A Hierarchy Navigation Framework: Supporting Scalable Interactive Exploration over Large Databases

by

Nishant K. Mehta

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

____

August 2004

APPROVED:

____

Professor Elke A. Rundensteiner, Thesis Advisor

____

Professor Matthew O. Ward, Thesis Advisor

____

Professor Craig Wills, Thesis Reader

____

Professor Michael Gennert, Head of Department

Abstract

Modern computer applications from business decision support to scientific data analysis use visualization techniques. However, visual exploration tools do not scale well for large data sets, i.e., the level of clutter on the screen is typically unacceptable. To solve the problem of cluttering at the interface level, visualization tools have recently been extended to support hierarchical views of the data, with support for focusing and drilling-down using interactive selection.

To solve the scalability problem, we now investigate how best to couple such a near real-time responsive visualization tool with a database management system. Our solution proposes a framework containing three major components: hierarchy encoding, caching and prefetching. Since the direct implementation of the visual user interactions on hierarchical data sets corresponds to recursive query processing, we have developed a hierarchy encoding method, called the MinMax tree, that pushes the on-line recursive processing step into an off-line precomputation step. The MinMax encoding scheme allows us to map the hierarchy to a 2-dimensional space and the recursive navigation operations at the interface level to 2-dimensional spatial range queries. These queries can then be answered efficiently using spatial indexes. To compliment this encoding scheme we employ a caching strategy that exploits user navigation characteristics to cache the nodes having high probability of being referenced again. Based on user characteristics we choose to implement two replacement policies one which exploits temporal locality (LRU) and the other exploits spatial locality (Distance). Also, to enhance the performance of the cache we propose using a prefetching mechanism that predicts and prefetches future user requests into the cache. Together the components form a comprehensive framework that scales the visualization tool to support navigation operations over large data sets.

The techniques have been incorporated into XmdvTool, a free software package for multi-variate data visualization and exploration. Our experimental results quantify the

effectiveness of each component and show that collectively the components scale the XmdvTool to support navigation operations over large data sets. Mainly, our experimental results show that together the components can achieve 63% to 96% reduction in response time latency even with limited system resources.

Acknowledgements

I would like to take this opportunity to thank all the people who helped me in my work in any way. A special thanks to Anilkumar Patro for his suggestions and effort to make this work successful.
I would like to express my greatest gratitude to my advisors, Prof. Elke A. Rundensteiner and Prof. Matthew O. Ward, for their guidance and invaluable contributions to this work.
I would like to thank the XMDV team: AnilKumar Patro, Jing Yang, Wei Peng and Shiping Huang.

This work is supported under NSF grant IIS-0119276.

Contents

1 Introduction **1**

2 Visual Data Exploration **5**

2.1 XmdvTool: The Motivating Application 5

2.2 Visual Brush-Based Exploration 6

2.3 Structure-Based Brushing in XmdvTool 7

2.4 Brush Semantics 9

3 MinMax Trees: Translating Navigation Operations **12**

3.1 Labeling the Nodes 12

3.2 2-D Hierarchy Maps 15

3.3 Using 2-D Hierarchy Maps to Implement Structure-Based Brushes 16

3.4 Translating Structure-Based Brushes into SQL 17

4 Backend Framework **19**

4.1 Spatial Index 20

4.2 Delta Calculator 21

4.3 Cache 23

4.3.1 Cache Replacement 24

4.3.2 Direction-based Prefetching 27

5 System Implementation **30**

5.1 System Architecture 30

5.2 WorkFlow 33

6 Experimental Results **35**

6.1 Experimental Setup 35

6.2 Metrics 36

6.3 Database Index 37

6.4 Cache Size 40

6.5 Comparison of Replacement Policies 42

6.6 R-Tree Cache Index 44

6.7 Prefetcher 47

6.8 User Trace Analysis 49

6.9 Discussion 51

7 Related Work **52**

7.1 Visual Hierarchy Exploration 52

7.2 Visualization-Database Integrated Systems 53

7.3 Hierarchy Encoding 54

7.4 Caching 56

7.5 Prior Work in XmdvTool 57

8 Conclusions and Future Work **60**

8.1 Conclusions 60

8.2 Future Work 62

List of Figures

2.1 Structure-based brush: *focus region* (a) and *density factor* (b). 7

2.2 Structure-based brush: *horizontal* (a) and *vertical* (b) selection. 7

2.3 Cluttered parallel coordinates. 8

2.4 Structure-based brush in XmdvTool. 8

2.5 After focused area drilled-down. 9

2.6 Structure-based brush showing drill-down. 9

2.7 After focused area rolled-up. 10

2.8 Structure-based brush showing roll-up. 10

3.1 Labeled Minmax tree 14

3.2 2-D Hierarchy Map 15

3.3 2-D brush selection with b_{min}=0.4, b_{max}=0.9 and lod=0.35 16

4.1 System architecture. Solid lines represent main modules. Ovals represent data. Arrows show control flow. 19

4.2 Computing Δ queries. The line segment represents the brush. $(b_{min} = 0.3, b_{max} = 0.78, lod = 0.6)$ 22

4.3 Snap shot of the cache state 25

4.4 Snap shot of cache contents before replacement 26

4.5 Snap shot of cache contents after replacement 27

4.6 Example of brush store with overlapping brushes 28
4.7 Analysis of User Traces 28
5.1 System Architecture 31
6.1 Latency in msec with and without database index, out5d data set 38
6.2 Latency in msec with and without database index, uvw data set 38
6.3 Latency reduction ratio with relative cache size 10 %, out5d data set . . . 39
6.4 Latency reduction ratio with relative cache size 2%, uvw data set 39
6.5 Comparison of cache size vs. average latency, out5d data set 40
6.6 Comparison of cache size vs. average latency, uvw data set 41
6.7 Cache size vs. hit ratio for Distance and LRU replacement, out5d data set 42
6.8 Cache size vs. *lrr* for Distance and LRU replacement, out5d data set . . . 43
6.9 Cache size vs hit ratio for Distance and LRU replacement, uvw data set . 43
6.10 Cache size vs. *lrr* for Distance and LRU replacement, uvw data set 44
6.11 Comparison of cache size vs. latency with and without main memory R-Tree index, out5d data set 45
6.12 Comparison of cache size vs hit ratio, out5d data set 45
6.13 Comparison of cache size vs latency with and without main memory R-Tree index, uvw data set 46
6.14 *lrr* of prefetch vs. no prefetch, user3, out5d data set 47
6.15 Hit ratio of prefetch vs. no prefetch, user3, uvw data set 48
6.16 Average *lrr* for 4 user traces, prefetch vs. no prefetch , out5d data set . . . 48
6.17 Average *lrr* for 4 user traces, prefetch vs. no prefetch , uvw data set . . . 49
6.18 Localtiy % vs. hit ratio, uvw data set 50
6.19 Directionality % vs. hit ratio, uvw data set 51

List of Algorithms

1 Vertical Selection 10
2 Delta Calculator 23

Chapter 1

Introduction

Whether the domain is stock market data, scientific data, or the distribution of sales, visualization is becoming an increasingly popular technique for data exploration. Visualization tools exploit the fact that humans can detect patterns and trends in the underlying data by just looking at it, *without* having to be made aware in advance about what pattern they'll face. Human perception is greatly influenced by the way information is presented. Thus various techniques for displaying data have been proposed, each of which emphasizes different characteristics of data. However, most of these techniques do not scale well with respect to the size of the data. As a generalization, [20] postulated that any method that displays a single entity per data point invariably results in overlapped elements and a convoluted display that is not suited for the visualization of huge datasets.

[19] proposed an approach called hierarchical displays for displaying and visually exploring large datasets. The idea was to present data at different levels of detail based on clustering the initial data points into a hierarchy called the *cluster tree*. The problem of clutter at the interface level is solved by displaying only one level of detail at a time. However, such hierarchical summarizations captured at different levels in the cluster tree in fact increase the size of the input data set by at least one order of magnitude, as the

clusters that store aggregate information are in addition to the already existing data points. Hence management of data remains an even more critical issue. While storing the data in main memory or in flat files is appropriate for small and moderately sized data sets, this becomes unworkable when scaling to large data sets on the order of 100,000 data points or more.

One solution to this is to integrate visualization tools with a back-end database management system. However, interactive exploration operators in structure space (cluster tree) like navigation and selection [55] are not directly supported by traditional database management systems. In particular the recursive processing involved when exploring hierarchies in main memory is no longer appropriate when storing these hierarchies on disk. Thus, in this thesis we propose a framework to meet the interactive response requirements for user exploration operations over large hierarchies stored in the database. The framework includes a hierarchy encoding mechanism, caching and prefetching strategy each of which work collectively to reduce the response time latency.

The hierarchy encoding technique, called *MinMax trees* allows us to map hierarchies to a 2-dimensional space called a *2D Hierarchy Map*. This mapping in turn allows us to represent visual navigation operations as spatial queries over the *2D Hierarchy Map*. This *2D Hierarchy Map* is stored in a database, where the searches are executed efficiently using spatial indexes.

Furthermore, interactive visual exploration tools exhibit a variety of characteristics that can be exploited to make the system scale to huge data sets. These include locality of exploration and data access, predictability of user's exploratory movements, and presence of idle time between user operations. To take advantage of the above characteristics we propose a caching strategy that buffers the recently used data items. The cache exploits the spatial mapping provided by the hierarchy encoding scheme to build a memory resident spatial index for fast cache look-up. Moreover, it also uses spatial space to implement

a semantic replacement policy that replaces cached objects based on the spatial distance from the current active selection.

The predictability of user movements and idle time between user operations can be effectively utilized for predicting and prefetching future user requests. We integrate a directional prefetcher proposed in [15] into our system. Prefetching helps in reducing cache misses and thus improves the performance.

We applied our proposed solution strategies to the hierarchical navigation tool (*structure-based brush*) in XmdvTool [52], a software package for exploring and visualizing multivariate data sets. However, this context is neither implicitly or explicitly assumed in this thesis. Visual navigation of huge hierarchies is a general problem and we describe a general approach towards solving the problem. The results of the performance study show that the approach scales to large data sets. Even for moderate data sets our solution reduce the user response time by 63 to 96 percent.

The main contributions of this thesis are:

- A hierarchy encoding technique that reduces the tree to an equivalent spatial representation. This representation allows us to map recursive hierarchy navigation operations to spatial search queries that can be answered efficiently using existing spatial index structures.

- A framework that exploits the encoding technique and characteristics of the visual navigation environment such as, locality of user exploration, for efficient retrieval of online data. The framework includes:

  1. A main memory caching strategy that buffers the recently used nodes to avoid database fetches and thus improves system response time.

  2. A cache replacement policy called distance that exploits spatial locality in user traces to replace the nodes with maximum distance from the current active

selection. This in turn leads to higher hit ratios and better cache performance.

3. Index structures that exploit the spatial representation derived using the hierarchy encoding technique to achieve faster searches on the cache and on the database contents.

4. A direction-based prefetching strategy that exploits the limited means of data requests via the visual interactive tools to predict future user requests and prefetch the required data into the cache.

- An object-oriented implementation of the complete framework using C++ and Oracle database. This framework forms the backend module of the XmdvTool and scales the hierarchical displays [55] to support navigation of large hierarchies.

- Experimental evaluation that quantifies the relative effectiveness of each component in the framework towards latency reduction. It shows that the components work collectively and in some cases can reduce the system latency up to 95%.

This thesis is organized as follows. Chapter 2 introduces the basic concepts in multivariate hierarchical visualization. The hierarchy encoding as well as the processing of the *MinMax* queries are presented in Chapter 3. Chapter 4 introduces the proposed framework and describes the important components of the framework. Chapter 5 describes the system implementation. Chapter 6 presents the results of the evaluation study. Chapter 7 surveys related work. Chapter 8 presents conclusions and directions for future work.

Chapter 2

Visual Data Exploration

2.1 XmdvTool: The Motivating Application

XmdvTool is a visualization tool designed for exploration and analysis of multivariate data sets. The tool provides four distinct visualization techniques namely, scatterplot matrices [9], parallel coordinates [23, 53], glyphs [1, 43, 7, 37] and dimensional stacking [29], with interactive selection operations and linked views. To scale the display techniques to large data sets, we need to reduce the amount of clutter in screen space. To address this issue, our efforts have produced versions of display techniques that allow multi-resolution data presentation [19, 20, 55]. Multi-resolution techniques allow users to view the data sets at an abstract level of detail and actively explore the datasets by zooming in (*drill-down*) or zooming out (*roll-up*) on subsets of the datasets. The subsections to follow explain these operations in detail.

The main objective of the work in this thesis was to improve the efficiency of database support in XmdvTool. However, the operations that we will introduce are general and can be used for visual exploration of arbitrary hierarchies, a common class of navigation operations in large scale visualization systems [18].

2.2 Visual Brush-Based Exploration

Brushing is the process of interactively painting over a subregion of the data display using a mouse, stylus, or other input device that enables the specification of location attributes [2, 52]. The location attribute values are then used to select subsets of the data.

Brushing can be performed in screen or data space to specify a *containment criterion*, i.e., whether a particular point is inside or outside the brush. In *screen space* techniques, a brush is specified by a 2-D contiguous subspace on the screen. In *data space* techniques, a specification consists of either an enumeration of the data elements contained within the brush or the N-Dimensional boundaries of a hyper-box that encapsulates the selection.

A third category, namely *structure space* techniques, that allows selection based on structural relationships between data points, was introduced in [20]. The *structure* of a data set specifies relationships between data points. This structure may be explicit (e.g., categorical groupings or time-based orderings) or implicit (e.g., resulting from analytic clustering or partitioning algorithms). Examples of structures include linear orderings, trees and directed acyclic graphs. In this work we focus on trees.

A *tree* is a convenient mechanism for organizing large data sets. By recursively partitioning data into related groups and identifying suitable summarizations for each cluster, we can examine the data set methodically at different levels of abstraction, moving down the hierarchy (*drill-down*) when interesting features appear in the summarizations and up the hierarchy (*roll-up*) after sufficient information has been gleaned from a particular subtree.

Brushing in structure space involves two containment criteria. For the first containment criterion lets us assume that the leaves of the tree are chained together. Chaining imposes an order on the set of nodes in the cluster tree. Given this order, nodes that fall into a user defined interval satisfy the containment criteria. Intuitively, the two values that

form the interval represent the left and the right-most leaves of the selected subtree. We call this process "horizontal selection".

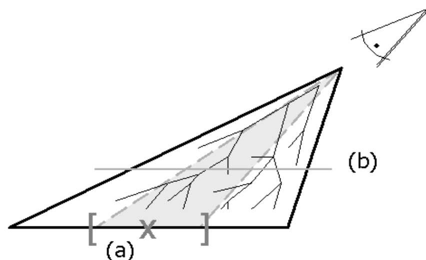

Figure 2.1: Structure-based brush: *focus region* (a) and *density factor* (b).

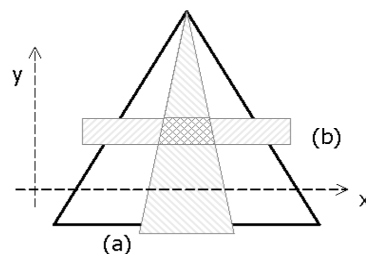

Figure 2.2: Structure-based brush: *horizontal* (a) and *vertical* (b) selection.

For our second containment criterion, we augment each node in the hierarchy, i.e., each cluster, with a monotonic value that indicates the *level-of-detail*. The nodes at the desired *level-of-detail* are selected. This process is called "vertical selection". The *level-of-detail* value can have different semantics. For example, it may represent the *width* of the cluster i.e., the number of leaf nodes the cluster encompasses. It could also signify the *distance* of the cluster from the root.

A structure-based brush is thus defined by a subrange of the structure extents and the *level-of-detail* values. Intuitively, if looking at a tree structure from the point-of-view of its root node (Fig. 2.1), the extent subrange appears as a *focus region* (with the focus point at its center), while the *level-of-detail* subrange corresponds to a sampling rate factor or a *density*. In a 2-D tree representation, the subranges correspond to a horizontal and vertical selection, respectively (Fig. 2.2).

2.3 Structure-Based Brushing in XmdvTool

Figure 2.3 shows a parallel coordinates display of a five dimensional data set having 16,384 records. In this display each of the N dimensions is represented by a vertical

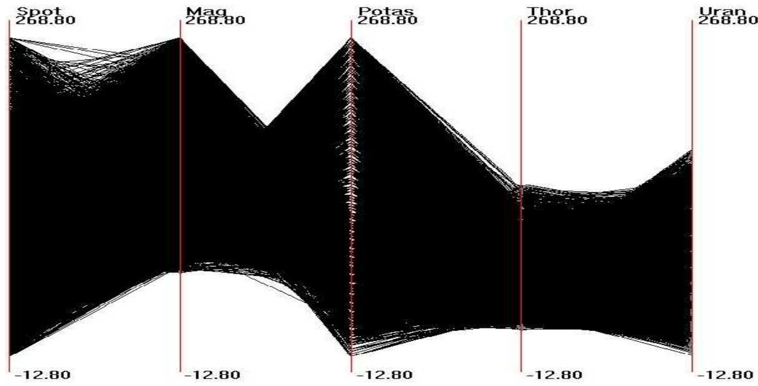

Figure 2.3: Cluttered parallel coordinates.

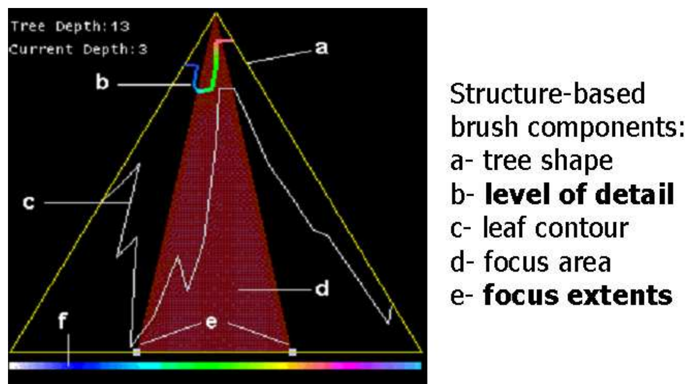

Figure 2.4: Structure-based brush in XmdvTool.

axis. A data point in N-dimensional space is mapped to a polyline that traverses across all N axes, crossing each axis at a position proportional to its value for that dimension. As seen from Figure 2.3, displaying all the data to the user at the same time results in display clutter. Hence, to support the visual navigation of cluster trees for large data sets, XmdvTool contains a structure-based brush.

Figure 2.4 shows the structure-based brushing interface implemented in XmdvTool. The triangular frame depicts the hierarchical tree. The contour near the bottom of the tree delineates the approximate shape formed by chaining together the leaf nodes. To navigate the hierarchy the tool provides two main “sliders”. The *level-of-detail* slider denoted by

'b' allows users to navigate the tree vertically and view clusters at different levels of detail. The *focus extents* slider denoted by 'e' allows users to move horizontally and focus on a subset of clusters within the same level. The left and right extents of the 'e' slider can also be adjusted individually to modify the width of the focus area. Figure 2.5 displays the same data set as Figure 2.3 but focused on a specific cluster of data points; this is after the user narrows the width of the focus area using 'e' and performs a drill-down operation using 'b' as reflected in Figure 2.6. Figure 2.7 displays the same data set as Figure 2.5 but showing the mean values and the range of the data points in that cluster. This is after the user performs a roll-up operation using 'b' as seen in Figure 2.8.

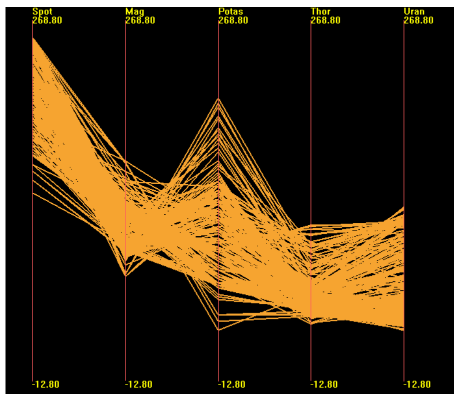

Figure 2.5: After focused area drilled-down.

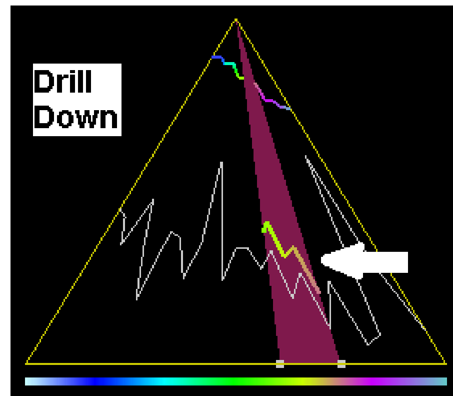

Figure 2.6: Structure-based brush showing drill-down.

2.4 Brush Semantics

A structure-based brush is defined as the intersection of two independent selections, the horizontal extents of the brush e_1 and e_2 and the $level - of - detail$. Setting such a brush requires two computational phases as well.

The first one, the horizontal selection, is accomplished in two steps. In the first step a set of leaf nodes is initially selected based on the order property. Basically, this step

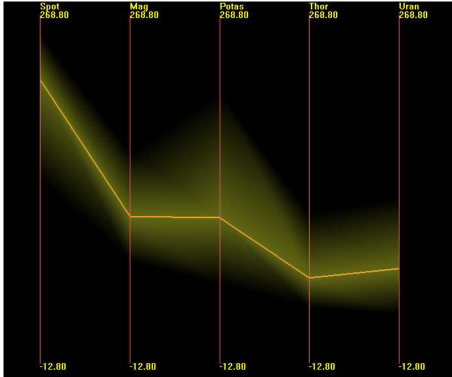

Figure 2.7: After focused area rolled-up.

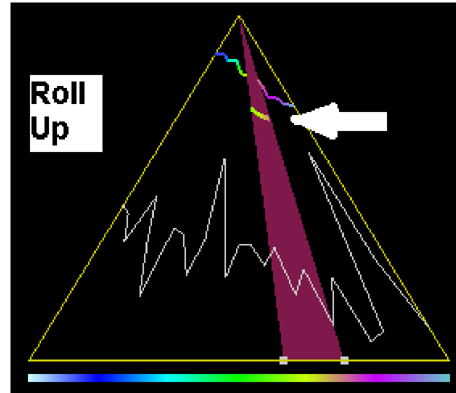

Figure 2.8: Structure-based brush showing roll-up.

corresponds to "select all leaves between the two extreme values e_1 and e_2 ". In the second step, the initial selection is propagated up towards the root based on either the ANY or the ALL semantics: "select nodes that have ANY (or ALL) of its children already selected ".

For vertical selection we use the *level-of-detail* value that has been associated with each node in the hierarchy. This can be any montonically increasing or decreasing value from the root towards the leaves. The algorithm below explains the process of vertical selection. The input to the algorithm is the *level-of-detail* of the brush. Here we assume that *level-of-detail* values are monotonically decreasing from the root towards the leaves. The function $lod(x)$ returns the *level-of-detail* of node x.

Algorithm 1 Vertical Selection

1: Let S and W be two sets of nodes.
2: Let S initially contain only root node. Let W be empty.
3: **while** S is not empty **do**
4:     Remove node n from S
5:     **if** $lod(n) \leq lod(brush)$ **then**
6:         Insert n into W
7:     **else**
8:         Insert descendants of n into S
9:     **end if**
10: **end while**

At the end of the run the set W contains the nodes that satisfy the vertical selection criteria. The algorithm starts traversing the tree from the root towards the bottom of the tree breadth-wise to find all the nodes that have the *level-of-detail* $\leq lod(brush)$. The main intuition is that, the *drill-down* or *roll-up* in the structure space corresponds to climbing up or descending a particular branch in the hierarchy.

The set of nodes that satisfy both the selection criteria forms the final set of the nodes in the brush. The brush operations, as described above, are inherently recursive. Recursive processing in relational database systems can be time consuming and thus is not suitable for interactive applications. In Section 3 we develop equivalent but non-recursive computation methods for setting structure-based brushes based on assigning precomputed values to the nodes that recast retrievals as range queries.

Chapter 3

MinMax Trees: Translating Navigation Operations

The question addressed in this chapter is how to translate the visualization operations into database operations? For this purpose we have developed an encoding technique called a *MinMax tree*.The method places the recursive processing into a *precomputation* stage, during which labels are assigned to all nodes. The labels provide a containment criterion. Thereafter, by looking only at a node's label independent of any other node in the hierarchy we can determine whether that node belongs to the active selection or not.

3.1 Labeling the Nodes

The containment criteria for a node in the cluster tree is based on two selections: horizontal and vertical. To map the recursive process of selection to a non-recursive one we augment each node in the cluster tree with horizontal and vertical extents. Each of these extents forms an interval. We call this tree a MinMax tree.

A MinMax tree is an n-ary tree. The horizontal extents of the nodes correspond to

open intervals defined over a totally ordered set, called an *initial set*. The horizontal extents of the leaf nodes in the tree form a sequence of non-overlapping intervals. The non-leaf nodes are unions of intervals corresponding to their children. The initial set can be continuous (such as an interval of real numbers) or discrete (such as a sequence of integers).

It is always possible to draw the tree such that all the leaf nodes are horizontally ordered. The leaf nodes are then labeled with pairs of values corresponding to the extents of their interval. As the intervals of non-leaf nodes are unions of their children intervals, it follows that a non-leaf node will be labeled with the minimum extent of its first interval and the maximum extent of its last interval. A node n having two children with intervals $c_1 = (\alpha, \beta)$ and $c_2 = (\gamma, \delta)$ such that $\alpha < \gamma$, will be labeled as $n = (\alpha, \delta)$. Figure 3.1 gives an example of a labeled cluster tree. For the tree in Figure 3.1 the process of assigning the horizontal extents started at the leaf nodes. The interval between 0 to 1 was divided equally between all leaf nodes. These intervals were propagated up towards the root. The interval for each non-leaf node is the union of the intervals of its children, as can be seen for Figure 3.1.

Given a MinMax tree T and two nodes x and y of T whose horizontal extent values are (x_1, x_2) and (y_1, y_2) respectively, node x is an ancestor of node y if and only if its horizontal extents $x_1 \leq y_1$ and $x_2 \geq y_2$. The containment property is based on the intuition that each node in the tree is included in its parent's interval. The horizontal extents of the nodes encode the ancestor-descendant relationships in the tree structure.

As described in Section 2.4, the horizontal selection first selects all the leaf nodes that lie within the focus extents of the brush, namely, e_1 and e_2. This selection is then propagated upwards by selecting the non-leaf nodes if ANY or ALL of their children are selected, depending upon the brush semantics. The output of this selection process is a set of subtrees of the cluster tree. The focus extents of the brush and horizontal extents of

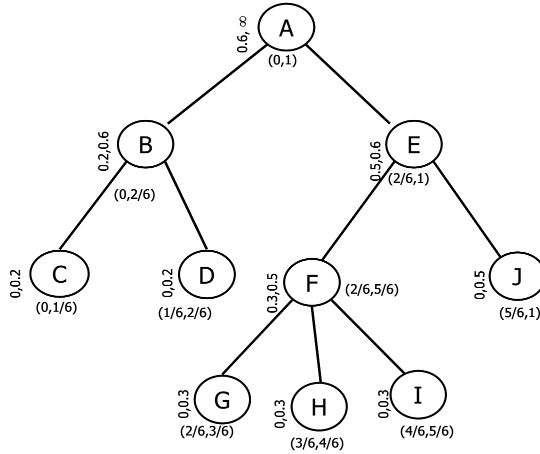

Figure 3.1: Labeled Minmax tree

each node in the MinMax tree both define an interval. Using MinMax trees we can reduce this process of selecting nodes in the ANY brush to searching for nodes in the MinMax tree whose interval intersects with the interval of the brush. Similarly, the process for the ALL brush maps to selecting nodes whose intervals are fully contained in the brush interval. Intuitively, we can see that if the horizontal interval of a node intersects with that of the brush, we know that the interval of at least one of its children intersects with that of the brush. If the horizontal interval of the node is fully contained within the brush, the horizontal interval of all its children is also contained in the brush.

For selection of the *level-of-detail* (vertical selection), the MinMax tree augments each node in the cluster tree with a vertical extent value. Given the brush semantics in Section 2.4 the process of labeling the nodes with vertical extents can be defined as follows. The vertical extent of a node A is the interval (v_1, v_2) where $(v_1 = lod(A), v_2 = lod(parent(A))$ where the function $parent(n)$ returns the parent node of node n. The node n with vertical extents (v_1, v_2) lies in the brush if $v_1 \leq lod(brush) < v_2$ is true. Note this is an alternate method of implementing the vertical selection algorithm described in Section 2.4. The vertically aligned extents in Figure 3.1 show the vertical extents for each node in the cluster tree.

Essentially, the process of labeling the nodes is a recursive one. The intervals are computed and assigned off-line at the time the hierarchy is created. By being off-line the cost of labeling does not affect the interactive user navigation response time. The codes assigned to the nodes in the cluster tree can be used for computing brush selections for navigation operations. The value and the distribution of the intervals (as well as the tree structure itself) depend on the technique used to create the hierarchy. However, it does not affect the correctness of the proposed method.

3.2 2-D Hierarchy Maps

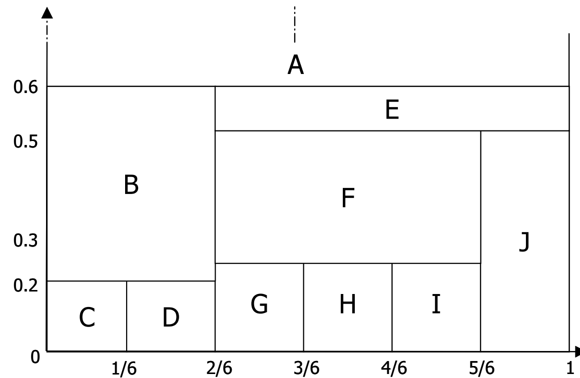

Figure 3.2: 2-D Hierarchy Map

We now make the important observation that the labels assigned by the MinMax procedure can be viewed as giving each node a spatial representation. The complete cluster tree can thus be mapped to a 2 dimensional space. We call this type of representation a *2-D hierarchy map*. Figure 3.2 shows a *2-D hierarchy map* for the MinMax tree in Figure 3.1. A node n with horizontal extents (h_{min}, h_{max}) and vertical extents (v_{min}, v_{max}) maps to a rectangular region in the *2-D hierarchy map* with the bottom left corner at (h_{min}, v_{min}) and the upper right corner at (h_{max}, v_{max}).

We observe that the 2-D hierarchy map exhibits the following important properties:

- The space between $(0,0)$ to $(1,1)$ is **completely filled**, i.e., given any point between $(0,0)$ and $(1,1)$ there exists a node that contains the point.

- Interiors of **no** two nodes **overlap** in the 2-D hierarchy map.

3.3 Using 2-D Hierarchy Maps to Implement Structure-Based Brushes

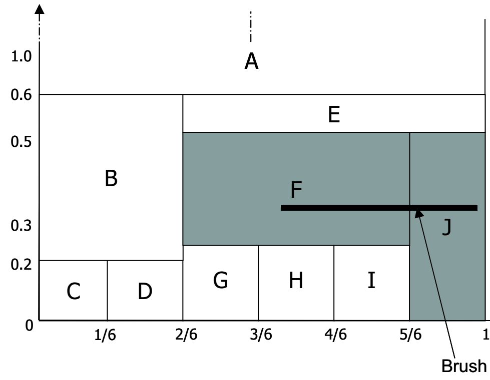

Figure 3.3: 2-D brush selection with b_{min}=0.4, b_{max}=0.9 and lod=0.35

From the 2-D hierarchy map, we can implement ANY and ALL structure-based brushes as non-recursive operations. The containment criteria for the ANY structure-based brush can be defined as follows. Given the brush's horizontal extents (b_{min},b_{max}) and the *level-of-detail*=lod, any node n having horizontal extents (h_{min}, h_{max}) and vertical extents (v_{min}, v_{max}) lies in the brush iff:

- The extents (h_{min}, h_{max}) intersect the brush interval (b_{min}, b_{max}), and

- $v_{min} < lod \leq v_{max}$.

A node n lies in the ALL structure-based brush iff:

- $(h_{min}, h_{max}) \cap (b_{min}, b_{max}) = (h_{min}, h_{max})$, and

- $v_{min} < lod \leq v_{max}$.

This containment criteria for the node n can also be stated differently. If we map the brush to a line segment with end points at (b_{min}, lod) and (b_{max}, lod) in the 2-D hierarchy map, a node n lies in the ANY structure-based brush if its representation in the 2-D hierarchy map intersects with that of the brush. A node n lies in the ALL structure-based brush, if the line segment representing the brush intersects both the right and left edge of the node. Our reformulation succeeds to map the process of searching for the nodes in the ALL and ANY brush into spatial queries.

Figure 3.3 gives an example of the selection for the ANY brush where b_{min}=0.4, b_{max}=0.9 and lod=0.35. Figure 3.3 shows the brush in black and all the selected nodes (i.e., the active set) in dark grey.

3.4 Translating Structure-Based Brushes into SQL

The 2-D hierarchy map technique reduces the containment criterion from initially recursive semantics to an inclusion test in the horizontal and vertical direction. We can thus decide whether a node should belong to the active set or not *independently* from the information stored in the other nodes. One scan of the hierarchy is hence sufficient to form the selection.

Let H be the relational table that stores the nodes in the hierarchy. Each tuple in H models one node in the cluster tree and has horizontal and vertical extents of the node, besides the node information. We have:

H (e_{min},e_{max},v_{min},v_{max}, ...)

An ANY **structure-based brush** having horizontal extents (b_{min}, b_{max}) and *level-of-detail* (lod) can be expressed as a range query as follows.

select * from H

where $e_{min} \leq b_{max}$ and $e_{max} \geq b_{min}$

and $v_{min} \leq lod$ and $v_{max} > lod$

An ALL structure-based brush query for the same parameters is specified by:

select * from H

where $e_{min} \geq b_{min}$ and $e_{max} \leq b_{max}$

and $v_{min} \leq lod$ and $v_{max} > lod$

While recursive processing would require an exponential processing time, this range query requires only a linear processing time for computing brushing results. Also we note here a few characteristics of the queries generated when moving the structure-based brush.

1. Regularity of query type: The structure of the query generated by user movements remains the same only; the parameters to the query differ depending on the user's current position.

2. Continuous user selection: The nodes in the selected subset when mapped to a 2D space yields a continuous subspace in the 2D Hierarchy Map.

3. Single level-of-detail display: The user selection can display nodes only at a particular level-of-detail. That is, no two nodes within the selected subset of nodes can have an ancestor-descendant relationship.

Chapter 4

Backend Framework

Given that we can translate the brush operations into spatial queries, we now describe the components in the backend framework. The components exploit this mapping along with typical user trace characteristics to reduce the user response time latencies and in turn scale the visualization application (front-end) to work with large data sets.

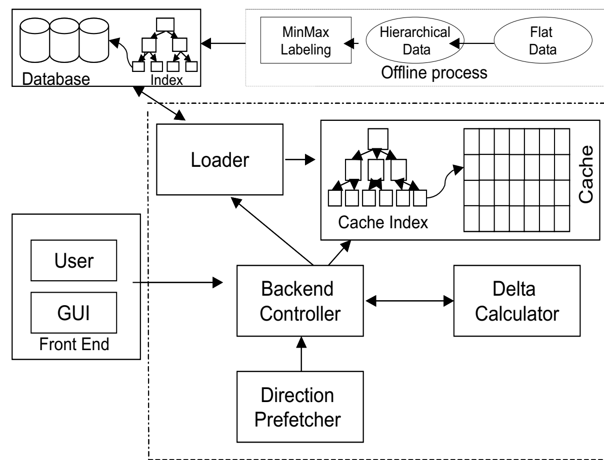

Figure 4.1: System architecture. Solid lines represent main modules. Ovals represent data. Arrows show control flow.

Figure 4.1 depicts the components in the backend framework of the XmdvTool. The cache is used to buffer the recently used data items. The prefetcher predicts user requests and fetches data into the cache. For each user request the cache is quickly searched to find

the requested objects. The cache may contain all the requested nodes, or only a subset. In the latter case the delta calculator computes a remainder query to fetch the subset of nodes not in the cache. The loader fetches the result of the remainder query into the cache. Once all the requested nodes are in the cache they are delivered to the front-end. The sections below explain each of the important components in the system depicted in Figure 4.1.

4.1 Spatial Index

For each request from the front end we need to quickly search the contents of the cache, compute the difference query and fetch the data from the database. We thus need a fast search mechanism both for the cache and for the database. A characteristic of the objects in the database and in the cache is that they are not referenced by their IDs when requested by the front-end. In other words, the front-end doesn't ask for the object x or y. Instead, it passes a query q to the back-end to search for objects that lie within the brush. Recall that query q in our exploration paradigm defines an instance of the brush with horizontal extents (b_{min},b_{max}) and *level-of-detail*=lod. This brush maps to a segment in the *2-D hierarchy map* (Section 3.2). The answer to the query is a set of clusters that intersect this segment. Therefore the query q is a 2 dimensional spatial range query as shown in Section 3.4. To execute this query efficiently we thus propose to use a spatial index.

A spatial index, in contrast to a B+ tree, utilizes spatial relationships to organize data entries with each key value seen as a point or a region in a k-dimensional space. Many spatial index structures have been proposed, each of which has its pros and cons. For our purpose we require a spatial index that works for spatial range queries and supports high update rates because the contents of the cache are continuously changing. Most spatial indexes do not perform well when the objects exhibit a high degree of overlap. However, note in the *2-D hierarchy map* the interiors of no two objects overlap. Thus this is not an

issue for us.

In our current implementation we use an R-Tree index structure described in [21]. It is a relatively simple multi-dimensional index structure, while its performance is comparable to the more complex index structures available [36]. To support fast insertions we use the *linear spit* method [21] when splitting nodes. However, this splitting method can lead to overlapping bounding boxes and a decrease in search performance. A prominent variant of the R-Tree, the R*-Tree [3], employs a set of carefully designed heuristics for node splitting to reduce the decrease in the search performance. However, the cost of splitting is high. There are other variations of the R-Tree index such as the LR-tree [4], that support fast updates. Also, LR-trees [4] do not degenerate with updates and give updated performance approximately equal to that of R*-Trees [3].

4.2 Delta Calculator

Each time the front end submits a query q, the backend searches the cache to find all the objects that lie in the brush. Given this list the backend computes the remainder query ($q_{\triangle}$), the query to be sent to the database to fetch the objects not in the cache. Next, we will explain why computing the query $q_{\triangle}$ is always possible. The computation is based on the property described below.

Let $\{b_1\}$ represent the set of nodes contained in brush b_1 having focus extents at (b_{min}, b_{max}) and $level - of - detail = lod$. A node n with horizontal extents (α, β) in the brush b_1 can be used to divide the brush into two disjoint brushes, b_2 and b_3, such that $\{b_2\} \cup n \cup \{b_3\} = \{b_1\}$, where b_2 has horizontal extents (b_{min}, α), $level - of - detail = lod$ and b_3 has horizontal extents (β, b_{max}), $level - of - detail = lod$. This property is based on the intuition that the horizontal extents of the brush and the node n both define an interval. Therefore we can divide the interval of brush b_1 into two disjoint intervals such

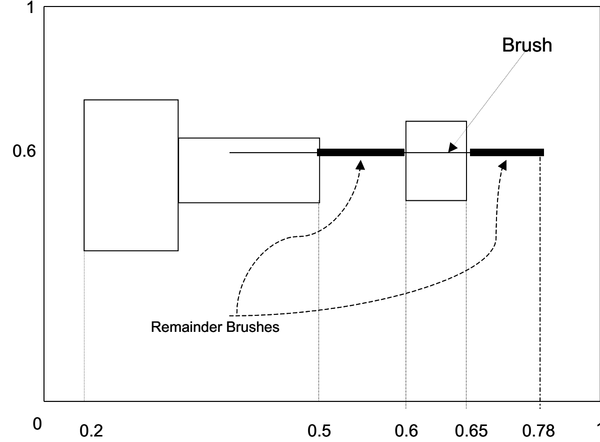

Figure 4.2: Computing Δ queries. The line segment represents the brush. $(b_{min} = 0.3, b_{max} = 0.78, lod = 0.6)$

that the result of the union of these two intervals b_2 and b_3 with the interval of n gives us the interval of brush b_1. Thus, to compute the remainder query we need to find the nodes in the cache that belong to the current brush. We can then check to see what parts of the brush interval are not occupied by the nodes in the cache. Each of these unoccupied intervals forms a remainder brush and a part of q_Δ.

Figure 4.2 gives an example of the above process. The brush has features $(b_{min} = 0.3, b_{max} = 0.78, lod = 0.6)$. The figure shows the 2D hierarchy map of the contents of the cache. The bold part of the brush in Figure 4.2 illustrates the remainder brushes. Algorithm 2 shows the procedure to compute a list of remainder brushes.

Input variables to the algorithm 2 are:

- *nodeList*, a list of nodes in the cache that lie in the current brush and
- *currentBrush*, contains the left extent, right extent and the *level-of-detail*

Output is

- *resultList*, containing a set of remainder brushes.

To summarize the working of the the algorithm 2. 2 sorts the list of cached nodes in ascending order of their horizontal extents. Then starting from the top it compares the

Algorithm 2 Delta Calculator

procedure $CalDelta(nodeList, currentBrush, resultList)$

1: Sort nodes in the nodeList in ascending order of left extents.
2: $tempE1 \leftarrow currentBrush.leftExtent$
3: **while** $nodeList$ not empty **do**
4: $\quad n \leftarrow nodeList.removeFirst()$
5: $\quad$ **if** $tempE1 < n.leftExtent$ **then**
6: $\quad\quad rb \leftarrow RemainderBrush(tempE1, n.leftExtent, currentBrush.lod)$
7: $\quad\quad resultList.insert(rb)$
8: $\quad$ **end if**
9: $\quad tempE1 \leftarrow n.rightExtent$
10: **end while**
11: **if** $tempE1 < currentBrush.rightExtent$ **then**
12: $\quad rb \leftarrow RemainderBrush(tempE1, currentBrush.rightExtent, currentBrush.lod)$
13: $\quad resultList.insert(rb)$
14: **end if**

extents of adjacent nodes in the list to find if they are contiguous in the 2D space. In case a gap exists between adjacent nodes a remainder brush is generated that will fetch the nodes to fill this gap.

4.3 Cache

Using main memory (cache) to store frequently used data items to reduce fetch latencies from secondary storage devices is a proven technique that is used in both the database and the systems context. Analysis of real user traces of our visualization environment done in [15] has shown that user traces exhibit characteristics such as:

1. Locality of Exploration: Users doing data exploration explore one area of the display at a time before moving on to another area.
2. Contiguous queries have similar answers: Exploration using visual navigation tools such as sliders and knobs translate to consecutive queries and the answers to these queries have a significant number of objects that are common.

3. Incremental user movement: User explorations using tools such as sliders are generally incremental that is fine grained, i.e., the users usually don't make any sudden big movements.

4. Presence of idle time: Users usually pause to understand the display and look for patterns in the data. So there is idle time between queries to the database.

5. User directionality or inertia: When using interactive navigation tools such as scroll bars for data exploration, it is likely that once started the user will navigate in the same direction for a while before changing to another direction.

Note the properties (1) and (2) correspond to the concept of spatial and temporal locality respectively. This suggests that the visual exploration paradigm is a good candidate for caching and if done correctly we can achieve considerable gains in performance with limited memory.

To exploit these characteristics we employ caching to reduce our response time and to avoid database fetches whenever possible. The cache in XmdvTool is a contiguous chunk of main memory. Each cache entry contains a cluster (node) from the cluster tree and a descriptor that describes the position of the node in the 2-D hierarchy map (Section 3.2). Fig 4.3 shows a snap shot of the state of the cache during execution mapped to the 2D space. Note the size of the cache is smaller than the size the of the data set. Thus the cache does not contain all the nodes in the data set. This is shown in the 2D hierarchy map using empty regions.

4.3.1 Cache Replacement

We use two replacement policies namely, LRU (Least Recently Used) and the Distance replacement policy exploiting temporal and spatial locality, respectively. The evaluation section compares the effectiveness of each of the policies.

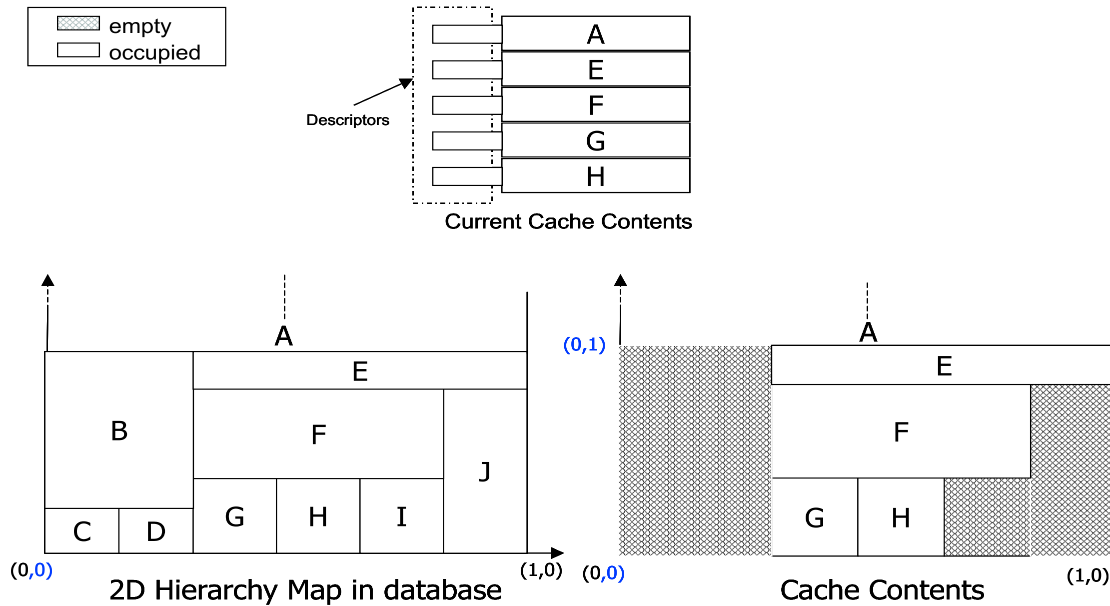

Figure 4.3: Snap shot of the cache state

LRU replacement policy replaces the object that has been unused for the longest time. This way it mainly exploits temporal locality. Note the user traces do exhibit temporal locality as shown in the previous section. Mainly, contiguous user queries having similar answers correspond to the concept of temporal locality. To realize the LRU replacement policy we maintain a linked list containing pointers to all the cache entries in the main memory. This list is called the LRU list. Each cache entry has a corresponding node in the LRU list. To keep track of this node each cache entry stores a pointer to its respective node in the LRU list. The head of the LRU list is the least recently used entry and the tail is the most recently used entry. Thus, the head of the list is always the next candidate for replacement. Each time a cache entry is selected as a member of the current user selection the corresponding LRU list entry is moved to the end of the list.

An alternative to using recency information for determining replacement candidates is to use *semantic distance* [12]. Intuitively we can say that the entry in the cache that is the furthest away from the current brush has a less chance of being referenced in the near future, as compared to the ones that are closer to the current brush. This is because, with

visual interface tools such as sliders and knobs the user movements are incremental and not random. Thus, the main idea here is to replace the entry in the cache that is furthest away from the current brush. The distance measure can be as simple as the length of the line from the center of the current brush to the center of the cache entry (node) in the 2D hierarchy map.

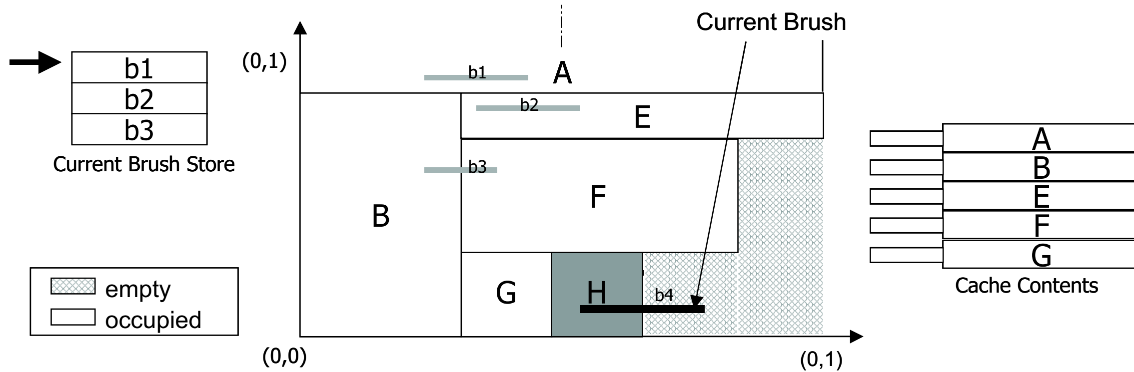

Figure 4.4: Snap shot of cache contents before replacement

We describe one way of realizing a replacement policy that replaces objects based on their distance from the current brush. In our implementation, the replacement policy maintains a lists of recent user requests in the form of the brush descriptors. This list is called a *brush store*. Each brush descriptor consists of the horizontal extents (e1,e2) and the *level-of-detail* of the brush. Figure 4.4 shows a snapshot of the state of the system during execution. The upper left corner in Figure 4.4 shows the state of the brush store. When we need to make room in the cache for new objects, the replacement policy iterates through the contents of the brush store to search for the brush *b* that is the furthest away from the current brush. The replacement policy then searches through the cache to find the contents of the brush *b* and replaces each node in the brush with a new node. If all nodes within the brush have been replaced the brush is then removed from the brush store.

Figures 4.4 and 4.5 show an example of this process. The brush store initially contains 3 brushes b1,b2 and b3 as shown in Figure 4.4. Assume that the cache is full. The current

brush b4 requires us to fetch a new node (I) from the database. To insert this node into the cache, the replacement policy selects the brush b1 to replace because it lies the furthest away from the current brush. The brush b1 only contains one node 'A'. Therefore the cache entry containing 'A' is selected as a replacement candidate and the entry is filled with the contents of new node 'I' as shown in Figure 4.5.

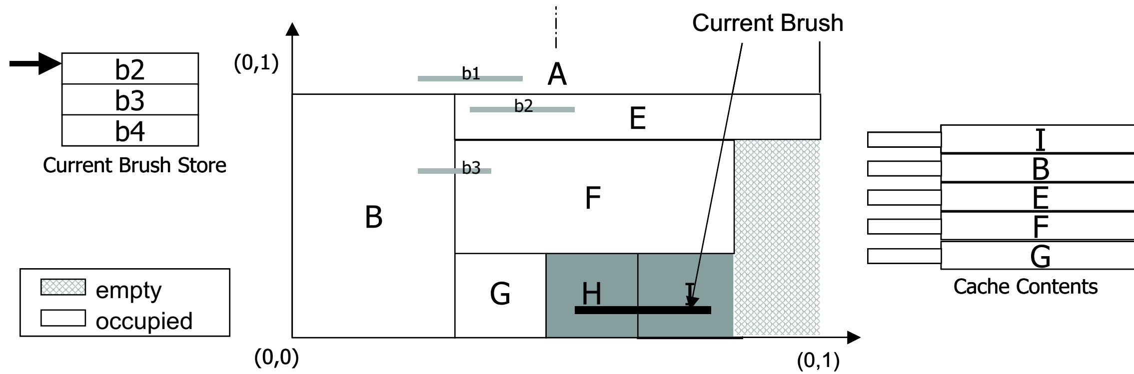

Figure 4.5: Snap shot of cache contents after replacement

Figure 4.6 shows another scenario where the contents of the brush in the brush store overlap. In such a scenario it is possible that we have already replaced all the contents of a particular brush, because one or more brushes around this brush were selected as candidates for replacement. Such a brush should ideally not occupy space in the brush store. To remove such brushes we examine the brush store periodically (during idle time) and delete them from the brush store. The process of examining involves searching through the cache to find the set of cached nodes for each brush in the brush store. If the set is empty then the corresponding brush can be removed from the brush store.

4.3.2 Direction-based Prefetching

To further improve the performance of subsequent user operations, XmdvTool uses a direction-based prefetcher proposed in [16]. The prefetcher mainly exploits the idle time in between user operations and user directionality to predict and fetch future user requests.

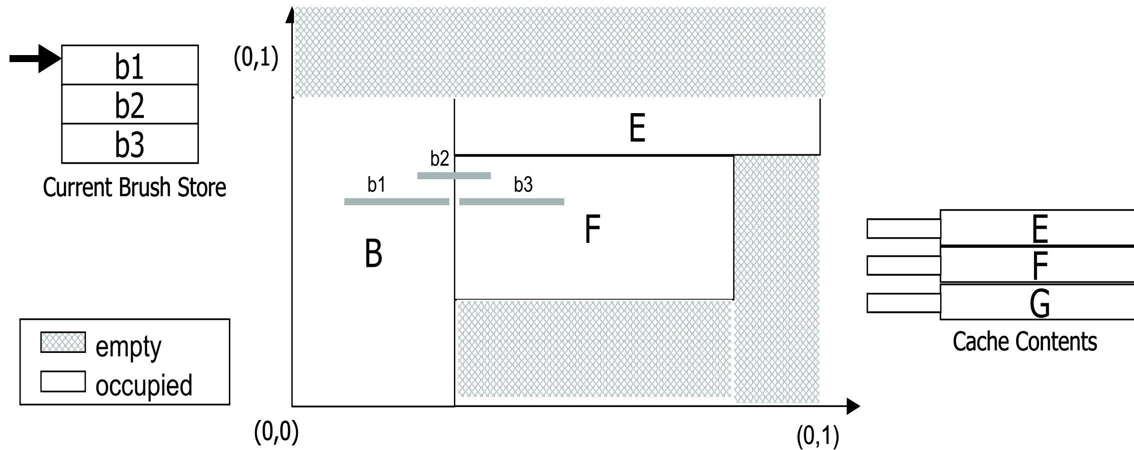

Figure 4.6: Example of brush store with overlapping brushes

The direction-based prefetch strategy is analogous to the sequential prefetching strategy proposed in other prefetching papers [11, 33]. The *direction strategy* assumes that the most likely direction of the next user movement can be determined. It is intuitive for instance that the user will continue to use the same navigation tool and move in the same direction for a while before changing either of them. We call this *user inertia*. Thus, based on the user's past explorations, the predictor computes the last direction of user movement. The prefetcher then issues a prefetch request with the brush moved in the same direction as that of the last brush movement.

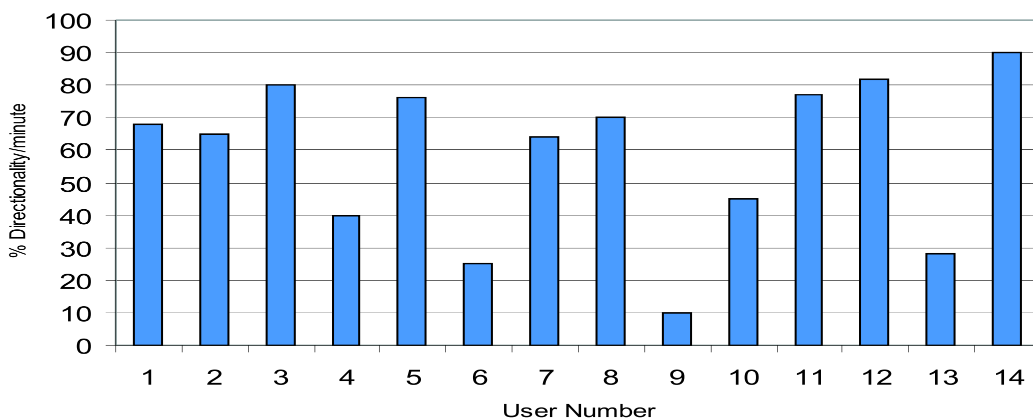

Figure 4.7: Analysis of User Traces

To compute the last direction of the user movement the prefetcher has to keep track of only the previous two brush requests. Each brush request consists of the focus extents (e1,e2) and the *level-of-detail*. To determine the direction of the user movement the prefetcher simply compares the two brushes to find which brush handle has been moved and in what direction.

To explore the extent of directionality exhibited in user traces, our prior work in [15] measured the percentage directionality per minute for a set of 15 real user traces. Figure 4.7 presents the results of the analysis. The horizontal axis lists the user number assigned to the user trace. From Figure 4.7 we can see that most of the real user traces do exhibit a high degree of directionality. On average the directionality per minute is around 60%.

Chapter 5

System Implementation

5.1 System Architecture

The complete system has been implemented as an extension to XmdvTool 6.0 [52, 40]. XmdvTool 6.0 was coded in C++ with TCL/TK and OpenGL primitives. Figure 5.1 depicts the main modules in the Xmdv backend. We used Oracle 9i as the database management system and Oracle spatial extension to construct the R-Tree index at the database. To communicate with the Oracle 9i server we used OTL (Oracle, Odbc and DB2-CLI Template Library [35]). The library provides an easy to use and efficient API to send queries and retrieve answers as C++ streams. The main memory R-Tree index was built using the spatial index library developed at University of California Riverside [44].

Figure 5.1 shows a more detailed version of the diagram shown in Section 4. It shows all the main components (classes) in the system that together comprise the backend of the Xmdv Visualization tool. The responsibilities of each component are explained below.

The **Backend Controller** forwards user requests to the cache manager. Each user request contains the current position of the structure-based brush, the focus extents (e1,e2) and *level-of-detail*. Moreover, at the end of each user request the backend controller sees

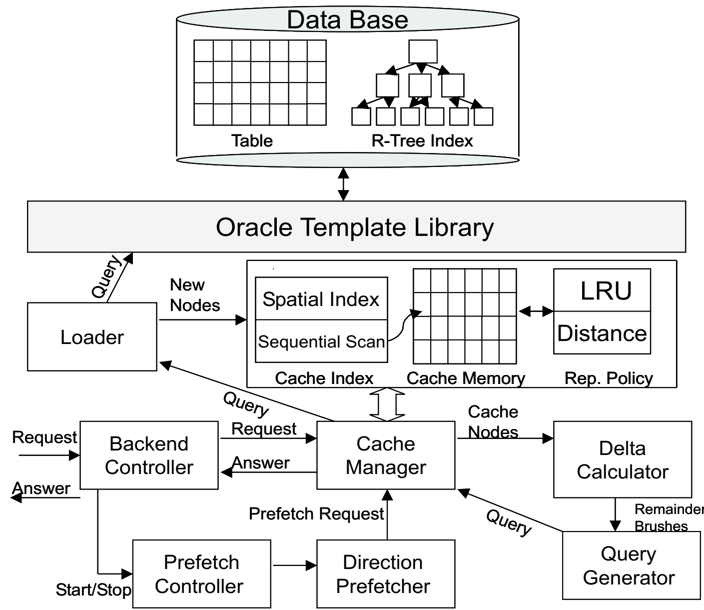

Figure 5.1: System Architecture

an opportunity to start the prefetcher. The prefetch controller is given a start signal which in turn activates the prefetch thread. If the backend controller receives a user request when the prefetch thread is active it sends a stop signal to the prefetch controller that in turn preempts the prefetch thread.

The **Prefetch Controller** activates the prefetch thread each time it gets a start message from the backend controller. Activation means giving permission for the prefetch thread to run. The permission is given by releasing a semaphore called *PrefetchPermit* that the prefetch thread is blocked on. The prefetcher in turn consumes this permission and starts running. On receipt of a stop message the Prefetch controller deactivates the prefetch thread. To do so it interrupts the prefetch thread. The interrupted prefetch thread now waits for another permission to run from the prefetch controller and thus blocks on the *PrefetchPerimit* semaphore, giving up the CPU.

The **Cache Manager** does the job of coordinating the responsibility of the various components. Each time a new request arrives from the client (user or prefetcher) it sends a search request to the cache. It then forwards the selected cache entries (nodes) to the delta

calculator. The query generated by the delta calculator is forwarded to the loader. The cache manager then loads the new nodes into the cache. It combines the result returned by the loader with the selected cache entries to form the final result set. This result set is then returned to the client by means of an iterator object.

The **Delta Calculator** computes a set of remainder brushes given a set of cached nodes and the current brush as input from the cache manager. It implements the Algorithm 2 described in Section 4.2.

The **Query Generator** generates a single SQL query for each remainder brush as shown in Section 3.4. If there is more than one remainder brush it combines their corresponding queries into a bigger query using the union operator.

The **Loader** communicates using the OTL API to retrieve the result of the query from the Oracle database server.

The **Cache** consists of the cache memory, cache index and the cache replacement policy. The cache is basically a chunk of contiguous main memory. Each cache entry stores the information about a particular node and its descriptor. The cache index can help provide faster *cache lookup*. *Cache lookup* is the task of searching for nodes in the cache that belong to the current user selection. The implementation currently supports two options for cache lookup:

- Spatial Index: Here the cache maintains a spatial main memory index, specifically an R-Tree index, to search through the nodes in the cache.

- Sequential Scan: For each user request, the contents of the cache are scanned sequentially to find cached nodes that belong to the current user selection.

Also the cache can be configured to use either the distance replacement policy or the LRU replacement policy. Thus given a search request the cache manager searches through the buffer using the chosen cache lookup mechanism and returns the selected cache entries

to the cache manager. Also each time it receives a request to load a new node it uses the replacement policy to select the next victim entry to remove. It overwrites the contents of the victim cache entry with the contents of the new node, and if necessary, updates the cache index.

5.2 WorkFlow

The **backend controller** receives a request to fetch the contents of a brush. This request is then forwarded to the cache manager. The **cache manager** finds the nodes in the cache that lie in the current brush. A list of these nodes is then forwarded to the delta computer. The **delta calculator** computes all the remainder brushes. The **query generator** generates a single SQL query that will include all the nodes in each of the remainder brushes. The **loader** executes the query and fetches the remainder brushes. The cache manager then takes the union of the cached nodes (in the brush) and the answer to the remainder query to form a complete list of nodes that lie in the current brush. All the new nodes are buffered inside the cache. The complete list of nodes (in the brush) is then sent to the front end by means of an iterator object. However as a special case if the data in the current brush cannot fit inside the cache all at once the cache manager loads the data incrementally. To do so it has to replace entries in the cache that lie in the current brush but were already served to the front end. We call this process *incremental loading*. This whole process is hidden behind the iterator interface so that the front end is not required to have any knowledge about this.

During idle times, i.e., when there is no active request, the backend controller activates the prefetcher. The **prefetcher** predicts user movements and issues prefetch requests to the backend controller. The backend controller in turn forwards these requests to the cache manager. The cache manager does not distinguish between a user request and a

prefetch request. It fetches the requested data and stores it in the cache. The prefetcher is implemented in its own separate thread; the backend controller activates the prefetch thread when there is no active user request. However, the user request has a higher priority then the prefetch request. Therefore, if the backend controller gets a user request when the prefetch thread is active the thread is preempted prematurely, thus giving the main thread all the CPU it needs.

Chapter 6

Experimental Results

6.1 Experimental Setup

All of our experiments were run on a Pentium 3 windows XP machine with 128 MB of memory running at 833 Mghz. The complete system was implemented in C++. We used the OTL oracle-odbc template library to access data on an oracle server running Oracle 9i. The oracle 9i was set up on a server running Redhat Linux 9.0 on a pentium 3 dual processor machine with each processor running at a 450 Mghz clock speed, and 512 MB of main memory. We installed the oracle spatial extension to construct the R-Tree index at the database server. For the main memory R-Tree index we used the spatial index library [44] developed at University of California Riverside.

To test the scalability of the system we used two real data sets named out5d and uvw. Out5d data set had 20,000 data points. It is a five dimensional remote sensor data set (SPOT, magnetics, and three radiometrics channels - potassium, thorium, and uranium). The Uvw dataset had 195,000 data points and 6 dimensions. It contains flow simulation data. We ran experiments over the out5d with a set of real user traces. We used a set of 4 real user traces each of half hour duration, collected as a part of the study performed in

[15]. For experiments over uvw data set we used a set of synthetic user traces. However, these traces were modeled based on the characteristics exhibited by the real user traces. [15] presents details of modeling user traces. We simulate locality of exploration in user traces using hot regions. Hot regions are places in the navigation space where the user spends most of his time. To generate the above set of user traces we declare 5 hot regions, well spaced from each other in the navigation space, and set the probability that a user request lies in any of the hot regions to 50%. We call these user traces 50% local. To simulate user inertia, we use a probability that gives the likelihood that the user keeps moving in the same direction. For the generation of the above 4 user traces we set this probaility to 50%. Thus we can say that the user moves pseduo-randomly in our experiments, i.e., some of the future possible actions are more probable than other, but choosing among these actions is still performed non-deterministically. In our last experiment we use user traces with varying locality and directionality to show how the cache responds to different types of user traces.

All the results reported in this section are an average taken over four runs.

6.2 Metrics

The main metric used to evaluate the performance of the cache is *latency*. The *latency* for a single user request is the time taken for the backend to serve the data once the request is submitted. To compute the latency for a complete user trace, we use the following formula:

$$latency = \frac{\sum_{i=1}^{N} L_i}{\sum_{i=1}^{N} T_i} \tag{6.1}$$

where N is the total number of requests, T_i is the number of objects (tuples) fetched in request i and L_i is the latency for request i. Equation 6.1 gives us the latency per object

fetched. It gives us a common ground to compare and combine the latency measures for different user traces.

A measure derived from latency is the *latency reduction ratio* (*lrr*). The *latency reduction ratio* for a particular system configuration is the ratio of the decrease in latency to the latency obtained when running the same experiment using the base configuration. In the base configuration the cache, prefetcher and the secondary index structure all are turned off so the user requests are sent directly to the database.

$$lrr = \frac{Latency_{base} - Latency}{Latency_{base}} \tag{6.2}$$

Equation 6.2 gives us the fraction of the latency reduced by a particular system configuration. This helps us to evaluate the relative usefulness of each configuration.

In addition we also use *object hit ratio* to measure the usefulness of the prefetcher and replacement policy. The hit ratio for a complete user trace is the ratio of the total objects fetched directly from the cache to the total number of objects requested by the complete user trace. Thus the hit ratio for the user trace is given by:

$$hitratio = \frac{\sum_{i=1}^{N} H_i}{\sum_{i=1}^{N} T_i} \tag{6.3}$$

where N is the total number of requests, H_i is the number of objects (tuples) fetched directly from the cache in request i and T_i is the total number of objects (tuples) requested in request i.

6.3 Database Index

The goal was to show the usefulness of the R-Tree index structure on the database server. To show this we ran two experiments. In the first experiment we ran four user traces over

data sets out5d and uvw. The cache was turned off. Each user request was sent directly to the database. We record the latency for each user trace with the database index on and off. Note that when the database index is off the system configuration is the same as the base configuration.

	No-Index	Index	lrr
User1	2.0253	1.5	0.25
User2	0.6444	0.35	0.45
User3	0.751336	0.5	0.33
User4	0.8855	0.6	0.32

Figure 6.1: Latency in msec with and without database index, out5d data set

	No-Index	Index	lrr
User1	3.90	0.94	0.75
User2	6.09	1.75	0.71
User3	2.05	0.71	0.65
User4	5.42	1.09	0.80

Figure 6.2: Latency in msec with and without database index, uvw data set

Figures 6.1 and 6.2 show the latency and also the latency reduction ratio for data sets out5d and uvw respectively. For out5d data set the latency reduction ratio on average is approximately 33%. However, for uvw data set it is approximately 70%. This is because the search time for the sequential scan increases linearly relative to the size of the data set, whereas with an index the search time increases almost logarithmically with the size of the data set. This shows that as the size of the data set increases, the benefits of the database index become more significant.

In the second experiment we measure the effectiveness of the index at the database with the cache turned on. The cache size is set constant 10% of the size of the data set for out5d and to 2% for uvw. Note that uvw is around 10 times larger than out5d. However, the cache size does not have to scale with the size of the data set. Therefore using these

settings we also plan to show that for the system to give reasonable or a better latency reduction ratio the cache size does not have to scale with the size of the data set. In this experiment we again expect to see that the gain in the latency reduction due to the index for the bigger data set (uvw) will be larger as compared to the gain for the smaller data set (out5d). Again the reason being the same as the previous experiment, i.e., the amount of time we save by using the index over the bigger data set (uvw) is greater than the amount of time we save by using index over the smaller data set. And since we are using the latency reduction ratio as the measure, this difference is reflected in the output.

	lrr (no-index)	lrr(index)	Δlrr
User1	0.50	0.68	0.18
User2	0.52	0.62	0.1
User3	0.25	0.52	0.27
User4	0.36	0.5	0.14

Figure 6.3: Latency reduction ratio with relative cache size 10 %, out5d data set

	lrr (no-index)	lrr (index)	$\Delta lrr.$
User1	0.73	0.95	0.22
User2	0.74	0.94	0.2
User3	0.70	0.91	0.21
User4	0.79	0.95	0.16

Figure 6.4: Latency reduction ratio with relative cache size 2%, uvw data set

Figures 6.3 and 6.4 show the *lrr* for four user traces using the data sets out5d and uvw respectively. Note that the Δ *lrr* is significant. Thus, the spatial index is beneficial. On average for out5d data set we gain approximately 17% and for uvw data set we gain approximately 20%. As the size of the data set increases the Δ lrr increases. The reason again is that the search time when using the index increases at a lower rate as compared to using the sequential scan. Thus the difference in the amount of time it takes to search for remainder queries on the database server for the two approaches will increase as the size

of the data set increases. This shows that the database index makes our system scalable.

Furthermore, we can also see that for out5d data set the latency reduction ratio on average is approximately 58% and for uvw data set it is approximately 94%. Recall that, the relative cache size for out5d data set is 5 times more than uvw data set. The reason being, the amount of time saved due to a cache hit for the bigger data set (uvw) is larger when compared to the amount of time saved due to a cache hit on the smaller data set (out5d) because the cost of searching through the bigger data set is higher than the cost of searching through the smaller data set. Thus, we can conclude that the size of the cache does not have to scale with the size of the data set.

6.4 Cache Size

Here we show the effect of the cache size on the latency for data sets out5d and uvw. For this experiment we turn on the cache and the index structure in the database. The cache uses LRU as the replacement policy and the prefetcher still remains off.

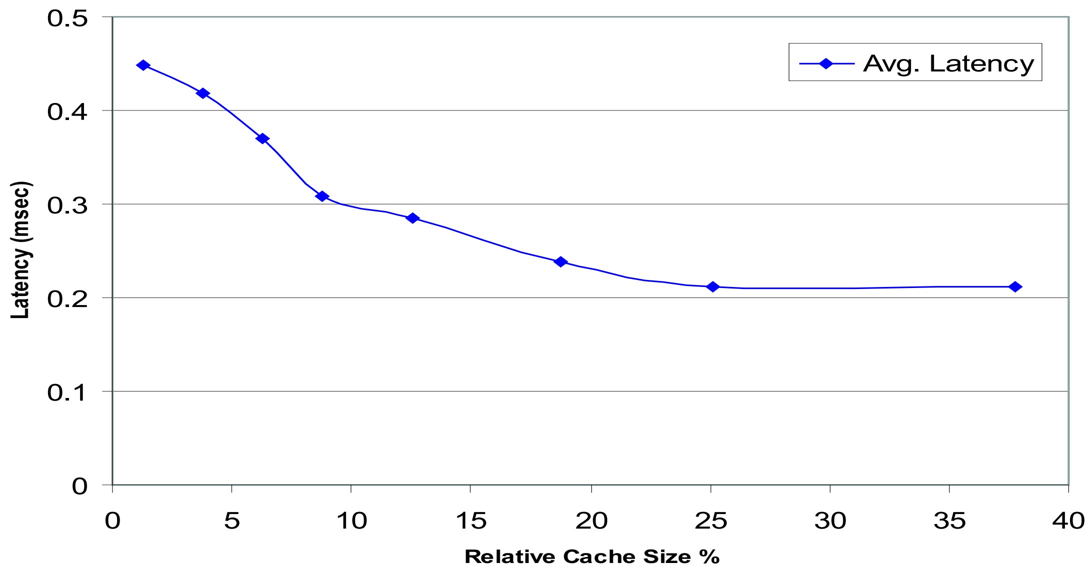

Figure 6.5: Comparison of cache size vs. average latency, out5d data set

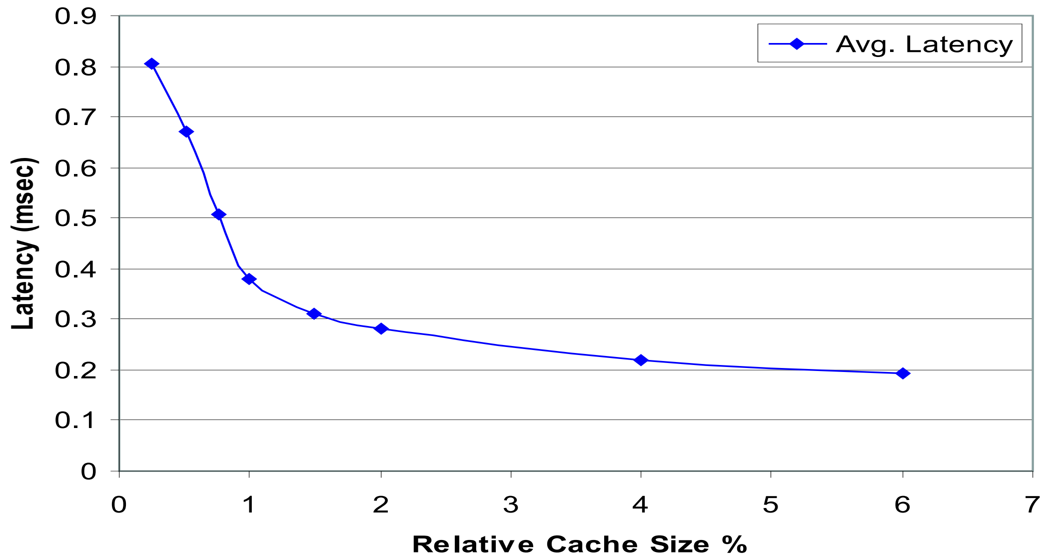

Figure 6.6: Comparison of cache size vs. average latency, uvw data set

We ran the same four user traces and four runs of each of them. In Figures 6.5 and 6.6 we plot the Relative Cache Size vs. Average Latency for all the user traces. As seen in the results, the Latency decreases at a high rate for smaller cache sizes. However, the curve flattens out and we get less gains for bigger cache sizes.

Latency is inversely correlated with hit ratio. For the user traces used in this experiment it appears that for big cache sizes the only misses we get are *compulsory misses*. *Compulsory misses* occur when the cluster is accessed for the first time by the user trace. *Compulsory misses* are independent of the cache size, at least until the prefetcher is inactive. Thus we can see that increasing the relative cache size from 20%-40% for out5d data set results only in little improvement in latency. Also note that the curve flattens out much earlier for the uvw data set when compared to the curve for data set out5d. This also shows that the size of the cache does not have to scale with the size of the data set. This property makes the system scalable for huge data sets.

6.5 Comparison of Replacement Policies

Here we compare the performance difference between the new replacement policy we have designed Distance, versus the well know LRU replacement policy. The prior is based on exploiting spatial locality whereas the latter exploits temporal locality in user traces to maximize hit ratio. We ran the same four user traces used in previous experiments for both out5d and uvw data sets. The experimental setup remains the same as in the previous experiment.

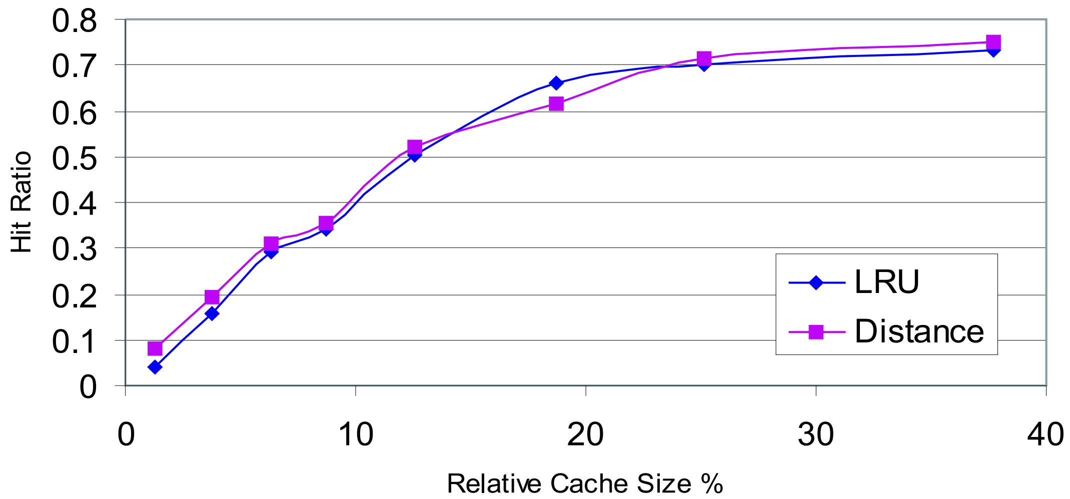

Figure 6.7: Cache size vs. hit ratio for Distance and LRU replacement, out5d data set

Figures 6.7 and 6.8 show the average hit ratio and average latency reduction ratio for the out5d data set. Figures 6.9 and 6.10 show the average hit ratio and average latency reduction ratio for the uvw data set. From the charts in Figures 6.7 and 6.9 we can see that in most cases the Distance replacement policy gives a higher hit ratio. In fact for the uvw data set the distance replacement policy gives a consistently higher hit ratio and at some points the difference is as much as 7%. We also get an improvement in latency reduction ratio of approximately 2% at some points as shown in Figure 6.10. So from the charts we

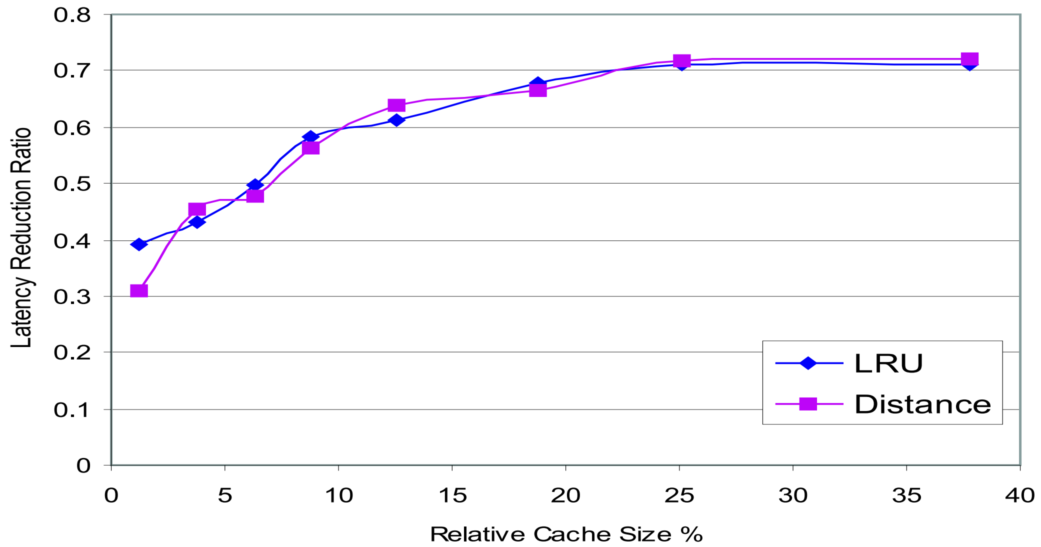

Figure 6.8: Cache size vs. *lrr* for Distance and LRU replacement, out5d data set

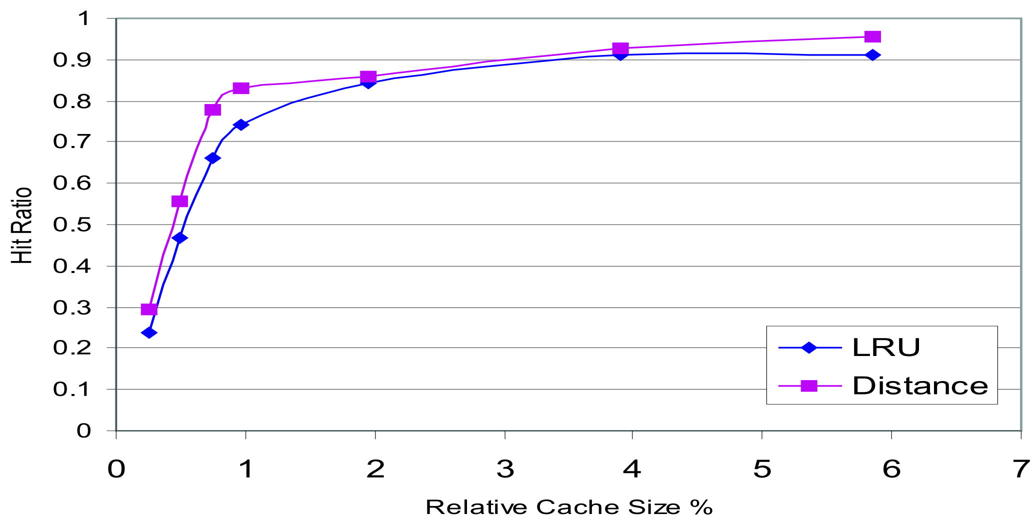

Figure 6.9: Cache size vs hit ratio for Distance and LRU replacement, uvw data set

can say that the Distance replacement policy performs at least as well if not better than the LRU replacement policy for the user traces we are using. This confirms that the collected user traces do exhibit a high degree of spatial locality and thus this characteristic can be

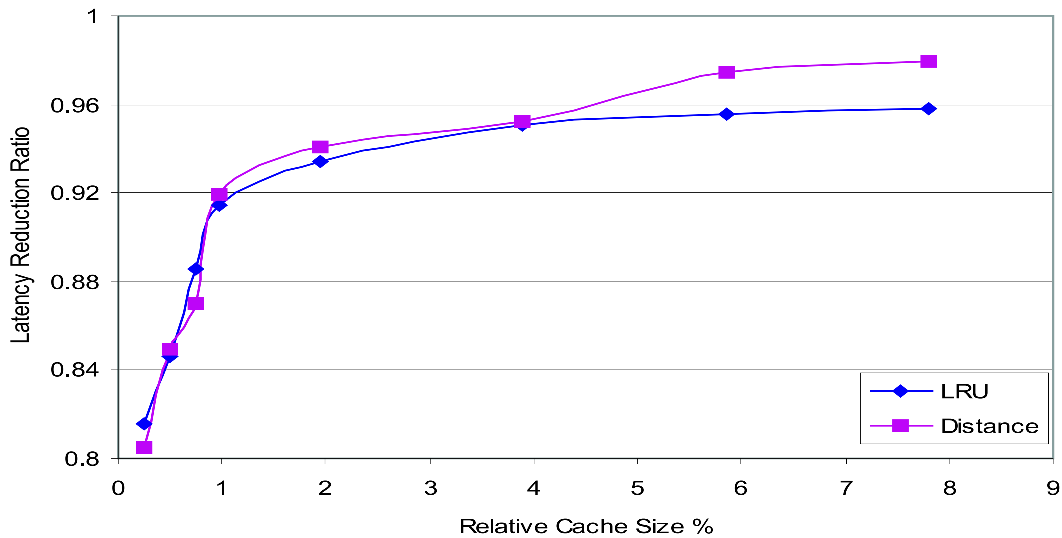

Figure 6.10: Cache size vs. *lrr* for Distance and LRU replacement, uvw data set

effectively exploited for cache replacement.

6.6 R-Tree Cache Index

Here we show the effectiveness of the R-Tree main memory index for cache lookup. To show this we ran an experiment with the R-Tree main memory index structure turned on and off. The spatial index on the database is turned on, the cache is configured to use the LRU replacement policy and the prefetcher is turned off. When the R-Tree main memory index is turned off, we revert back to using the sequential scanning to find the requested objects in the cache. We again used the same four user traces as in the previous experiments over data sets out5d and uvw.

Figure 6.11 shows the average latency for the four user traces for the out5d data set. From the figure we can see the cache index performs worse than the sequential scan for small cache sizes. The main reason is that for small cache sizes the hit ratio is low. This means that cache contents are changing very frequently. Thus, the R-Tree index has to be

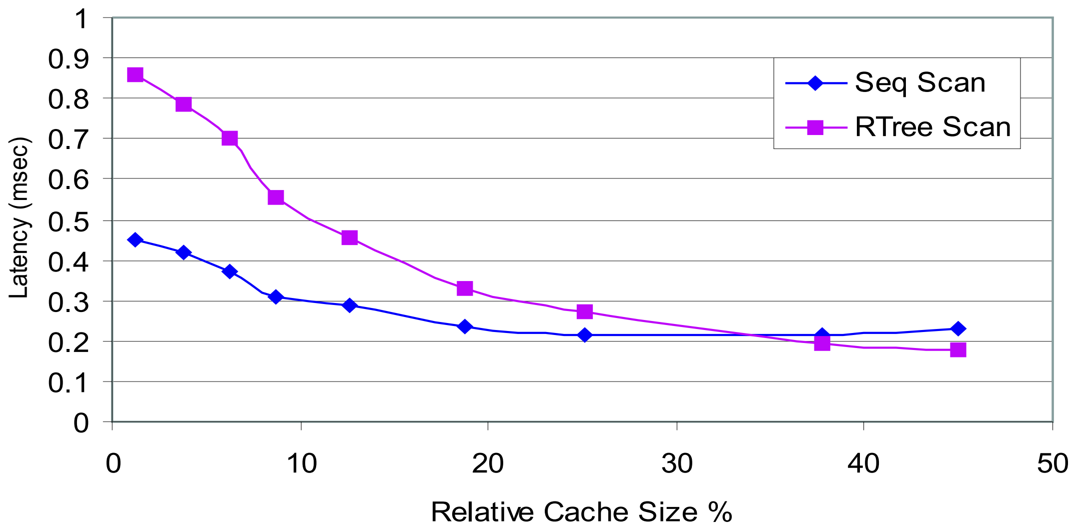

Figure 6.11: Comparison of cache size vs. latency with and without main memory R-Tree index, out5d data set

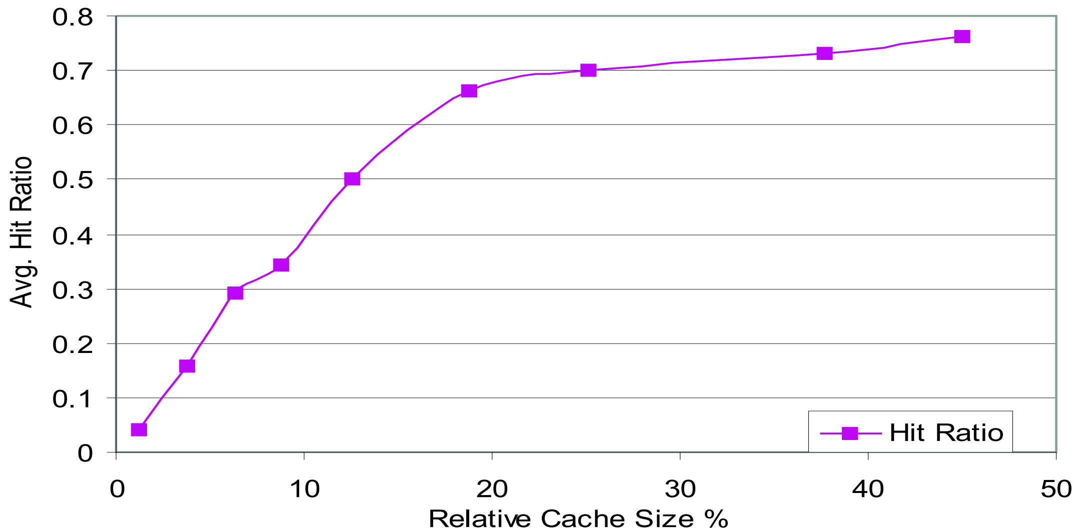

Figure 6.12: Comparison of cache size vs hit ratio, out5d data set

updated very frequently. Also, since our implementation uses the *linear split* method for insertions the quality of the R-Tree index can degenerate quickly with frequent updates. Moreover, the cost of updating the R-Tree index is high as compared to practicaly no

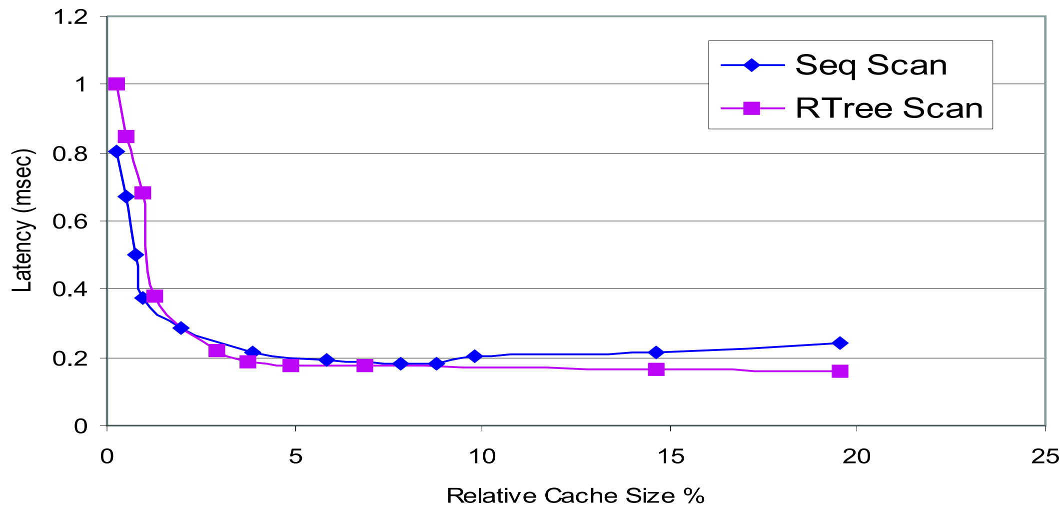

Figure 6.13: Comparison of cache size vs latency with and without main memory R-Tree index, uvw data set

update cost when considering the sequential scan.

However, for bigger cache sizes the R-Tree curve does cross-over and gives lower latency values. Figure 6.12 shows the hit ratio for the same experiment. Note there is only one curve because the hit ratio for both the R-Tree index and sequential scan is the same. The only difference really is in the cache look up latency. If we compare Figures 6.11 and 6.12 we see that around the 80% hit ratio mark the R-Tree starts performing better than sequential scan. Figure 6.13 shows the results of the same experiment over the uvw data set. The R-tree curve crosses over the sequential scan curve when the relative cache size is around 5% mark. The hit ratio for this point is around 85%.

From the results we can conclude that the index structure can be helpful in certain situations. The effectiveness of the main memory spatial index largely depends on the cache size which in turn determines the hit ratio. With the increasing size of RAMs in modern computers the users can allot more cache size and in such cases the main memory index become useful. However, to make the index work for lower hit ratios we may

implement an index such as the LR-Tree [4] or Grid File [34] that supports high update rates as well as high query rates. This is however beyond the scope of this thesis.

6.7 Prefetcher

The goal of this experiment is to show the effectiveness of prefetching. The prefetcher uses time in between user requests to fetch data that has high probability of being requested in the near future. To show its usefulness, we ran an experiment using 4 user traces over data sets out5d and uvw. We turn on the cache, the index structure on the data base and use sequential scan as the cache look up policy. We plot the charts with the prefetcher on and off to show its usefulness.

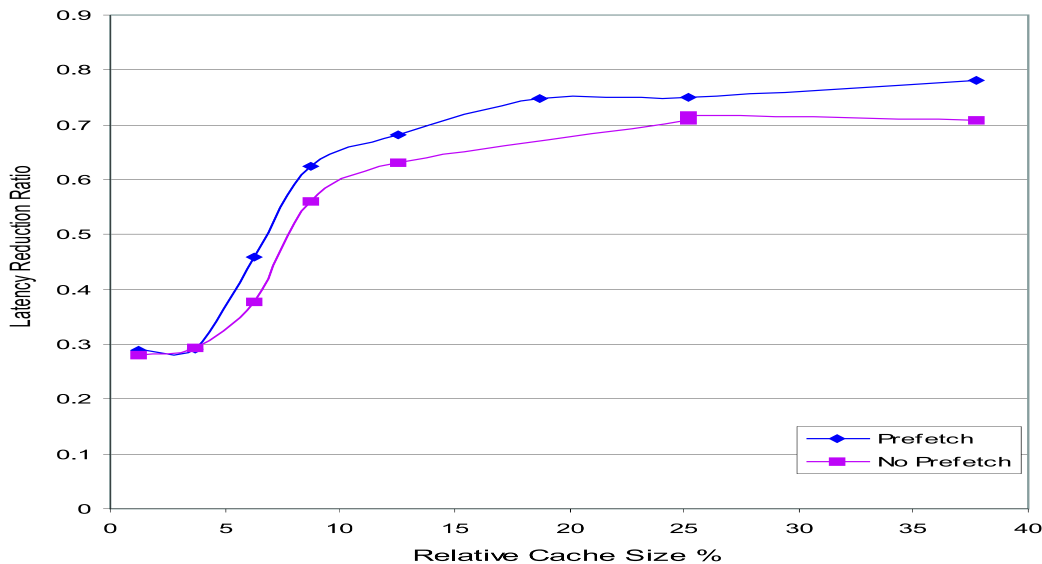

Figure 6.14: *lrr* of prefetch vs. no prefetch, user3, out5d data set

Figure 6.14 shows the *lrr* with and without the prefetcher for user3. Here we can see that the prefetch curve is above the no prefetch curve for practically all cache sizes. This shows that the prefetcher improves the performance of the cache by around 8% on average. This can be directly attributed to the improvement in hit ratio. Figure 6.15 shows

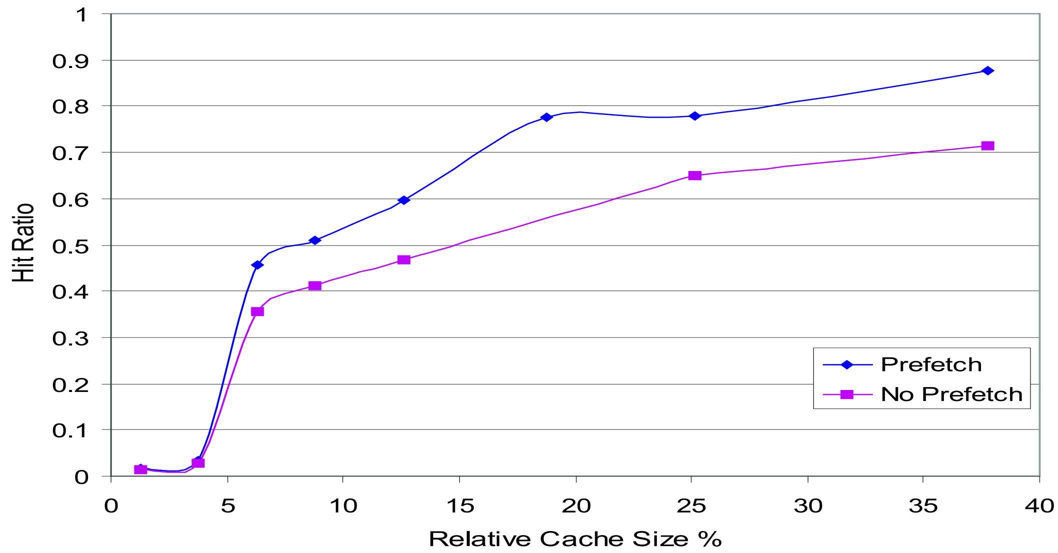

Figure 6.15: Hit ratio of prefetch vs. no prefetch, user3, uvw data set

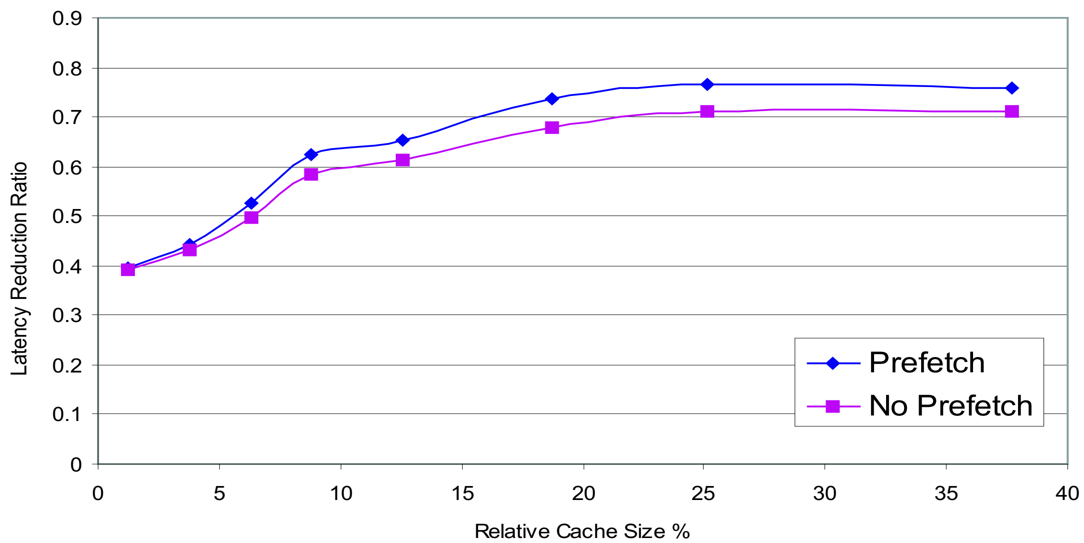

Figure 6.16: Average *lrr* for 4 user traces, prefetch vs. no prefetch , out5d data set

the hit ratio for the same user trace with and without the prefetcher. The curves for the remaining user traces also show the same trend. Figures 6.16 and 6.17 show the average *lrr* for the four user traces for data sets out5d and uvw respectively. From the figure we

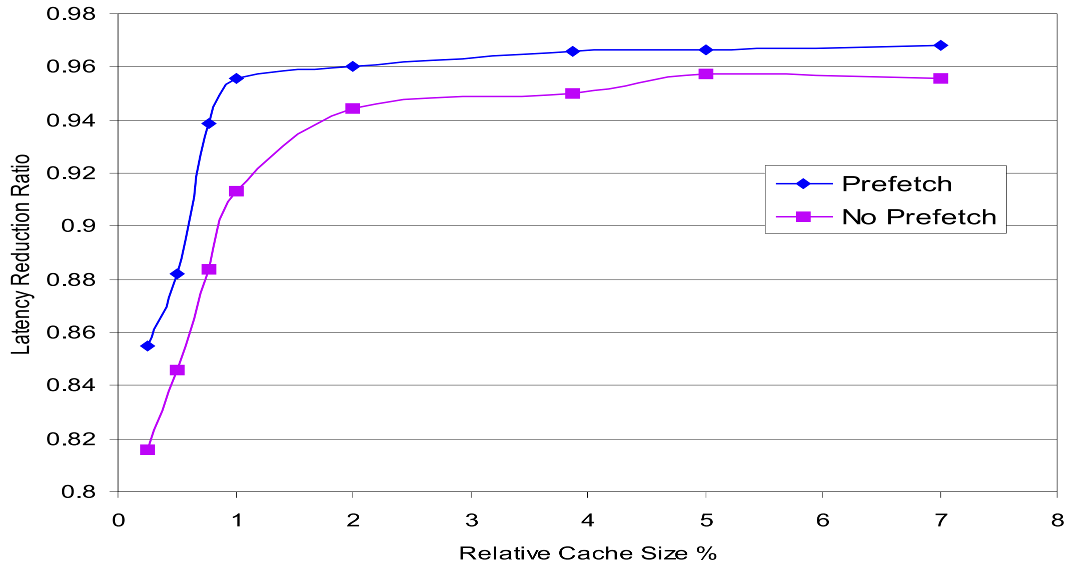

Figure 6.17: Average *lrr* for 4 user traces, prefetch vs. no prefetch , uvw data set

can see that enabling the prefetcher increases the *lrr* by around 4% on average for out5d data set and 3% on average for uvw data set.

6.8 User Trace Analysis

The goal here is to show how the cache responds to different types of user traces. To show this we record the cache hit ratio for different types of user traces. We vary 2 parameters in the user traces, the locality and the directionality. For this experiment the spatial index at the data base is turned on, the cache is turned on and the prefetcher is turned off.

In the first experiment we test the performance for the cache for user traces with varying locality. We set the directionality of the user trace to 50%. That means there is a 50% chance that the user will move in the same direction as that of the last user movement. User traces with x% locality means there is x% chance that the user will be in one of the hot regions. As described in Section 6.1 hot regions are used to simulate locality of exploration. Thus we declare 5 hot regions, well spaced from each other in the

navigation space.

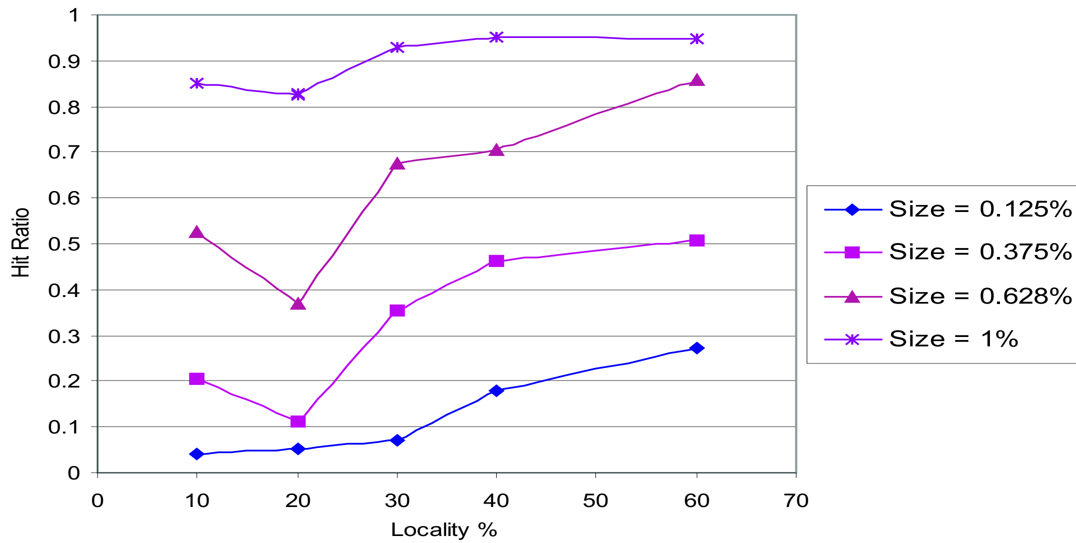

Figure 6.18: Localtiy % vs. hit ratio, uvw data set

Figure 6.18 shows the results of the experiment for different cache sizes. From the figure we see that as locality increases the hit ratio also increases. This is mainly because with increasing locality the chances that the user revisits a particular node also increase and thus the hit ratio also increases. Also, as the cache size increases the curves shift upwards. This confirms the effect we see in Section 6.4.

In the second experiment we test the performance of the cache for user traces with varying directionality. We set the locality of all the user traces to 50%. Other than that, experiment setup stays the same as in the previous experiment.

Figure 6.19 plots directionality versus hit ratio for different cache sizes. From Figure 6.19 we do not see any direct correlation between directionality and hit ratio. This is mainly because our caching (replacement) strategy does not exploit the directionality of user traces. However, the curves in this experiment also shift upwards with an increasing cache size. This again confirms the effect we see in Section 6.4.

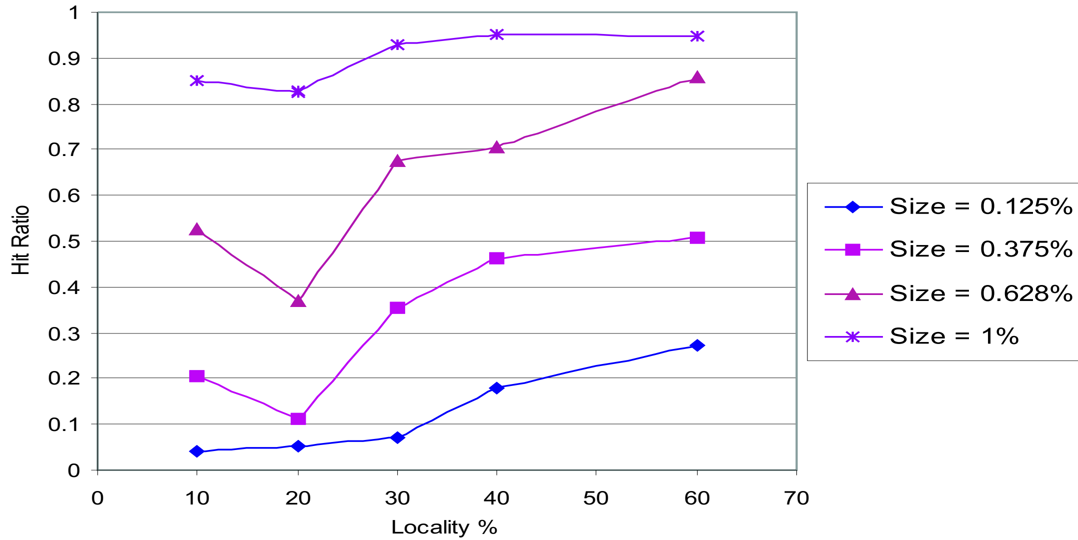

Figure 6.19: Directionality % vs. hit ratio, uvw data set

6.9 Discussion

The experiments have demonstrated that each component of the framework plays a significant part in reducing the response time of the system. Also each of the methods scales, i.e., perform relatively better with bigger data sets. Thus our proposed framework that includes a hierarchy encoding technique, caching and a prefetching strategy work in synergy to exploit characteristics of visual visual exploration environments such as locality of exploration, temporal locality in user traces and user inertia to support visual exploration operations over large data sets. Due to this success, our framework will be released with the next version of the freeware XmdvTool [54].

Chapter 7

Related Work

7.1 Visual Hierarchy Exploration

There has been considerable research in the visualization area toward finding effective methods to display and explore hierarchical information, such as Tree-Maps [42], Cone-Trees [38] and Reconfigurable Disc Trees [24]. Most of these methods provide only modest modes of interaction for navigating the hierarchy. Navigation plays an important role in aiding users to find their way through the complex structure: to see where they are, what information is available and how to identify information of interest.

On the other hand, techniques for visual exploration of hierarchies [30] have independently been proposed. Hierarchy visualizations are evident, for instance, in many commercial applications, such as Microsoft Windows Explorer, Norton Commander, and so on. The major disadvantage of such interfaces however is that there is a limited display space for the hierarchy. Hence, they are not suitable for displaying large data sets. The visualization technique we use in this work [19, 20] has both the capability of interactively navigating the hierarchical structures and the capability of displaying large datasets.

7.2 Visualization-Database Integrated Systems

Database support systems for visualization systems such as Polaris [45], Tioga [46], ADR [28], USD [25] and IDEA [41] represent work closely related to ours in the sense that all of them work towards making visualization systems run over large persistent data. However, practically all of the detailed techniques used are rather different in each of these systems.

Polaris [45] is an interface for exploring multi-dimensional databases that extends the well-known Pivot Table interface. Similar to our structure-based brush, Polaris includes an interface for constructing visual specifications of table-based graphical displays and the ability to generate a set of relational queries for the visual specifications. But Polaris does not study the effect of caching and prefetching to improve performance.

ADR [28] provides a general framework to provide support for applications that employ range queries on large scale multi-dimensional data sets. ADR targets a distributed memory architecture with one or more disks attached to each node. However in our case we propose a framework specifically optimized for navigation of hierarchies.

USD [25] proposes a semantic net model to store and retrieve unstructured data. Instead, in our case we assume the data is structured in a hierarchical fashion and specifically optimize the system to store and retrieve hierarchies. Tioga [46] proposes a user interface paradigm in which the user can construct a visual query (recipe) to retrieve and process information for the purpose of data visualization. However, the problem of query translation is not present since the user directly specifies the query using the interface.

IDEA [41] is an integrated set of tools to support interactive data analysis and exploration. Similar to the XmdvTool the tool focuses on integrating multiple display views, but on-line query translation and memory management are not addressed. In sum this work is unique in that it focuses on the issue of what core database techniques can be

utilized to effectively exploit and thus support visual structural exploration environments.

Other systems that have a visual interface and a database back-end include dynamic query histograms [13] and direct manipulation histograms [22]. However, the operations translate differently: to dynamic range queries in [13] and to temporal queries in [22]. Neither deals with hierarchy exploration support nor with caching or prefetching.

7.3 Hierarchy Encoding

Our approach toward hierarchy labeling is related to the nested interval [51] method and the nested set [6] method. The Nested interval method generalizes the nested set method [6]. The interval boundaries do not have to be integers; they can be rational or even real numbers. The labels assigned to the node by the nested interval approach are similar to the horizontal extents. We augment this labeling scheme with labels for vertical extents to incorporate the vertical selection semantics of the structure-based brush in our labeling scheme. Furthermore our work goes beyond hierarchy encoding. We propose a method that allows us to map the encoded hierarchy to a 2D space and the navigation operations to spatial queries. Moreover we propose a framework to exploit the encoding scheme which includes spatial indexing, caching, spatial replacement and prefetching.

Ciaccia et al. [8] used the mathematical properties of *simple continued fractions* for encoding tree hierarchies. Basically, each node of the tree has a unique label that encodes the ancestor path from that node up to the root. The trees are assumed to be ordered (i.e. children have order numbers) so that the ancestor paths simply correspond to a sequence of integers. The sequence gives us the code of the ancestors of a node without any physical access to the data. This information is sufficient for performing some operations, such as getting the first common ancestor of 2 nodes or testing if a node is the ancestor of another one, without any recursive retrieval of data. However, given a node n, this method cannot

efficiently provide the list of descendants of n. This limitation reduces the number of operations that can be supported.

A similar idea was introduced by Teuhola [50] who used a so called *signature* for encoding the ancestor path. Given a node n, the code of n is obtained by applying a hash function to it and by concatenating the resulting value with the code of its parent. The non-unique code can make the number of tuples retrieved be much larger than needed. Moreover, the code obtained by the concatenation of all ancestor codes could exceed the available precision for deep trees.

The above encoding schemes and many others come under the category of prefix-based encoding schemes. [26] trys to optimize space requirements of labels assigned by the encoding scheme. It derives the space requirements of nested interval encoding schemes and suggests a prefix-based encoding scheme with lesser space requirements. The encoding scheme is based on assigning binary strings to each edge and the label assigned to a node is derived from concatenating the strings assigned to the edges in the path from the node to the root. They show that in most cases, this scheme reduces total size of the labels by around 10%. However, the scheme suffers from bad worse case guarantees, that is for a tree with a long path the labels can get very long. There is no clear winner and this is still an open research topic. Moreover, in this thesis we do not focus on minimizing the size of the labels. This can be one possible area of future research.

Recently, there has been considerable work in the area of hierarchy representation for XML indexing and for retrieval of XML documents. [31] extends the idea proposed in Dietzs numbering scheme [14] to support dynamic insertions with recomputation of labels. However, in our case we assume that the data is static. The main idea in [14] was to use tree traversal order to determine the ancestor-descendant relationship between any pair of tree nodes. Each node is labeled with a pair of preorder and postorder numbers. In the tree, we can tell node (1,7) is an ancestor of node (4,2), because node (1,7) comes

before node (4,2) in the preorder (i.e. ,1 $<$ 4) and after node (4,2) in the postorder (i.e., 7 $>$ 2). This scheme does not incorporate the vertical selection semantics of the ALL and the ANY brush. Moreover, unlike the labels assigned by our scheme, the labels assigned to the nodes do not easily map to a 2D space. Thus the structure-based brush selection cannot be expressed as simple spatial intersection operations as in our framework.

[49] proposes the dewey order encoding to answer queries such as *following* and *following sibling*. Here each node is assigned a vector that represents the path from the document's root to the node. The labels for each node can get very big for deep trees. Also note that when implementing structure-based brushes we only need to answer ancestor queries. Thus, for our visualization application the labels do not have to contain information about the absolute position of the node in the complete tree.

7.4 Caching

Unlike the microprocessor's instruction and data caches where objects in the cache are directly referenced by their ids (address), in our case the objects in the cache are referred by a property they satisfy, i.e, if they lie in the current structure-based brush. Thus a set of objects is referenced using a query (current brush). High level caching system in which objects are not individually referenced is called *semantic caching* [12] or *predicate caching* [27]. Semantic caches organize or group the contents of the buffer by queries that retrieved them. The groups are called *semantic regions* and the queries are called *semantic descriptors*. The cache lookup and replacement is then done at the granularity of semantic regions. However, our memory management system is different from the one proposed for semantic caches [12, 27]. The contents of the buffer are not organized by the set of queries that retrieved them; rather, each cached object is handled individually and stores a *descriptor* that helps to identify if the object falls in the current query. Thus, we

handle data at a smaller granularity (i.e., object level) with regards to cache lookup and replacement. This approach avoids the question of cache utilization raised in [12]. One can argue that this gain comes as a trade off against the higher cost of cache lookup, since in our architecture the complete cache has to be scanned for each request. To counter this argument we exploit characteristics of visual exploration environments such as regularity of the query type (Section 3.4) and spatial encoding of navigation operations to use a main memory spatial index structure that can eliminate the need to scan the complete cache. Other work in the area of object level caching for database applications has been addressed for example in [10, 39]. Also, object-based caching has been studied recently in the context of web applications [17].

[12] uses *semantic distance* for replacing semantic regions. It uses *Mahattan distance* as a *semantic value function* to determine the importance of each semantic region. Here each semantic region is assigned a replacement value between the "center of gravity" of that region and the most recent query. The semantic region with the maximum distance from the most recent query is replaced. This idea is similar to the Distance replacement policy we use, the only difference being we do not replace complete semantic regions (brushes); we only replace individual objects. The main reason being for visual exploratory environments such as ours consecutive queries (brushes) exhibit a high degree of overlap. Emptying the results of complete queries (brushes) from the cache can remove potentially useful objects from the cache.

7.5 Prior Work in XmdvTool

We use this section to describe the exact contributions of this thesis to XmdvTool. There has been a significant effort over the past four years to scale hierarchical displays in XmdvTool to support navigation operations over large data sets. Onekey effor of this

thesis work has been simply to fix many of the assumptions and the limitations of the prior work.

[48] proposed to use a hierarchy encoding and caching scheme to support navigation operations over large data sets. However, the hierarchy encoding technique did not correctly incorporate the vertical selection semantics of the structure-based brush. Precisely a node n could satisfy the vertical selection criteria iff the $lod(n) = lod(brush)$. This criteria does not give the same result as Algorithm 1. To solve the problem we change the criteria. That is a node n satisfies the vertical selection criteria iff $lod(parent(n)) < lod(brush) \leq lod(n)$. Furthermore, we also extend the idea and propose a mapping of the hierarchy to the *2D Hierarchy Map* and the navigation operations to spatial queries. This mapping now allows us to use spatial index structures to execute efficient searches.

The caching scheme proposed in [47] also suffered from the same problem. For efficient cache lookup it proposed hashing the nodes based on their *level-of-detail* value. Assumed that the node satisfies the vertical selection criteria only if the desired *level-of-detail* is exactly equal to that of the node. As per the structure-based brush semantics a node n is a part of the vertical selection iff $lod(parent(n)) < lod(brush) \leq lod(n)$. In this thesis we propose a caching scheme described in Section 4.3 that overcomes this limitation. That is we no longer hash on the *level-of-detail*. Instead for fast cache lookup we propose maintaining a memory-resident spatial index that exploits the spatial mapping derived from the hierarchy encoding to search for selected nodes in the cache.

Furthermore the implementation of the cache in [47] was based on the assumption that the contents of the brush will always fit into the cache. This assumption seriously limits the scalability of the system, mainly because the maximum brush size is equal to the size of the unclustered data set (or the leaves in the cluster tree). For the system to be scalable the cache size should not directly depend on the data set size. To remove this assumption, we have implemented an incremental loading scheme (Section 5.2) that handles the case

when the brush size exceeds the cache size. It loads the data from the database in blocks equal to the cache size and delivers these blocks to the front end. Once a block is served it can be replaced by the new block coming from the database.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

With the increasing amount of data being accumulated nowadays, the need for visually exploring large datasets becomes imperative. A viable way to achieve scalability for visualization tools is to integrate them with database management systems [32]. Such integrations raise two problems: First it requires the organization of data on persistent storage such that visual exploration operations can be mapped to queries that can be efficiently executed on persistent data. Second, a good main memory management strategy is needed that exploits unique properties of visual exploration systems to reduce the overhead of database fetches and thus make the use of the database transparent to end-users. This thesis presents a framework of components that collectively address both these requirements.

The approach is being used to couple the XmdvTol 6.0, a visualization application for interactive exploration of multivariate data, with an Oracle 9i database management system. Experiments for assessing the approach show that collectively the components scale XmdvTool 6.0 to handle visual exploration operations over large hierarchies. In

summary the main contributions of the approach are:

- A hierarchy encoding technique that reduces the tree to an equivalent spatial representation called the *2-D Hierarchy Map*. This representation allows us to map recursive hierarchy navigation operations to non-recursive spatial search queries that can be answered efficiently using existing spatial index structures. We show the effectiveness of the spatial index structure.
- A caching strategy that exploits the characteristics of the visual navigation environment such as, locality of user exploration to buffer the recently used nodes to avoid database fetches and thus improve system response time. The main features of the cache are:
    1. Spatial Index based Cache Lookup Strategy: The cache exploits the spatial representation of the tree derived from the hierarchy encoding technique for efficient cache lookup. Specifically it builds a memory resident spatial index for fast searches over the cache.
    2. Distance based Cache Replacement: To exploit spatial locality exhibited in user traces [15], we implement a Distance replacement policy. The policy chooses a cache entry that is furthest away from the current active selection as the replacement victim.
- Integrated the direction-based prefetching strategy proposed in [15]. It mainly exploits user inertia, i.e, the tendency of the user to move in the same direction, to predict and buffer future user requests and thus reduce system response time.
- Performed experiments to quantify the usefulness of each component in the system. The experiments show that each of the components in the framework contributes

significantly to reducing system latency and thus will scale XmdvTool to work over large data sets. The summary of the experimental results is as follows:

1. The Spatial Index at the database used alone reduces the latency by up to 72%.
2. The cache when used together with the spatial index at the database reduces the latency by up to 94%.
3. The distance replacement policy performs as well or better than LRU replacement in most cases.
4. The prefetcher used together with the cache and the database index can reduce the latency by up to 96%.

8.2 Future Work

Directions for further research include both refining the current approach and making it more general by dropping some of the constraints that we are still enforcing now.

To improve the performance of the cache look up index (Main Memory R-Tree index) we can use a spatial index such as the LR-Tree [4] or Grid File [34] that claim to support fast updates and do not degenerate with frequent updates.

Integrated caching and prefetching [5] for visual exploratory environments is also an interesting area of future research. Prefetching and replacement both try to make the optimal use of the main memory available to the application. The former fetches data having high probability of being referenced next, and the latter replaces the data items with the least probability of being referenced when the need arises. This suggests that if both the strategies work together this may significantly improve the cache performance. Thus, the study of the inter-play between integrated caching and prefetching can be useful.

The system could also be functionally extended by dropping some of the current constraints, for example the "static" assumption. Currently we assume that the data set is

static. This assumption has two aspects. First, we might consider dynamic changes of the data set. It is more and more common to analyze information that suffers intensive updates during the exploration. Second, we might dynamically change the tools that we are using during the exploration itself. Dynamic clustering or dynamic computation of aggregates would be possible, for instance.

Another interesting extension to the system could be to consider how to extend the system to support multiple active brushes, this can affect the three main components of the framework i.e the hierarchy encoding, caching and prefetching. The hierarchy encoding scheme should work fine because each active brush can be represented as a line and the nodes in the brush are the nodes that intersect this line. However, having multiple brushes can raise some interesting questions for caching specifically cache replacement and prefetching.

For caching and prefetching for multiple active brushes we need to study user trace characteristics to see how often user's tend to switch between brushes? also whether the user trace characteristics listed in Section 3.4 still hold true? This study can help answer questions such as what nodes to cache? What nodes to replace? Can the nodes still be replaced based only on recency?

Moreover, to incorporate for multiple brushes we will need to modify the proposed cache replacement policies because the LRU and Distance replacement policy assume only one active brush. We can extend distance replacement policy to compute the distance of node using the distance from each active brush or extending the LRU replacement policy to maintain multiple lists each for a different brush. Also prefetching now will have to keep track of current active brush and predict which of the brushes will be moved and in what in direction.

Bibliography

[1] D. Andrews. Plots of high dimensional data. *Biometrics, Vol. 28, p. 125-36*, 1972.

[2] A. Becker and S. Cleveland. Brushing scatterplots. *Technometrics, Vol 29(2), p. 127-142*, 1987.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference*, pages 322–331, Atlantic City, NJ, June 1990.

[4] P. Bozanis, A. Nanopoulos2, and Y. Manolopoulos2. LR-tree: Logarithmic decomposable spatial index method. *British Computer Society*, pages 319–331, May 2003.

[5] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. *ACM SIGMETRICS*, pages 171–182, 1995.

[6] J. Celko. *Joe Celko's Trees and Hierarchies in SQL for Smarties*. Morgan Kaufmann, May 2004.

[7] H. Chernoff. The use of faces to represent points in k-dimensional space graphically. *Journal of the American Statistical Association, Vol. 68, p. 361-68*, 1973.

[8] P. Ciaccia, D. Maio, and P. Tiberio. A method for hierarchy processing in relational systems. *Information Systems*, 14(2):93–105, 1989.

[9] W. Cleveland and M. McGill. *Dynamic Graphics for Statistics*. Wadsworth, Inc., 1988.

[10] G. P. Copeland, S. Khoshafian, M. G. Smith, and P. Valduriez. Buffering schemes for permanent data. In *Proc of the Second Intl Conf on Data Engineering, Feb 5-7, 1986, Los Angeles, Cal, USA*, pages 214–221, 1986.

[11] F. Dahlgren, M. Dubois, and P. Stenström. Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors. In *1993 International Conference on Parallel Processing*, volume 1, pages 56–63, 1993.

[12] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proc of 22th Intl Conf on Very Large Data Bases*, pages 330–341. Morgan Kaufmann, 1996.

[13] M. Derthick, J. Harrison, A. Moore, and S. Roth. Efficient multi-object dynamic query histograms. *Proc. of Information Visualization*, pages 58–64, Oct. 1999.

[14] P. F. Dietz. Maintaining order in a linked list. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 122–127. ACM Press, 1982.

[15] P. R. Doshi, E. A. Rundensteiner, and M. O. Ward. A strategy selection framework for adaptive prefetching in data visualization. *15th International Conference on Scientific and Statistical Database Management*, pages 107–116, 2003.

[16] P. R. Doshi, E. A. Rundensteiner, and M. O. Ward. Prefetching for visual data exploration. *Eighth International Conference on Database Systems for Advanced Applications*, pages 195–202, March 2003.

[17] D. Florescu, A. Y. Levy, D. Suciu, and K. Yagoub. Run-time management of data intensive web sites. In *WebDB (Informal Proceedings)*, pages 7–12, 1999.

[18] Y. Fua, M. Ward, and E. Rundensteiner. Structure-based brushes: A mechanism for navigating hierarchically organized data and information spaces. *Submitted to IEEE Transactions on Visualization and Computer Graphics*, June 2000.

[19] Y. H. Fua, M. O. Ward, and E. A. Rundensteiner. Hierarchical parallel coordinates for exploration of large datasets. *IEEE Proc. of Visualization*, pages 43–50, Oct. 1999.

[20] Y. H. Fua, M. O. Ward, and E. A. Rundensteiner. Navigating hierarchies with structure-based brushes. *Proc. of Information Visualization*, pages 58–64, Oct. 1999.

[21] A. Guttman. R-trees: a dynamic index structure for spatial searching. *ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[22] S. Hibino and E. A. Rundensteiner. Processing incremental multidimensional range queries in a direct manipulation visual query environment. In *Proc of the Fourteenth Intl Conf on Data Engineering, Orlando, Florida, USA*, pages 458–465, 1998.

[23] A. Inselberg and B. Dimsdale. Parallel coordinates: A tool for visualizing multidimensional geometry. *Proc. IEEE Visualization*, pages 361–378, 1990.

[24] C. Jeong and A. Pang. Reconfigurable disc trees for visualizing large hierarchical information space. *Proc. of Information Visualization '98, p. 19-25*, 1998.

[25] R. R. Johnson, M. Goldner, M. Lee, K. McKay, R. Shectman, and J. Woodruff. USD - a database management system for scientific research. *ACM SIGMOD Conf*, pages 4–4, 1992.

[26] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *Proceedings of the thirteenth annual ACM-SIAM Symposium on Discrete algorithms*, pages 954–963. Society for Industrial and Applied Mathematics, 2002.

[27] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *VLDB Journal*, 5(1):35–47, 1996.

[28] T. Kurc, U. Atalyurek, C. Chang, A. Sussman, and J. Salz. Exploration and visualization of very large datasets with the active data repository. Technical report, University of Maryland, 2001.

[29] J. LeBlanc, M. Ward, and N. Wittels. Exploring n-dimensional databases. *Proc. of Visualization '90, p. 230-7*, 1990.

[30] Y. Leung and M. Apperley. A review and taxonomy of distortion-oriented presentation techniques. *ACM Transactions on Computer-Human Interaction, Vol. 1(2)*, pages 126–160, June 1994.

[31] Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 361–370. Morgan Kaufmann Publishers Inc., 2001.

[32] X. Ma, M. Winslett, J. Norris, X. Jiao, and R. Fiedler. Godiva: Lightweight data management for scientific visualization. *20th International Conference on Data Engineering*, pages 732–744, 2004.

[33] S. Manoharan and C. R. Yavasani. Experiments with sequential prefetching. *In Proceedings of the High-Performance Computing and Networking Conference*, pages 322–331, 2001.

[34] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, pages 38–71, 1984.

[35] Oracle, odbc and db2-cli template library. http://otl.sourceforge.net/.

[36] R. Ramkrishnan and J. Gehkre. *Database Management Systems*. McGraw-Hill Higher Education, 3rd edition, 2002.

[37] W. Ribarsky, E. Ayers, J. Eble, and S. Mukherjea. Glyphmaker: Creating customized visualization of complex data. *IEEE Computer, Vol. 27(7), p. 57-64*, 1994.

[38] G. Robertson, J. Mackinlay, and S. Card. Cone trees: Animated 3d visualization of hierarchical information. *Proc. of Computer-Human Interaction '91, p. 189-194*, 1991.

[39] N. Roussopoulos, C.-M. Chen, S. Kelley, A. Delis, and Y. Papakonstantinou. The ADMS project: View R us. *Data Engineering Bulletin*, 18(2):19–28, 1995.

[40] E. A. Rundensteiner, M. O. Ward, J. Yang, and P. R. Doshi. Xmdvtool: visual interactive data exploration and trend discovery of high-dimensional data sets. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 631–631. ACM Press, 2002.

[41] P. G. Selfridge, D. Srivastava, and L. O. Wilson. IDEA: Interactive data exploration and analysis. In *Proc of the 1996 ACM SIGMOD Intl Conf on Management of Data*, pages 24–34. ACM Press, 1996.

[42] B. Shneiderman. Tree visualization with tree-maps: A 2d space-filling approach. *ACM Transactions on Graphics*, 1992.

[43] J. Siegel, E. Farrell, R. Goldwyn, and H. Friedman. The surgical implication of physiologic patterns in myocardial infarction shock. *Surgery Vol. 72, p. 126-41*, 1972.

[44] Spatial index library. http://www.cs.ucr.edu/ marioh/spatialindex/.

[45] C. Stolte, D. Tang, and P. Hanrahan. Multiscale visualization using data cubes. In *Proceedings of the Eighth IEEE Symposium on Information Visualization*, pages 1–8, Boston, Oct. 2002. IEEE.

[46] M. Stonebraker, J. Chen, N. Nathan, C. Paxson, and J. Wu. Tioga: Providing data management support for scientific visualization applications. In *19th Intl Conf on Very Large Data Bases, 1993, Dublin, Ireland*, pages 25–38. Morgan Kaufmann, 1993.

[47] I. D. Stroe. Scalable visual hierarchy exploration. Master's thesis, Worcester Polytechnic Institute, May 2000.

[48] I. D. Stroe, E. A. Rundensteiner, and M. O. Ward. Scalable visual hierarchy exploration. In *Database and Expert Systems Applications, Greenwich, UK*, pages 784–793, Sept. 2000.

[49] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of the International Conference on Management of Data (SIGMOD'02), Madison, Wisconsin, USA*, pages 204–215, 2002.

[50] J. Teuhola. Path signatures: A way to speed up recursion in relational databases. *IEEE Transactions on Knowledge and Data Engineering*, 8(3):446–454, June 1996.

[51] V. Tropashko. Trees in sql. http://www.dbazine.com/tropashko4.shtml.

[52] M. Ward. Xmdvtool: Integrating multiple methods for visualizing multivariate data. *Proc. of Visualization '94, p. 326-33*, 1994.

[53] E. Wegman. Hyperdimensional data analysis using parallel coordinates. *Journal of the American Statistical Association, Vol. 411(85)*, page 664, 1990.

[54] Xmdvtool home page. davis.wpi.edu/ xmdv.

[55] J. Yang, M. Ward, and E. Rundensteiner. Interactive hierarchical displays: A general framework for visualization and exploration of large multivariate data sets. *Computers & Graphics*, 27(2):265–283, 2003.