the understanding of such storage structures. The motivation for visualization stems from the fact that multidimensional storage structures are complex data structures that are difficult to understand and perceive. This is one of the reasons for their lack of use. The visualization depicts the multidimensional growth and shrinkage of the structure in response to user adding and deleting data. We have also illustrated the processing of exact match queries. The concurrency in our visualization is implemented by means of thread programming in Java. Our simulations in chapter 5 further establish the fact that Multidimensional storage structures process different class of queries with reasonable efficiency.

We hope the visualization of multidimensional storage structures in 2 dimensions and their extension to visualization in 3 and higher dimensions would appeal to the database community. This would hopefully also be a step further in the recognition of multidimensional storage structures and in future be incorporated in commercial database products. Benefits of employing them in applications like GIS (Geographical Information Systems) and CAD (Computer Aided Design) can be very rewarding. However, visualization of very large datasets in 3 or higher dimensions need to be implemented to establish the benefits of multidimensional storage structures.

The simulation studies carried out for the three splitting policies establish the result that splitting policy 2 (SP2) has the best performance results. These results are for 10,000 records uniformly distributed in 2 and 3 dimensions. The simulation results also indicate that dynamic multipaging also provides high query processing efficiency for different classes of queries. In particular for range and partial range queries which are central to applications like GIS, CAD, VLSI etc..

Yet another result is a generalized N dimensional implementation of dynamic multipaging. The obvious advantage of the generalized implementation is the structure can be used directly for visualizations in higher dimensions and taking performance results for higher dimensions is just a matter of reusing the structure implementation, which is developed as a Java Package. Any Java application that needs to use the structure can do so by simplifying importing the dynamic multipaging package.

In summary, the main results achieved in this thesis are 1) a visualization tool that simplifies the understanding of the complexity of multidimensional storage structures, 2) performance benefits of using splitting policy 2 (SP2) for implementing multidimensional storage structures, 3) high query processing efficiency for range and partial range queries which are central to several applications, and 4) a generalized N dimensional implementation of dynamic multipaging facilitating higher dimensional visualization and performance studies in future

References

- [1] Ekow, J. Otoo and Merrett, T. H., "Dynamic Multipaging: A Multidimensional Structure for Fast Associative Searching," TR SOCS-81-39, McGill University, 1981.
- [2] Merrett, T. H. and Louis Hamel, "Dynamic Multipaging in Even Less Space," *SIGMOD RECORD*, Vol. 17, No. 4, Dec 1988, pp. 56-61.
- [3] Merrett, T. H., Otoo, E. J., "Multidimensional Paging for Associative Searching", TR SOCS-81-18, McGill University, 1981.
- [4] Otoo, E. J., Merrett, T. H., "A Storage Scheme for Extendible Arrays", McGill University, TR SOCS-81-25, 1981.
- [5] Burkhard, W. A., "Interpolation-based Index Maintenance", *Proc, 2nd Symp. on Principles of Database Systems*, 1983, pp. 76-89.
- [6] Fagin, R., Nievergelt, J., Pippenger, N. and Strong, H. R., "Extendible Hashing: A Fast Access Method for Dynamic Files", *ACM Trans. on Database Systems*, Vol. 4, No. 3, Sept. 1979, pp. 315-344.
- [7] Hutflesz, A., Six, H. W. and Widmayer, P., "Globally Order Preserving Multidimensional Linear Hashing, Proc. of the 4th IEEE International Conference on Data Engineering, Los Angeles, 1989, pp. 572-579.
- [8] Kriegel, H. P., Seeger, B. "Multidimensional Order Preserving Linear Hashing with Partial Expansion", Proc. of International Conference on Database Theory, Sept. 1986, pp. 203-220.

- [9] Merett, T. H., Otoo, J. E., "Dynamic Multipaging: A Storage Structure for Large Shared Data Banks, Proc. 2nd International Conference on Databases. Improving Usability and Responsiveness, Academic Press, 1992, 237-256.
- [10] Nievergelt, J., Hintenberger, H., Sevick, K. C., "The Grid File: an Adaptable, Symmetric Multi-key File Structure, ACM Trans. on Database System, Vol. 9, No. 1, Mar 1984, pp. 38-71.
- [11] Otoo, E. J., "A Mapping Function for the Directory of a Multidimensional Extendible Hashing", *Proc. 10th International Conference in Very Large Data Bases*, Singapore, 1984, pp. 493-506.
- [12] Otoo, E. J., "Balanced Multidimensional Extendible Hash Tree", *Proc. 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 1986, pp. 110-113.
- [13] Ouksel, M., Scheuermann, P., "Storage Mapping for Multidimensional Linear Dynamic Hashing", *Proc. 2nd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 1983, pp. 90-105.
- [14] Larson, P. A., "Dynamic Hashing", *BIT* 18, 1978, pp. 184-201.
- [15] Bentley, J.L., "Multidimensional Binary Search Trees in Database Applications, *IEEE Trans. Software ACM*, SE-5, 4, 1979, pp. 333-340.
- [16] Finkel, R.A., Bentley, J.L., "Quad Tree - A Data Structure for Retrieval on Composite Keys", *Acta Informatica*, 4, 1, 1974, pp. 1-9.
- [17] Guting, H., Kriegel, H.P., "Multidimensional B-tree: An Efficient Dynamic File Structure for Exact Match Queries", *Proc. 10th GI Annual Conference, Informatic-Fachberichte*, 33, 1980, pp. 375-388.
- [18] Orenstein, Jack A., "A Class of Data Structures for Associative Searching", *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 1984, pp. 181-190.

- [19] Fagin, R., Nievergelt, J., et. al., "Extendible Hashing - a Fast Access Method for Dynamic Files", *ACM Trans. on Database Systems*, Vol. 4, No.3, Sept. 1979, pp.315-344.
- [20] Litwin, W. "Linear Hashing: a New Tool for Files and Table Addressing", *Proc. 6th International Conference on Very Large Databases*, 1980, pp. 212-223.
- [21] Tamminen, M., "The Extendible Cell Method for Fast Geometric Access", TR-HTKK-TKD-A20, Lab of Info. Proc. Sc., Helsinki Univ. of Technology.
- [22] Otoo, E. J., "Linearizing the Directory Growth in Order Preserving Extendible Hashing", *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems* 1986, pp. 580-588.
- [23] Otoo, E. J., "Balanced Multidimensional Extendible Hash Tree", *Proc 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 1986, pp. 110-113.
- [24] Hinrichs, K., "The Grid File System: Implementation and Case studies of Applications", Ph.D. Dissertations, Swiss-Federal Institute of Technology, Zurich. 1985.
- [25] Lin. Ying, "Ameliorating the drawbacks of the Grid File", *M.C.S. Thesis, Carleton University, Ottawa, Canada*, 1993.
- [26] Gutman, R., "R-trees: A dynamic index structure for spatial searching", *Proc. of ACM/SIGMOD Conf. on Management of Data*, Boston, MA, 1984.
- [27] Freeston, M., "The Bang File: A new kind of Grid File", *Proc. ACM SIGMOD*, 1987, pp. 260-268.
- [28] Robinson, J. T., "The K-D-B-tree: A search structure for large multidimensional dynamic indexes", *Proc. of ACM SIGMOD Conf. on Management of Data*, Ann Arbor, MI, 1981.

- [29] Lomet, D., Salzberg, B., "The hB-tree: A multidimensional indexing method with good guaranteed performance", *ACM Trans. Database Systems*, 15, 1992, pp. 625-658.
- [30] Evangelidis, G., Lomet, D., Salzberg, B., "The hB^π -tree: A multi-attribute index supporting concurrency, recovery and node consolidation", *VLDB Journal*, 6, 1997, pp. 1-25.
- [31] Anuff, Ed., "Java Source Book: A Complete Guide to Creating Java Applets for the Web", John Wiley & Sons, Inc., U. S. A., 1996.
- [32] Flanagan, D., "Java in a Nutshell", O'Reilly & Associates, Inc., U. S. A., Feb. 1996.
- [33] Keim, D. A., Kriegel, H.-P., "VisDB: A System for Visualizing Large Databases", *Proc. ACM SIGMOD Conf. on Management of Data*, San Jose, CA, 1995.
- [34] Keim, D. A., Kriegel, H.-P., Ankerset, M., "Recursive pattern: A Technique for Visualizing Very Large Amounts of Data", *Proc. Visualization'95*, Atlanta, GA, 1995.
- [35] Lee, J. P., "Data Exploration Interactions and the Exbase System", *Database Issue for Data Visualization*, eds., Springer-Verlag, Lecture Notes in Computer Science, Vol. 871, Nov. 1994, pp. 118-137.
- [36] S. Mukherjea, Foley, J., "Navigational View Builder: A Tool for Building Navigational Views of Information Spaces", *ACM SIGCHI Conf. Companion*, Boston, MA, Apr. 1994, pp. 289-290.
- [37] M. Stonebraker et al. "Tioga: Providing Data Management Support for Scientific Visualization Applications", *Proc. 19th Int. Conf. on Very Large Databases*, Dublin, Ireland, August 1993, pp. 25-38.

- [38] M. P. Consens et al. "Architecture and Applications of the *Hy*⁺ Visualization System.
- [39] Inselberg, A., Dimsdale, B., "Parallel Axis techniques for Visualization of Multidimensional data", *Proc. IEEE Conf. on Visualization*, San Francisco, Oct. 1990, pp. 361-375.
- [40] Bergeron, R. D., Grinstein, G., "A Reference Model for the Visualization of Multidimensional data", *Proc. Eurographics*, Amsterdam, 1989, pp. 393-399.
- [41] Mihalisin, T., Timlin, J., Schwegler, J., "Visualization and Analysis of Multivariate Data: A Technique for all Fields", *IEEE Visualization*, 1991, pp. 171-178.
- [42] Anupam, V., Dar, S., Leibfried, E., "Data Space-3-D Visualization of Large Databases", *Proc. Int. Symp. on Information Visualization*, Atlanta, GA, 1995, pp. 82-88.
- [43] G. G. Robertson, J. D. Mackinlay, S. Card, "Cone Trees: Animated 3D Visualizations of Hierarchical Information", *Proc. of ACM SIGCHI Conf. on Human factors in Computing Systems*, New Orleans, La, April 1991, pp. 189-194.
- [44] Young, F. W., P. Rheingans, "Visualizing Structure in High Dimensional Multivariate Data", *IBM Journal of Research and Development*, Vol. 35, No. 1/2, 1991, pp. 97-107.
- [45] Buja, A., D. Cook, D. Swayne, "Interactive High Dimensional Data Visualization", *Journal of Computational and Graphical Statistics*, Vol. 5, No. 1, 1996, pp. 78-99.
- [46] Keim, D. A., "Enhancing the Visual Clustering of Query-Dependent Database Visualization Techniques using Screen-Filling Curves", *Proc. of IEEE Visualization Workshop*, Lecture Notes in Computer Science, Atlanta, Georgia, USA, Oct. 1995, pp. 101-110.

- [47] S. Mukherjea, J. Foley, S. Hudson, "Visualizing Complex Hypermedia Networks through Multiple Hierarchical Views", *Proc. of ACM SIGCHI Conf. on Human Factors in Computing Systems*, Denver, Co, May 1995, pp. 331-337.
- [48] Java Workshop
"http://www.sun.co.jp/sunsoft.jp/products/Devpro/JavaWorkshop.html".
- [49] Lynn Weaver, Bob Jervis, "Inside Java Workshop (Java Series)", Bk&Cd-Rom Edition Paperback, Prentice Hall Publication.
- [50] Rodger Lea, R., Matsuda, K., Miyashita, K., Roger Lea, "Java for 3D and VRML Worlds, Bk&Cd-Rom Edition Paperback, Sam's Publication.
- [51] Scott Oaks, Henry Wong, "Java Threads", 1 Ed., Paperback, O'Reilly Associates Publication.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved