Adaptive Prefetching for Visual Data Exploration

by

Punit R. Doshi

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

January 2003

APPROVED:

Professor Elke A. Rundensteiner, Thesis Advisor

Professor Matthew O. Ward, Thesis Co-advisor

Professor Craig E. Wills, Thesis Reader

Professor Micha Hofri, Head of Department

Abstract

Loading of data from slow persistent memory (disk storage) to main memory represents a bottleneck for current interactive visual data exploration applications, especially when applied to huge volumnes of data. Semantic caching of queries at the client-side is a recently emerging technology that can significantly improve the performance of such systems, though it may not in all cases fully achieve the near real-time responsiveness required by such interactive applications. We hence propose to augment the semantic caching techniques by applying prefetching. That is, the system predicts the user's next requested data and loads the data into the cache as a background process before the next user request is made. Our experimental studies confirm that prefetching indeed achieves performance improvements for interactive visual data exploration. However, a given prefetching technique is not always able to correctly predict changes in a user's navigation pattern. Especially, as different users may have different navigation patterns, implying that the same strategy might fail for a new user. In this research, we tackle this shortcoming by utilizing the adaptation concept of strategy selection to allow the choice of prefetching strategy to change over time both across as well as within one user session. While other adaptive prefetching research has focused on refining a single strategy, we instead have developed a framework that facilitates strategy selection. For this, we explored various metrics to measure performance of prefetching strategies in action and thus guide the adaptive selection process. This work is the first to study caching and prefetching in the context of visual data exploration. In particular, we have implemented and evaluated our proposed approach within XmdvTool, a free-ware visualization system for visually exploring hierarchical multivariate data. We have tested our technique on real user traces gathered by the logging tool of our system as well as on synthetic user traces. Our results confirm that

our adaptive approach improves system performance by selecting a good combination of prefetching strategies that adapts to the user's changing navigation patterns.

Acknowledgements

I would like to take this opportunity to thank all the people who helped me in my work in any way. A special thank to Geraldine Rosario for her 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 often think that I am very lucky that I can have two such excellent professors as my advisors at the same time.

I would like to thank Prof. Craig E. Wills for being the reader of this thesis and giving me much valuable feedback. Also, he had been patient enough to wait for my never ending work to be peacefully done.

I really appreciate *XmdvTool* team members for their cooperation and help whenever I needed. They made my programming easier by giving me proper documentation and clean code for reference.

I want to forward my sincere gratitute to all the people who took part in our user study and helped us to evaluate the performance of our system. I believe that their participation is also going to help our team for long run.

My family had been very supportive to me and pumping some energy into me whenever I needed it.

Also, thanks to NSF for funding this project under their grants IIS-0119276 and IIS-9732897.

Contents

1 Introduction **1**

1.1 Caching and Prefetching for Data Visualization 1

1.2 Our Approach 2

1.3 Contributions 4

1.4 Thesis Organization 5

2 Visualization Tool Case Study **6**

2.1 Brush Basics 7

2.2 Structure-Based Brushes 9

3 Semantic Caching **13**

3.1 Semantic Caching 13

3.1.1 Cache Replacement 14

4 Introduction to Prefetching **19**

4.1 Tasks Involved in Prefetching 19

4.2 Static Prefetchers Implemented in XmdvTool 20

4.2.1 Random Strategy 22

4.2.2 Direction Strategy 22

4.2.3 Focus Strategy 23

4.3 Disadvantages of Static Prefetchers 24

5 Adaptive Prefetching 25

5.1 Introduction to Adaptive Systems 25

5.2 Proposed Solutions 26

5.2.1 Strategy Refinement 26

5.2.2 Strategy Selection 26

5.3 Adaptive Focus Strategy 27

5.3.1 How do we determine the hot regions? 28

5.3.2 Access statistics 29

5.3.3 Hot-Region Calculation Algorithm 29

5.4 Strategy Selection Details 30

5.4.1 Set of Individual Prefetching Strategies 30

5.4.2 Performance Measures 31

5.4.3 Fitness Functions 34

5.4.4 Strategy Selection Policy 36

6 Implementation of Prefetching Architecture 38

6.1 System Architecture 38

6.2 Adaptive Prefetching Architecture 40

7 Experimental Evaluation 42

7.1 Experimental Inputs 42

7.2 Settings 43

7.3 Experiments Related to Strategy Refinement 44

7.4 Experiments Related to Strategy Selection 45

7.4.1 Case Study I 46

7.4.2 Case Study II 51
7.4.3 Summary Charts 55

8 Related Work 59
8.1 Adaptive Prefetching 59
8.2 Caching 61
8.3 Database Support for Interactive Applications 61

9 Conclusions and Future Work 63
9.1 Summary and Conclusions 63
9.2 Future Work 64

List of Figures

2.1 Structure-based brush as combination of a horizontal (a) and a vertical (b) selection. 8
2.2 Parallel Coordinates Display of the 5-dimensional Remote Sensing data set with 16,384 data points. 9
2.3 Structure-based brush in XmdvTool. (a) Hierarchical tree frame; (b) Contour corresponding to current level-of-detail; (c) Leaf contour approximates shape of hierarchical tree; (d) Structure-based brush; (e) Interactive brush handles; (f) Color map legend for level-of-detail contour. 9
2.4 Parallel Coordinates Display of Remote Sensing data set after Roll-up operation from Figure 2.2. 10
2.5 Roll-up operation of the Structure-based brush from Figure 2.3. 10
2.6 Parallel Coordinates Display of Remote Sensing data set after Drill-down operation. 11
2.7 Drill-down operation of the Structure-based brush. 11

3.1 Cache content for a three level, twelve object example in case of a reduced probability table. 17

4.1 Hierarchy of Prefetching Strategies. 21
4.2 Random Strategy. 22

4.3 Direction Strategy. 22
4.4 Hot Regions. 23

5.1 Data structure used to keep access statistics 29
5.2 Algorithm for calculating the hot region(s) 30
5.3 Prediction Measurements 33

6.1 System architecture. Dotted-line rectangles show the separation between the on-line and the off-line computation. Solid-line rectangles represent the modules. Ovals represent data. Arrows show the control flow. 39
6.2 Adaptive Prefetching Framework. 41

7.1 Response time vs. hot-region update frequency 44
7.2 Prediction rates vs. hot-region update frequency 45
7.3 Directionality vs. Time for User A 46
7.4 Number of Queries vs. Time for User A 47
7.5 Regions visited vs. Time for User A 47
7.6 Brush movements vs. Time for User A 48
7.7 Mis-Classification Cost vs. Time for User A 49
7.8 Strategy selection vs. Time for User A 49
7.9 % of Not predicted objects vs. Time for User A 50
7.10 % of Mis-predicted objects vs. Time for User A 50
7.11 % of Correctly predicted objects vs. Time for User A 51
7.12 Response Time vs. Time for User A 51
7.13 Directionality vs. Time for User B 52
7.14 Number of Queries vs. Time for User B 52
7.15 Regions visited vs. Time for User B 53
7.16 Brush movements vs. Time for User B 53

7.17 Mis-Classification Cost vs. Time for User B 54
7.18 Strategy selection vs. Time for User B 54
7.19 % of Not predicted objects vs. Time for User B 55
7.20 % of Mis-predicted objects vs. Time for User B 55
7.21 % of Correctly predicted objects vs. Time for User B 56
7.22 Response Time vs. Time for User B 56
7.23 Global Average Mis-classification Cost (Averaged) For Different User Clusters 57
7.24 Normalized Response Time (Averaged) For Different User Clusters . . . 57

Chapter 1

Introduction

1.1 Caching and Prefetching for Data Visualization

Visualization provides valuable techniques for the analysis of data. While statistics offers us various tools for testing model hypotheses and finding model parameters, the task of guessing the right type of model to use is still a process that cannot be automated. Thus, whether the domain is stock data, scientific data, or the distribution of sales, visualization plays an important role in analysis. Humans can sometimes detect patterns and trends in the underlying data by just looking at it.

Techniques for representing the data greatly influence human perception. Thus, various techniques for displaying data have been proposed over the years, each of which focuses on emphasizing some of the characteristics of data [4, 31, 9, 18, 35]. However, most of these techniques do not scale well with respect to the size of the data. As a generalization, [22] postulates 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 large datasets.

For displaying large datasets we [48] present the data at different levels of detail by

applying an aggregation function to a hierarchical structure, a structure that might result for instance from a clustering process. The problem of clutter at the interface level is thus solved by displaying only a limited set of aggregates at a time. However, such hierarchical summarizations increase the size of data to be managed by at least one order of magnitude.

Storing and retrieving the data sets efficiently has often been ignored in the context of visualization applications. While storing the data in main memory and flat files is appropriate for small and moderate sized data sets, it becomes unacceptable when scaling to large datasets. One possible solution to enable scaling is to integrate visualization applications with database management systems [40, 2, 10]. Techniques from the database field should be able to greatly contribute to increasing the performance of a data intensive application such as exploratory visualization.

Although coupling of visualization tools and databases is a challenging task in itself, this work concentrates on further improving the performance of a visualization system within the context of such an integrated system. Loading of data from persistent storage results in larger response times faced by the user because it involves I/O operations. Caching the data is one of the options in order to achieve a better performance (response time) [42]. Prefetching the data during the idle time would further improve the response time [42]. Our work deals with investigating different prefetching techniques to improve the performance of an interactive system.

1.2 Our Approach

Our approach to reduce the system response time (i.e., the on-line computation time) is to exploit the characteristics of the visualization environment such as:

- Locality of exploration - Since the graphical interface allows the user limited moves only, the user explorations are local compared to those that could be specified via a

generic SQL query interface. For this reason, a part of the user query might already be there in the cache because of the overlap between the previous and current user query.

- Predictability of user movements - Users are more predictable because of the limited types of data requests feasible via the visual interactive tools and also due to existence of some typical styles of exploration. This is confirmed by our analysis of actual log files of usage of our tool.

- Idle time - Since the user would be spending quite some time analyzing the data (displayed in different formats by the visualization system), the system would remain idle for that time. This idle time can be used to prefetch the next most probable data to reduce system response time.

The locality of exploration and predictability of user movements can be used to predict the most probable data requested by the next user query. Idle time allows us to prefetch the data before the next user requests comes in. We have already began to exploit these characteristics to develop simple static prefetching schemes [16]. These static prefetching schemes are based on the characteristics of both users and data. This resulted in improvement over static prefetching using just user characteristics.

Static prefetching of data helps to improve the performance by 32% in our system [16]. These strategies are not tailored for changes in the user navigation patterns. Static prefetching in the context of an exploratory data visualization environment has several drawbacks:

- No single static prefetcher will work best for all types of users as confirmed by our experimental evaluation.

- Navigation patterns of a single user even within one single usage session is likely

to change as the user gains more knowledge about the data, becomes more familiar with the data visualization tool, or changes her goal of exploration.

- Static prefetchers do not consider how they have performed in their previous predictions. They typically generate predictions independent of their past performance on previous predictions.

A logical next step is to make prefetching adaptive. An adaptive prefetcher changes its prediction behavior in response to a changing environment with the goal of improving response time. We can use user navigation patterns, user type, data patterns as the symptoms of changing environment in interactive visualization applications.

The adaptive prefetching topic is multi-faceted. In this research, we target the specific hypothesis that different prefetching strategies work well for different usage situations. Navigation patterns vary greatly and are influenced by the user's inherent navigation preference (erratic vs. directional), user's familiarity with the dataset, user's familiarity with the visualization tool, and the data patterns present in the dataset. As such, we focus on investigating strategy selection mechanisms that can adaptively shift between prefetching strategies within a given user session.

1.3 Contributions

Our main contribution consists of developing a framework for adaptation of prefetching in any interactive visualization system. We have applied this adaptation to an interactive visualization tool, but this idea can be extended for any interactive system. The characteristics of such system for which the framework can be applied is given in Chapter 2.

In this project, we first designed and implemented simple static prefetching strategies that improve the performance of the system in terms of response time [16]. We utilized a high level caching [42] that reduces the system response time by incrementally loading

the data into the cache. When the system is idle, a prefetching strategy will bring the data that is most likely to be used next into the cache. We performed experiments and evaluated the results to show that prefetching together with caching help to improve our system's performance by about 80% in some cases [16].

In this thesis, we utilized adaptation techniques to fine-tune existing static prefetching techniques and also adaptively select between already existing simple prefetching techniques. We gather the statistics about user's past exploration and determine the regions of interest for her. These statistics are gathered whenever the user changes the selected data to be visualized. The regions of interest are updated periodically to reflect the recent regions under consideration by the user. We also utilized performance measurement functions to dynamically select between prefetching techniques. Our evaluation confirms that adapting between strategies helps to further improve the performance of the system.

1.4 Thesis Organization

Chapter 2 gives an overview of the XmdvTool system which we are using as a testbed for our research. The details of our caching architecture are specified in Chapter 3. While Chapter 4 introduces static prefetching, Chapter 5 discusses our approach to adaptive prefetching. We present the implementation of our framework and the experimental evaluation in Chapters 6 and 7. Related research in different fields are covered in Chapter 8. Finally, we describe our conclusions and open issues for future work in Chapter 9.

Chapter 2

Visualization Tool Case Study

The work presented in this paper was triggered by our goal of scaling *XmdvTool* to work on large data [48]. The following sections are explained in more details in Daniel's thesis [42]. XmdvTool in a software package designed for the exploration of multivariate data. The tool provides four distinct visualization techniques (scatterplot matrices, parallel coordinates, glyphs, and dimensional stacking) that allow interactive selections. We have produced various displays that allow multi-resolution data presentation [21, 54] We cluster the datapoints based on some distance metric, apply an aggregation function to the datapoints from each cluster and have those aggregate values displayed instead of the datapoints themselves. The model can be conceptualized as a hierarchy that provides the capability of visualizing data at various *levels of abstraction*. The hierarchical structure can be explored by interactively selecting and displaying points at different levels of detail. We term this exploration process *navigation*. In what follows we describe these visual exploration operations in more detail and then provide a formal model that summarizes the semantics of these operations.

2.1 Brush Basics

Selection is a process whereby a subset of entities on a display is isolated for further manipulation, such as highlighting, deleting, or analysis. Wills [51] defined a taxonomy of selection operations, classifying techniques based on whether memory of previous selections is maintained or not, whether the selection is controlled by the underlying data or not, and what specific interactive tool (e.g., brushing, lassoing) is used to differentiate an area of the display.

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. The principles of brushing were first explored by Becker and Cleveland [5] and applied to high dimensional scatterplots. Ward and Martin [48, 34] extended brushing to permit brushes to have the same dimensionality as the data (N-D instead of 2-D).

Another category, namely *structure space* techniques, which allows selection based on structural relationships between data points, has been introduced in [22]. The *structure* of a data set specifies relationships between data points. This structure may be explicit or implicit. Examples of structures include linear orderings, tree hierarchies, and directed acyclic graphs. In this work we focus on tree hierarchies. We have tried two clustering algorithms to build tree hierarchies in our system, but others would be suitable as well. Specifically, we have used BIRCH [55], DYSECT [3] as well as a simple one of ours [53]. A *tree* is a convenient mechanism for organizing large data sets. By recursively clustering or 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.

As described earlier, brushing requires some containment criteria. For our first containment criterion, we augment each node in the hierarchy, that is each cluster, with a monotonic value relative to its parent. This value can be, for example, the level number, the cluster size/population, or the volume of the cluster (defined by the minimum and maximum values of the nodes in the cluster). This assigned value determines the control for the level-of-detail. Our second containment criterion for structure-based brushing is based on the fact that each node in a tree has extents, denoted by the left- and right-most leaf nodes originating from the node. In particular, it is always possible to draw a tree in such a way that all its children are horizontally ordered. These extents ensure that a selected subspace is contiguous in structure space.

A structure-based brush is thus defined by a subrange of the structure extents and level-of-detail values. In a 2-D representation of the tree (Fig. 2.1), the subranges correspond to a horizontal and vertical selection, respectively.

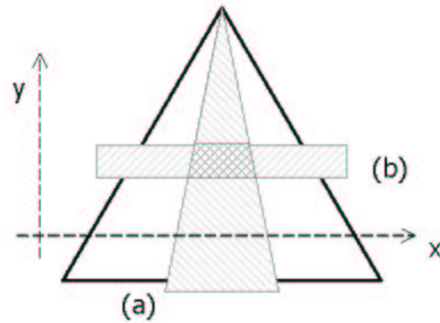

Figure 2.1: Structure-based brush as combination of a horizontal (a) and a vertical (b) selection.

2.2 Structure-Based Brushes

Figure 2.2 shows the parallel coordinates display of a five dimensional data set containing 16,384 records. In this display technique, each of the N dimensions is represented as a vertical axis, and the N axes are organized as uniformly spaced lines. A data element in an N-dimensional space is mapped to a polyline that traverses across all of the N axes crossing each axis at a position proportional to its value for that dimension. For example, in Figure 2.2, there is a polyline that intersects the "Spot" axis at value 119, the "Mag" axis at 149, the "Potas" axis at 41, and so on to display the tuple (Spot, Mag, Potas, Thor, Uran) = (119, 149, 41, 56, 56).

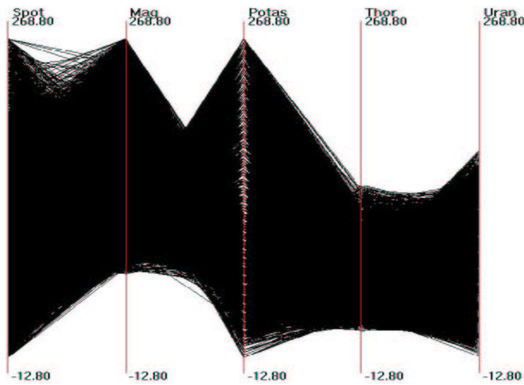

Figure 2.2: Parallel Coordinates Display of the 5-dimensional Remote Sensing data set with 16,384 data points.

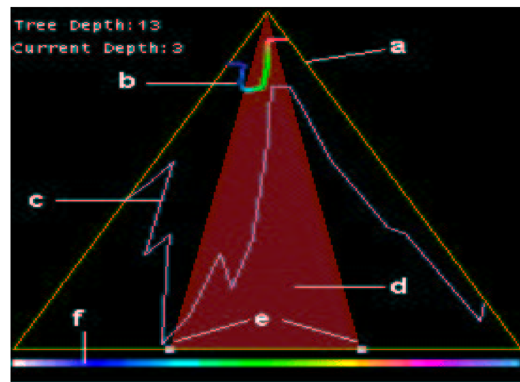

Figure 2.3: Structure-based brush in XmdvTool. (a) Hierarchical tree frame; (b) Contour corresponding to current level-of-detail; (c) Leaf contour approximates shape of hierarchical tree; (d) Structure-based brush; (e) Interactive brush handles; (f) Color map legend for level-of-detail contour.

As seen from Figure 2.2, displaying all the data to the user at the same time results in display clutter. Hence we need to provide the user with operations such as drilling down and rolling up the level of detail of the data. Towards this end, XmdvTool first clusters the data points into a cluster hierarchy, and then associates aggregate information with

the resulting clusters [49], such as extents and level-of-detail to map the hierarchy into a two-dimensional plane. Different levels in the cluster tree represent different degrees of abstraction of the data.

In order to improve the support for visual navigation through this cluster tree of data sets with millions or more records, we have designed a visual navigation tool that we term a *structure-based brush* (as depicted in Figure 2.3) [20]. A structure-based brush can be used to explore the data by interactively selecting and displaying the data at different levels of detail of the cluster hierarchy. In Figure 2.3, the brushing tool component marked '*e*' selects cluster(s) to be displayed, while '*b*' selects the level of detail for the selected cluster(s). While exploring the data, a user may navigate by sliding the extents of '*e*' horizontally to select a particular cluster in the tree hierarchy, or by moving the level brush '*b*' vertically to display data at different levels of detail.

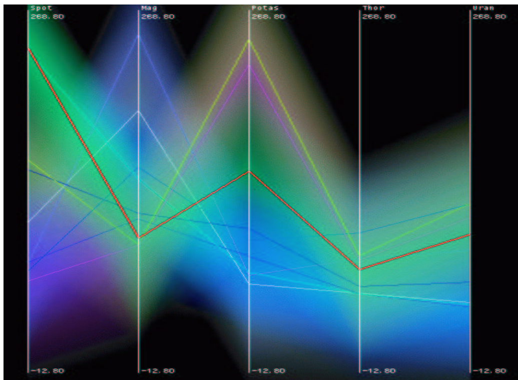

Figure 2.4: Parallel Coordinates Display of Remote Sensing data set after Roll-up operation from Figure 2.2.

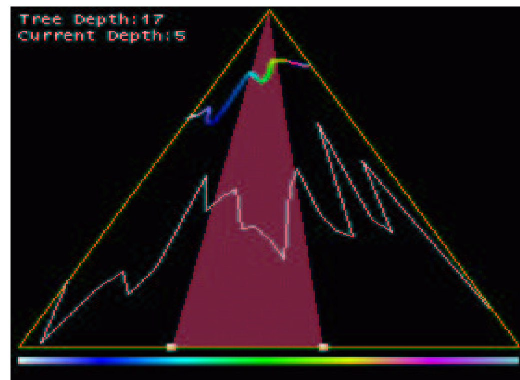

Figure 2.5: Roll-up operation of the Structure-based brush from Figure 2.3.

Figure 2.4 shows the display of the same data set when the user performs a roll-up operation using the structure-based brush, corresponding to sliding the level-of-detail 'b' from Figure 2.3 up to the setting of the brushing tool shown in Figure 2.5. While Figure 2.6 shows the display when the user performs a drill-down operation using the structure-based brush in the brushing tool as shown in Figure 2.7.

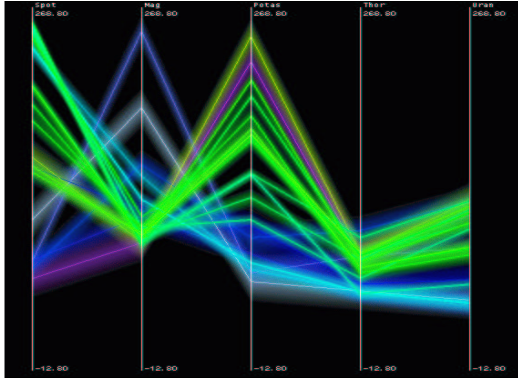

Figure 2.6: Parallel Coordinates Display of Remote Sensing data set after Drill-down operation.

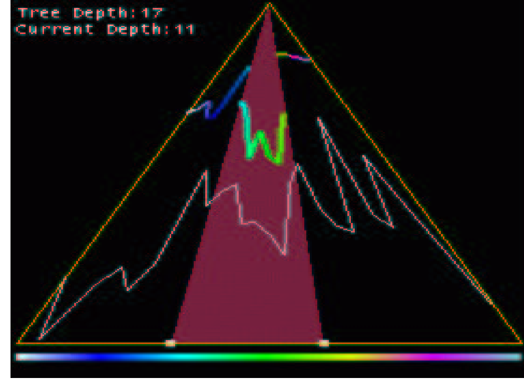

Figure 2.7: Drill-down operation of the Structure-based brush.

The user navigation operations expressed by our brushing tool are translated into queries to the database. The queries are *contiguous* rather than ad-hoc, since the visual interface provides only controlled means of expressing navigational requests via the structure-based brush. Such contiguity of user queries can be exploited when caching queries, as there is a high probability of a partial query result from a prior query still being relevant (and thus in our cache) for the next user request.

Second, we note that the user may be viewing the data around a particular region for a while before moving to another region. In other words, the user navigation tends to be composed of several small and *local* movements of the brushing tool (queries) rather than major global and unrelated movements.

Furthermore, users' exploratory movements are somewhat more *predictable* when they explore the data using such visualization tools, as such explorations are different from, say, random accesses via an ad-hoc SQL query interface. We thus postulate that prefetching may be a suitable mechanism for improving the performance of visualization and other exploratory applications.

Since the user will be examining the visual displays for interesting patterns, there typically would be *delays* between two user operations. Such delays could provide us

with the opportunity to prefetch highly probable data into main memory during idle times.

To summarize, typical characteristics of interactive visual exploration tools that can be exploited for caching and prefetching are:

1. contiguous queries passed to the database,

2. locality of exploration and thus data access,

3. predictable user's exploratory movements, and

4. significant delays between user operations.

Chapter 3

Semantic Caching

This chapter is explained in more details in Stroe's thesis [42]. Memory organization is critical in interactive applications since it influences the performance of the subsequent operations. When a request for new objects is issued by the front-end, the difference between the new active set (i.e., the set of objects just selected) and the current content of the cache has to be quickly computed. Thus, we need to be able to know in each moment what data resides in the memory without full traversal of the cache.

3.1 Semantic Caching

Semantic caching is a high level type of cache in which queries are cached rather than pages or tuples. A characteristic of the objects that are placed in the cache is that they are not referenced by their IDs when accessed by the front-end. In other words, the front-end doesn't ask for objects using requests such as $ID = x$ or $ID = y$; instead, it passes a query $q_{requested}$ to the back-end: "are the objects with these characteristics (within this brush) available?". We maintain a set of queries $q^i_{content}$ is associated with the cache, similar to *semantic caching* [12]. The query $q_{requested}$ is then compared with each $q^i_{content}$

to determine what objects from $q_{requested}$ are not in $q^i_{content}$, and those objects are retrieved next. This difference results in new queries (q^i_+) that correspond to those *to be loaded next* objects.

The problem of determine the q^i_+ queries is usually known as *query folding* [38]. It has been shown that the problem is reducible to the query containment problem [6]. Query containment is undecidable in the general case but decidable in the case of conjunctive queries [8]. As shown in Chapter 2 the queries in our case are all range queries, and therefore conjunctive.

Special attention has to be paid in a semantic caching environment to not allow duplicates in the cache. Thus, when more than one $q^i_{content}$ query is stored in the cache, they are *forced* to be disjoint. This means that a new $q_{requested}$ query will modify the semantic of the existing $q^i_{content}$ queries such that they do not refer to any common objects any more.

In order to make the object additions and subtractions efficient, we store the objects in the cache ordered by their extent value. The order can be ensured by the *query mechanism* itself or can be added as a new processing step. In our case we can request that the objects in all queries (as defined in Section 2) be retrieved in order by adding an *ORDERED BY* clause to MinMax-derived SQL queries.

A problem that all cache strategies need to solve is the cache replacement policy, i.e., to determine what objects have to be removed from the cache to make room for new objects. The first step in implementing a replacement policy is to provide an estimation strategy able to measure the likelihood that an object will be needed in the near future. We have used a probability function that also defines a partition on the set of objects.

3.1.1 Cache Replacement

The cache is first organized as a *bucket table* based on the probability values. The objects in the cache are hashed by rounding, based on a fixed number of values (a given precision).

The buckets will thus have values ranging uniformly from 0 (an open entry) to 1 (an object being currently in the active set). The objects in the same bucket are linked by a double linked list. Independently, the cache is also hashed based on the level value. Again, we have a bucket table with as many buckets as the level values. In addition, the header keeps a pointer to the last element in the list.

The main task of a cache replacement policy is to find in the cache the entries that have the lowest probability of being used and to remove them when more room is needed. This operation needs to be efficient, since it occurs frequently.

When new objects are brought in they have to comply with the internal organization. Updating the hash tables is then required. When a request is issued by the front-end, a containment test is performed. The system first checks whether the requested data reside entirely in memory or not. In case it doesn't, a *compensation* query has to be send to the *loader*, an agent that fetches the data from the persistent storage. The front-end may also send "refresh" queries when all objects within the current selection are needed.

An important requirement of the system that comes from its interactive nature is that the user needs to be able to preempt the other agents' actions. Thus, when the current selection changes, the loading process is interrupted and will only restart after recomputing the new probability values for the objects.

In conclusion, the cache access operations can be summarized as:

A: **Remove old objects.** Get the objects with the lowest probability that reside in the cache (and further remove them one at a time when more room in the cache is needed).

B: **Get new objects.** Place an object from the database *cursor* into the memory cache (and rehash the cache entry).

C: **Display active set.** Get those objects from the cache that form the active set (and

send them to the graphical interface to have them displayed).

D: **Recompute probabilities.** Recompute the probabilities of the objects in the cache once the active window gets changed (to ensure accurate predictions in the future).

E: **Test containment.** Test whether the new active set fully resides in the cache and gets the missing objects (if any) from the support set (when a new request is issued).

In the remainder of this section we will show how these operations are implemented in our cache strategy.

A speed up in the cache processing can be achieved by using a simplified version of the probability-based bucket table in which instead of storing all objects of the same probability in one bucket, we only store the ones that are extreme elements (first and last) in the level based lists.

As an example let us consider the navigation grid displayed in Fig. 3.1. We have here twelve regions of equal probability, the active window covering the middle two ones. For simplicity we consider that only one object resides in each region. We also number the objects from 1 to 12. The picture presents only three levels (1, 2, and 3). Also, probabilities are assigned to each region and implicitly to each object, based on a "operation-driven" probability model. Thus, objects 6 and 7 have a probability of 1, there is 40% chance that the window expands to the left, and so on. In this example a probability precision of 0.1 is assumed, and consequently 10 probability-based buckets are used. The probability table is reduced; one can see here for instance that only 5 and 8 are hashed out of the entire level 2.

An important assumption made at this point is that the query mechanism is able to provide the objects from the active set in both the increasing and the decreasing order of their extent value. If not, a sorting stage has to follow all calls to the query mechanism.

The main idea behind the cache access strategy is to keep the sets of objects on each

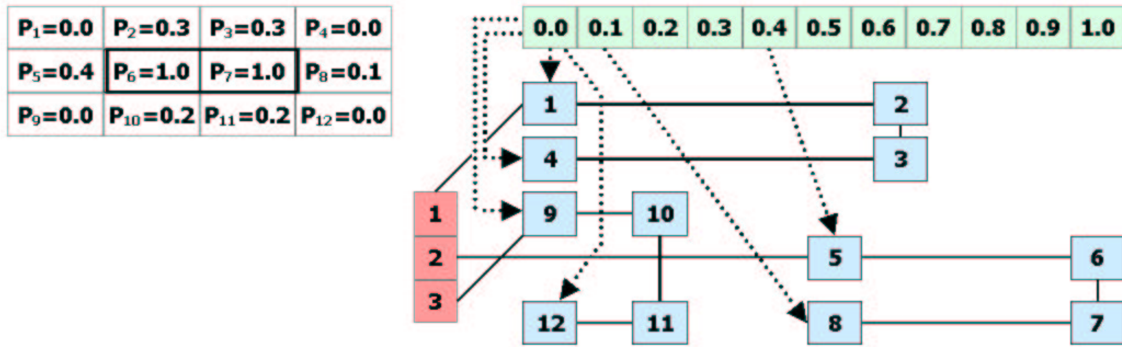

Figure 3.1: Cache content for a three level, twelve object example in case of a reduced probability table.

level always *convex* in the cache (with respect to the relation of total order defined among the objects of the same level). This is possible due to the fact that the lowest probability objects that need to be replaced (and thus removed from that level list) are always at the extreme of the list.

In what follows, the implementation of the cache access operations is described.

Operation A – **remove old objects** is equivalent to retrieving elements from the non-empty probability buckets in increasing order of their bucket value. The operation thus requires a scan of the probability table interleaved with traversals of the bucket lists.

Operation B – **get new objects** is equivalent to hashing an entry with respect to both its level and its probability value. Hashing an entry with respect to its level value is done in exactly one or two operations. Since the set of entries on each level is convex and contains the higher probability objects of the level, then the newly added object, which has the highest probability among the objects not yet in the cache, will necessarily be at the extreme. The entries on each level are ordered based on the extent value. Therefore the new object is either the new first element on that level's list if it is less than the previous first object, or the new last element if (necessarily) it is greater than the previous last element. Hashing with respect to the probability unfortunately takes $m/2$ operations in the average case, where m is the length of the current probability list. We need to make

sure than the object is correctly inserted (with respect to its extent value) in the sequence of elements having the same level in its probability bucket, so that removing the elements one by one from the probability list leaves all the affected level lists still convex. The efficiency will be improved though by using a reduced probability list.

Operation C – **display active set** is simply read all entries from the probability 1 bucket (last bucket). The number of cache accesses is the number of elements in the bucket.

Operation D – **recompute probabilities** requires at least one complete scan of the data. The objects preserve their level value so no change of the level lists is needed. However, the probability table needs to be rebuilt, and this takes $n + n^2/2$ operations. n is for deleting the lists (this can be done together with the probability recomputation step) and $n^2/2$ for creating the new ones (it is basically one list insertion for each object).

Operation E – **test containment** is composed of two steps. The inclusion test is ensured by the convexity property. An active set corresponding to an active window (e_1, e_2, k) is included in the cache, if and only if the list corresponding to level L_k is $o_{k1}, ..., o_{kn}$ and the left extent (geometrical extremity) of o_{k1}, $left(o_{k1})$ is less than or equal to e_1 and the right extent of o_{kn}, $right(o_{kn})$ is greater than or equal to e_2. If not included, the difference between the intervals (e_1, e_2) and $(left(o_{k1}), right(o_{kn}))$ gives us the next request(s) to be addressed to the *query mechanism*.

Chapter 4

Introduction to Prefetching

In visualization applications such as XmdvTool, users typically spend a significant amount of time interpreting the graphical presentation of the selected data, while the processor and I/O system are idle. We thus postulate that it is beneficial to predict what data the user will request next, and start fetching that data into the cache *before* the user asks for it. Thus, when the user requests that data later, she should perceive a faster response time. Due to the properties of visual exploration, such as the contiguity of queries, we can often accurately predict the user's next movement. *Prefetch* the data before the request comes from the tool is likely to be viable. The ideas of static prefetching that we present here have been published in [16].

4.1 Tasks Involved in Prefetching

Prefetching cannot be done in just one step. It involves doing the following steps:

- Predicting - The first step involved in prefetching is to predict the next most probable data. Different prefetching strategies have their own ways of predicting the next most useful data. This may involve keeping statistics for better prediction. The

infomation that can be gathered might be about the data and/or the user (in the case of interactive tools).

- Loading - The next step in prefetching is loading the data into the cache. This can be done by the same mechanism responsible for fetching the data when explicitly requested by the user. This might involve removing some of the lower probable data from the cache to make space for the prefetched data. Cache replacement policy discussed in Section 3.1.1 determines which objects to remove from the cache.

- Purging - The query descriptors and data replaced might need to be purged in order to reduce the overhead of storing information that is not required. Purging might involve combining descriptors to reduce the redundant data descriptions. Making sure that the data is stored in proper order inside the cache structure.

- Maintaining cache correctness - After loading the data, the cache should be in a correct state, as it was before prefetching was called. It means that the description of the data should represent actual data in the cache. This is very important as the prefetching can get preempted if a new user query comes in and we must assure that partially loaded data is properly checked in the cache.

4.2 Static Prefetchers Implemented in XmdvTool

In our target visualization application, prefetching requests may often be interrupted by further user requests, resulting in less data being prefetched at a time. For this reason we have designed an object-granularity cache management scheme as described in Section 3.1 so that partially prefetched results can be kept in the cache and put to future use.

Without *apriori* knowledge of the user request patterns [19], prefetching must be spec-

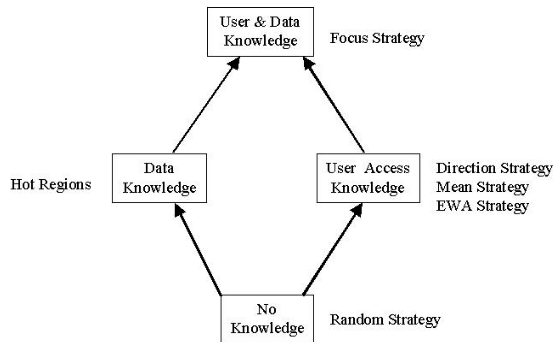

Figure 4.1: Hierarchy of Prefetching Strategies.

ulative as it has to guess. We have developed several speculative strategies for prefetching, as described below, in order to perform a comparative evaluation of their applicability to exploratory visualization systems. Figure 4.1 organizes these prefetching strategies into a hierarchy based on different hints they utilize. The assumption is that the predictor can discover the hints *gradually* rather than requiring complete knowledge *apriori*. The approach implies an evolutionary behavior; at the beginning, less information is available to the predictor and therefore the number of prefetching hints that it can discover (with a reasonable confidence) is also low. In time, more information (e.g., statistics) becomes available, and therefore more patterns (and implicitly hints) can be discovered. In all cases the prefetcher bases its strategy on the maximum amount of information it can find. In our case, we assume that the predictor can discover two types of navigation patterns. Specifically, we assume that the predictor can detect if the user tends to use more frequently the current navigation direction instead of changing it, and also it can detect if the data being analyzed has some regions of interest (so called *hot regions*) towards which the user will very likely go, sooner or later. Based on these assumptions, we have designed five prefetching strategies: *random* (S1), *direction* (S2), and *focus* (S3). In experiments, we also consider the case of not prefetching, which then is referred to as case S0.

4.2.1 Random Strategy

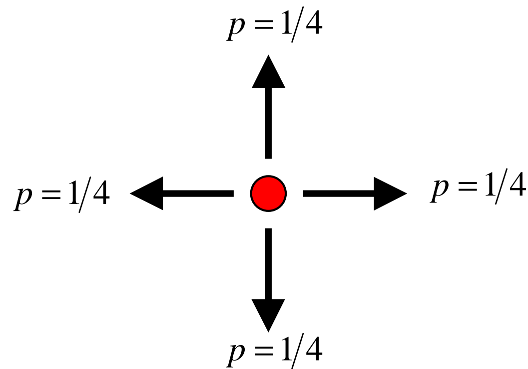

Figure 4.2: Random Strategy.

As shown in Figure 4.2, strategy S1 (random) is based on randomly choosing the direction in which to prefetch next. The directions are either lateral (left or right at the same level in hierarchy) or vertical (increase or decrease level of detail). Our visualization tool only allows manipulation in one of those four directions (using six possible operations). This strategy is appropriate when the predictor either cannot extract prefetching hints or provides hints with a low confidence measure.

4.2.2 Direction Strategy

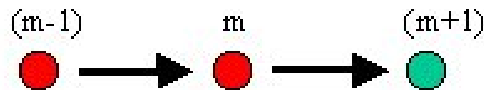

Figure 4.3: Direction Strategy.

Strategy S2 (direction) is analogous to the sequential prefetching scheme discussed in many of the prefetching papers [11, 33]. This *direction* strategy assumes that the most likely direction of the next operation can be determined. It is intuitive, for instance, that the user will continue to use the same navigation tool for a while before changing to

another one. In our system, each navigation tool of the structure-based brush happens to precisely control one direction only. Based on a user's past explorations, the predictor would assign probabilities to all the four directions. The prefetching strategy (S2) then is to "prefetch data in the direction" currently with the highest probability. As depicted in Figure 4.3, if $(m-1)$ and m are the last two directions navigated into by the user, then the *direction* strategy may predict $(m+1)$ as the next direction to be visited by the user in the same direction of the previous two movements.

4.2.3 Focus Strategy

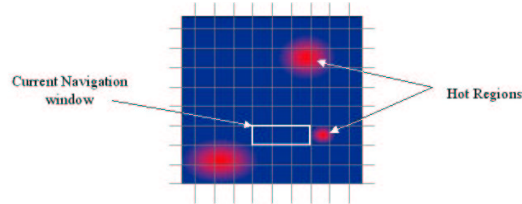

Figure 4.4: Hot Regions.

Strategy S3 (focus) uses information about the most probable next direction (by keeping track of user's previous movement) as well as hints about regions of high interest (hot regions, as depicted in Figure 4.4) in the data space as identified based on prior navigations of this same data by other users. We found the hot regions for each user by keeping the statistics of the regions visited by the user during the past explorations, and then maintaining all the regions that have frequency of visits above a particular threshold as hot regions. This strategy will continue to prefetch data in the given direction using the above mentioned direction heuristics. However, when a hot region is around the current navigation window, the prefetcher switches from the default *direction* prefetching to prefetch in that now more desirable direction. The hypothesis is that the user will likely stop at such a region of interest to explore those hot regions only if she got close enough

to notice it.

4.3 Disadvantages of Static Prefetchers

Our experimental evaluation of these static prefetching strategies [16] shows that static prefetching helps in reducing the response time visible to the user by around 30%. But these static prefetching strategies are not accurate in predicting the next possible user query resulting in large numbers of mispredicted objects being prefetched in some cases. Below is the list of disadvantages of using static prefetchers:

- No single static prefetcher will work best for all types of users as seen in Sections 7.4.1 and 7.4.2.
- Navigation patterns of a single user within a single usage session are likely to change as the user gains more knowledge about the data and becomes more familiar with the data visualization tool. This is shown in Chapter 7.
- Static prefetchers do not consider how they have performed in their previous predictions. They typically generate predictions independent of their performance on previous predictions, not allowing improvement over time.
- Focus strategy has a further disadvantage that you need to know the hot regions in advance for proper functioning. Also, these regions might be different for different users.

Chapter 5

Adaptive Prefetching

5.1 Introduction to Adaptive Systems

In general, an adaptive system extracts feedback from its environment in order to monitor the effects of its actions on its environment. The adaptive system also determines the effects of exploiting previously beneficial behavior and exploring alternative behavior [37]. Hence feedback is a key component to the effectiveness of an adaptive system.

Specifically, an adaptive prefetcher changes its prediction behavior in response to a changing environment. The changes in the environment in our case are for example the changes in user navigation patterns, user type, data patterns, etc.

Adaptive prefetching is a multi-faceted topic. In this research, we target the specific hypothesis that different prefetching strategies work well for different usage situations (Sections 7.4.1 and 7.4.2). Navigation patterns vary greatly and are influenced by the user's inherent navigation preference (erratic vs. directional), user's familiarity with the dataset, user's familiarity with the tool, and the data patterns present in the dataset. As such, we focus our discussion in this section on the strategy selection mechanisms that can adaptively shift between prefetching strategies within a given user session. We have

also worked with strategy refinement where we adapt the focus strategy by finding regions of interest to the user at particular intervals.

5.2 Proposed Solutions

5.2.1 Strategy Refinement

Each prefetching strategy has a set of input parameters that control how it operates and how well it performs. For the focus prefetching strategy, for example, the input parameters include the set of hot regions. Input parameter such as how much data to prefetch in a given direction (i.e., step size) is common to all prefetching strategies.

One way to improve the performance of an individual prefetching strategy is to periodically refine the values of its input parameters to adapt to changing user navigation patterns. Refinement involves the issues of feedback gathering (e.g., what response to gather and how often to gather) and of updating statistics (e.g., what/how/when to update statistics, how to age feedback and other information to give emphasis to recent activity). In this paper, we apply strategy refinement concepts in creating an Adaptive Focus Strategy. This is discussed in Section 5.3.

5.2.2 Strategy Selection

Given a set of individual prefetching strategies, a simple prefetching approach is to select one strategy at the start of a user session and use that same strategy throughout the session. A more flexible approach that suitable for adaptation is to allow the choice of prefetching strategy to change over time within a single session to adapt to changing user navigation patterns [36]. This is called strategy selection. This involves competition among the individual prefetching strategies in order to be chosen.

In using strategy selection for prefetching, every time prefetching needs to be done, a strategy is selected from a set of prefetching strategies based on its performance on past prefetching requests within the current session. We had indicated earlier in this section that extracting feedback is important in an adaptive system. In strategy selection, we extract feedback by measuring the performance of each prefetching strategy every time it is selected. This helps us monitor if our past selections were beneficial or detrimental to the overall performance of the system, and consequently helps the adaptation framework to decide what strategy to select next.

The basic building blocks of a strategy selection routine include:

1. set of strategies - a set of individual prefetching strategies to choose from are essential for selection,

2. performance measures - a set of statistics used to measure how well each prefetching strategy has performed,

3. a strategy selection policy - a set of rules used to determine which among competing strategies to select, and

4. a fitness function - a function of one or more performance measures that is used by the strategy selection policy to decide which strategy to select.

The details of our design of a complete strategy selection framework is given in Section 5.4.

5.3 Adaptive Focus Strategy

Recall from Section 4.3 that hot regions are the regions in the navigation space which a user visits frequently as he/she may have found some interesting patterns in those regions.

If the user visits these places frequently, then we may want to prefetch and cache data within these regions in order to reduce response time.

Also recall that in the static focus strategy, the set of hot regions used for a given user session is pre-determined at the start of the session and remains unchanged throughout that user session. The hot regions could be pre-determined from past sessions on the same data set by the same user or from all past users. However, the regions of interest may be different among users, and may vary for the same user within the same session. So one way to improve the performance of the focus strategy is to update the set of hot regions periodically within the same session to make sure that they represent the latest regions of interest (i.e., adapt to the user's changing regions of interest).

5.3.1 How do we determine the hot regions?

For any given display used for navigation (e.g., the structure-based brush in XmdvTool in Figure 2.3), we can construct a corresponding rectangular navigation grid similar to Figure 5.1. Any navigational movements made by the user on the navigation display can be mapped and recorded in this navigation grid.

As a concrete example, let us take a closer look at the structure-based brush in XmdvTool (Figure 2.3) which is a display used for navigating a data hierarchy tree. We can construct a corresponding navigation grid (Figure 5.1) whose rows are the levels in the data hierarchy tree, and whose columns are equal-sized vertical pie slices of the data hierarchy tree. The width of each column can be set to match the pre-defined granularity of the hot regions that we want to monitor, so each cell in the navigation grid corresponds to a region in the structure-based brush. Each time the user moves over a region in the structure-based brush, we record access statistics in the corresponding cells in the navigation grid. The details about statistics are given in the next section.

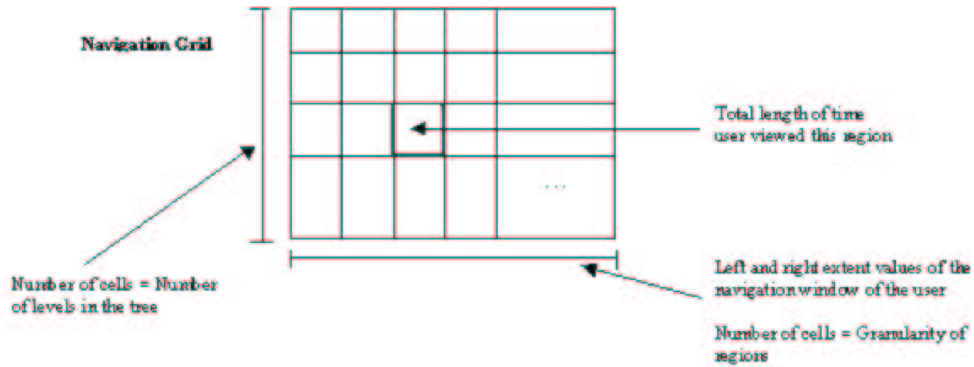

Figure 5.1: Data structure used to keep access statistics

5.3.2 Access statistics

The access statistics we record are the total time spent by the user in a region (time of next brush movement - time of current brush movement). The hypothesis is that if a user is interested in a region, she will spend a longer time viewing it. Another access statistic that could be monitored is the number of times a user visited a region; when a region is often visited, it means its data is loaded into memory often so it is a candidate for caching. But this keeps track of the regions that might be frequently visited when passing through to where user really wants to go because of the way navigation space is structured. Unlike these regions, we are interested in the regions the user often wants to analyze.

The access statistics are updated for every query (or brush movement). To ensure that the most recent regions of interest are recorded, we recalculate the set of hot regions after some pre-defined n number of queries. We clear the grid again for collecting statistics of next n queries and this procedure is repeated throughout the user session.

5.3.3 Hot-Region Calculation Algorithm

Figure 5.2 shows the algorithm for calculating and updating hot-regions. After finding the hot-regions, like static focus strategy, the adaptive focus strategy also uses information

```
n_queries = 100 /* update frequency */
cutoff=0.8 /* cut-off value for hot-region = 80% */
max_n_regions=5 /* max # of Hot Regions = 5 */
initialize access statistics to 0 for all cells in navigation grid
initialize set_of_hot_regions =
after serving every user request
    update the access statistics
    after every n_queries
        determine maximum access statistic value
        let candidate_hot_regions = all cells with access statistic ≥ cutoff * maximum
        among candidate_hot_regions, combine adjacent regions (within the same row) into single hot region
        let set_of_hot_regions = top max_n_regions hot regions
        re-initialize access statistics to 0 for all cells in grid
    end
end
```

Figure 5.2: Algorithm for calculating the hot region(s)

about the most probable next direction (by keeping track of the user's previous movement) as well as hints about hot regions. This strategy will continue to prefetch data in the given direction similar to the static direction prefetching strategy, subject to the available cache space. However, when there is a hot region in the direction of the current movement, the prefetcher switches from the default direction prefetching (prefetching as much possible in the current direction) to prefetch just up to the hot-region. The hypothesis is that the user will likely stop there to explore that hot region he came across.

5.4 Strategy Selection Details

This section gives the details of our implementation of the strategy selection framework.

5.4.1 Set of Individual Prefetching Strategies

For strategy selection, we have selected the following individual prefetching strategies:

- No Prefetch
- Random
- Direction

Sometimes when all the prefetching strategies are failing to improve the performance (perhaps because the time difference between queries is too short to allow for prefetching, or the remaining cache space is always too small to accommodate prefetched data), it is better not to prefetch the data. Hence, we have the *no prefetching* strategy in our list as well.

5.4.2 Performance Measures

The quality of a prefetcher can be measured by the following two factors [37]:

- Response time - how quickly can the data corresponding to the next user request be displayed to the user. A prefetcher is good if this value is low. This can be achieved by correctly predicting as many objects before the user request comes in, thus implying lower not predicted objects.

- Network traffic - how much communication is required between the tool on the client side and the database on the server side. Since accessing the database (a slow persistent storage) is slow, we want network traffic to be low. This value is determined by the number of objects retrieved from the database during fetching and prefetching. The lowest possible value would occur when (i) we do not prefetch (so by just fetching, we will load only data that is required) or (ii) we do not prefetch any data that will not be required in the next user request. In order to reduce network traffic, we want to minimize the number of mis-predicted objects.

These two factors suggest the following concrete performance measures:

- response time
- number of correctly predicted objects
- number of not predicted objects

- number of mis-predicted objects

The last three statistics measure the accuracy of a prefetcher in predicting which objects are needed by the next user query [28]. These concepts of preditions are also analogous to standard *precision and recall* metrics [23].

Response Time

Response time (RT) measures how quickly the data corresponding to the next *fetch* query (following the *prefetch* query of interest) can be loaded into memory.

$$responsetime(RT) = time(dataDisplayed) - time(userRequest) \tag{5.1}$$

Response time is easy to calculate, involves low calculation overhead and can be used for comparing no-prefetching versus with-prefetching experiments.

However, any performance measure involving time may be affected by external factors such as the database server workload, the size of the cache, and the presence of a lot of pre-empted prefetching operations. Hence, care must be taken in interpreting results based on response time. Also, the value of response time depends on the size of the user query. Hence, even if the prefetcher did not mis-predict any required object, the next *fetch* query may still have a slow response time simply because the query is large. Also, response time does not measure the accuracy of the prefetcher. The response time may be fast simply because it is based on previously fetched objects by some other prefetcher.

Prediction measurements

In order to overcome the drawbacks of response time, we came up with the following measures that are inspired by [37]:

1. Number of correctly predicted objects

Required by user query

Predicted by prefetcher

	Yes	No
Yes	Correctly predicted	Mis-predicted
No	Not predicted	

Figure 5.3: Prediction Measurements

2. Number of not predicted objects

3. Number of mis-predicted objects

These metrics are depicted in Figure 5.3.

The number of correctly predicted objects (CP) represents the coverage, or the part of the cache content that was prefetched and then required by the next FETCH query. The larger values for this, the better. This would never exceed 100% of the normalized user query.

The number of not predicted objects (NP) represents the part of the user query that was not predicted by the prefetcher and had to be demand-fetched or were there in the cache because of previous queries. The number of correctly predicted and not-predicted objects contribute to 100% of the user query.

Mis-predicted objects (MP) are the part of predicted objects that were incorrectly predicted and resulted in wasted bandwidth. Smaller values are better for this component. It indicates the accuracy of the prefetcher, because a more accurate prefetcher would fetch fewer objects that will not be used.

Note that different query sizes would result in different numbers for these three measurements. It is to normalize their values to allow proper comparison among the prefetch-

ing strategies. One way to normalize these measurements is given by the formula below:

$$\%CorrectlyPredicted(\%CP) = \frac{CP}{CP + NP} \times 100$$

$$\%NotPredicted(\%NP) = \frac{NP}{CP + NP} \times 100$$

$$\%MisPredicted(\%MP) = \frac{MP}{CP + NP} \times 100$$

Note that we have the divisor for mis-prediction as $(CP + NP)$ because we want to compare it with the actual required number of objects. The value for %MisPredicted can go beyond 100%.

5.4.3 Fitness Functions

In order to take into account all the performance measurements above, we need to come up with a function that would summarize the overall performance of a prefetcher with a single number. This function, which we call a fitness function (a term taken from the study of Genetic Algorithms [36]), is a function of one or more performance measures. We use it for the strategy selection policy to decide which strategy to select.

$$fitnessfunction = f(RT, CP, NP, MP)$$

The fitness function should take into account the performance of the prefetchers on several queries, and not just on one single query. For this research, we chose misclassification cost as our fitness function. Misclassification cost defined as cost associated with making a wrong classification (of required data) [50] is given by:

$$Cost = \frac{(C_{NP} \times NP) + (C_{MP} \times MP)}{CP + NP + MP}$$

where C_{NP} is the penalty assigned for not predicting required objects, and C_{MP} is the penalty assigned for mis-predicting unnecessary objects. We have set the value of $C_{NP} = 0.5$ and $C_{MP} = 0.5$. As can be noted from the formula, since the numerator of the fraction has variables that we want to minimize, the lower the value for misclassification cost, the better the prefetcher.

We chose misclassification cost for the following reasons:

- Since response time is affected by so many other external factors (as mentioned in Section 5.4.2), we decided not to directly include it in our fitness function. However, since response time is correlated with the percentage of not-predicted objects, it is therefore indirectly incorporated in the misclassification cost.

- Prefetching is like a statistical binary outcome prediction problem. From Statistics, one way to measure the accuracy of such a predictor is with the use of misclassification cost. Misclassification cost takes into account the two types of errors a predictor can make - not predicted and mis-predicted (Figure 5.3). The cost components (C_{NP} and C_{MP}) represent the weight given to each type of error.

One misclassification cost value can be calculated for each *prefetch* query. But we do not want to base the strategy selection on just the cost of the most recent query; instead we want to summarize the cost for a series of recent queries, giving more weight to the more recent queries over the older ones. For this, we use exponential smoothing:

$$localAvg[1] = misClassCost[1].$$

$$localAvg[t] = \alpha \times misClassCost[t] + (1 - \alpha) \times localAvg[t-1]$$

where α = smoothing parameter between $[0, 1]$. The choice of the smoothing parameter α dictates the aggressiveness of decaying older values. Use α close to 0 (e.g., 0.1) to

Strategy	Exponentially smoothed average misclassification cost	# times selected
No Prefetching	0.5	100
Random	0.4	25
Direction	0.3	225
Overall	0.4	350

Table 5.1: Strategy Selection Table

give almost equal weight to all values; use α close to 0.5 (e.g., 0.4) to gradually decay the values; and use α close to 1.0 (e.g., 0.9) to rapidly decay the values, giving very little weight to the older values [13]. For our system, we used 0.75 as the exponential smoothing factor to give more weight to the performance of the most recent queries. We summarize the strategy selection information in Table 5.1.

We initialize this table as follows: Since the individual prefetching strategies are supposed to compete against each other, it is best to put all of them on equal footing to begin with. One way to initialize the fitness function is to set their misclassification costs all to 0 (the best), setting all strategies to be equally good initially. This allows all strategies to be selected during the first few calls to prefetching. We also initialize the *number of times selected* to 1 for all strategies. The misclassification cost is calculated for the selected strategy after the next user query is served. The metrics for only that selected strategy are updated.

5.4.4 Strategy Selection Policy

A strategy selection policy is rule used to determine which among competing strategies to select. Several selection policies already exist, as mentioned in the Related Work section (Section 8.1). For our problem, we have applied two policies [36] that are in contrast to each other:

Select-Best: This policy chooses the strategy with the best fitness function value. If ties occur, we randomly choose among the tied strategies. One downside of this policy is that a single strategy might dominate the selection early in the process and no other strategies might get selected from then onwards.

SelectProp: This policy chooses a strategy with a probability proportional to its fitness function value (assuming that higher fitness value means better performance). In Genetic Algorithms [36], such a type of strategy is called "fitness proportionate selection". The general algorithm proceeds as follows:

- Let f_i be the fitness value of strategy i and let $\hat{f}$ be the overall average fitness for all the strategies i.e.,

  $$\hat{f} = \frac{\sum_{i=1}^{N} f_i}{N}$$

  where N is the number of prefetching strategies in the table.

- The probability of a strategy being selected is given by:

$$p_i = \frac{f_i}{(\sum_1^N f_i)} = \frac{f_i}{N \times \hat{f}}$$

In the case of fitness functions involving response time where a smaller value means better fitness, this algorithm is adjusted such that the probability of selection for a strategy is inversely proportional to its fitness function.

Select Proportionate [36] strategy allows for some degree of exploration, i.e. allows currently lesser performing strategies to be selected and executed. This is designed to prevent a single strategy from dominating the selection early in the process.

Chapter 6

Implementation of Prefetching Architecture

6.1 System Architecture

The caching as well as prefetching strategies described above have all been fully implemented and incorporated into XmdvTool 5.0, a freeware package for the interactive visual exploration of multivariate data sets developed at WPI [53]. A high-level diagram of the modules added and the relationships between these modules and the original system is depicted in Figure 6.1. XmdvTool 4.0 was coded in C++ with Tcl/Tk and OpenGL primitives. The newly added modules are written in C with Pro*C (embedded SQL) primitives for Oracle8i. First, an off-line process clusters the flat data sets and then transforms the hierarchical data into MinMax trees [43], a pre-coded indexing structure that allows us to express hierarchical navigation as range queries. The transformed data is then loaded into the database.

The interaction with the original system (shown as GUI in Figure 6.1) has been encapsulated as a database access API. We have implemented what we called a prepare/iterate

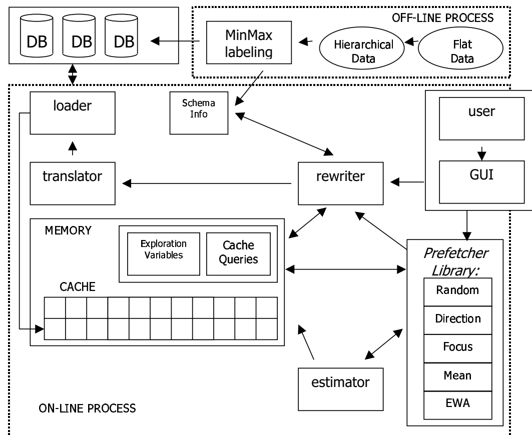

Figure 6.1: System architecture. Dotted-line rectangles show the separation between the on-line and the off-line computation. Solid-line rectangles represent the modules. Ovals represent data. Arrows show the control flow.

paradigm. Thus when the user issues a new request, the front-end only has to inform the back-end about the request by calling the *prepare* function. After that it can retrieve the desired objects one at a time from the buffer by repeatedly calling the *iterate* function. If all the objects needed were unable to fit in the cache, the process *prepare* function is called again to load the remainder of the data. The *iterate* function again iterates to display the remainder of the data. This process continues until all the data to be displayed are iterated.

Our semantic cache indexes the cache content with descriptors of the queries used to retrieve the cached data. This allows for a fast look-up, since only a few set-based operations are performed to compare a new query against cached queries (see Chapter 3). The caching architecture forms the foundation for the implementation of all of our prefetching strategies. More details of our caching system can be found in [17].

When the system is idle, the *prefetcher* thread is created which communicates with the *estimator* to make decisions about the next most probable data to be prefetched depending on the selected prefetching strategy and the current cache content. The visual navigation operations effect a change in the active set, a *producer* thread is created, while the GUI itself acts as *consumer*. Basically, two threads operate concurrently on the buffer data: the consumer and the producer. The active set information from the GUI is passed to a

rewriter. The prefetching query is then passed to a *rewriter*. The *rewriter* consults the semantics of the cache (expressed by cached queries) and generates a set of sub-requests to incrementally adjust the data in the cache. Each sub-request is transformed into an SQL query by the *translator*. The queries are passed to the *loader*, which fetches the necessary objects from the database using a cursor and places them in the cache. From here, the *reader* reads them and sends them to the display. Once the *reader* is done with the reading of data, if time permits, the prefetcher will signal the *rewriter* to prefetch the next most probable data depending on the current prefetching strategy. Whenever the cache is full, the *estimator* removes the objects from the cache by examining probability values that depend on their semantic distance from the active region.

When the user uses the structure-based brush (see Figure 2.3) to navigate to some other data in the cluster hierarchy, then the fetching process is started by the *main* thread to service the explicit user request. If the *prefetcher* thread is still running in the system, it is preempted and the contents of the cache are adjusted for consistency.

6.2 Adaptive Prefetching Architecture

Figure 6.2 shows the diagrammatic representation of the components involved in strategy selection. As shown, we have a list of prefetching strategies. Among the strategies that we have in the list are *no prefetching* strategy, *random* strategy, *direction* strategy and *focus* strategy. These strategies may require some form of summarization of the past user exploration for prediction. For example, in the *direction* strategy, we keep track of the past two user operations to figure out the direction of user movements. The *focus* strategy requires holding statistics about all the past explorations so that it can figure out the regions of interest to the user.

We also maintain a strategy table that keeps statistics to evaluate the performance

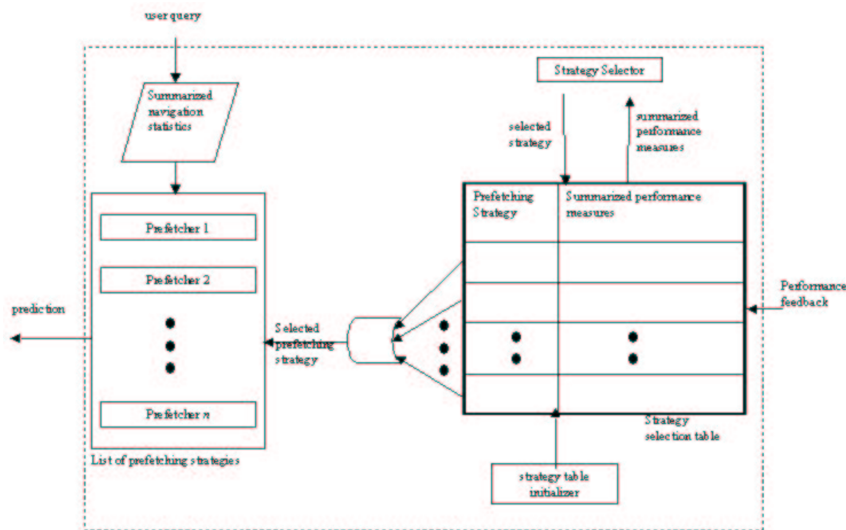

Figure 6.2: Adaptive Prefetching Framework.

of each prefetching strategy. The summarizations include keeping track of correctly predicted objects, not predicted objects, mis-predicted objects, local average mis-classification cost and number of times each strategy got selected. These values are initialized by the strategy table initializer. Note that we can give higher preference to a strategy by setting appropriate values for local average mis-classification costs for individual strategies. These summarizations are updated when the user query is completed and the prefetcher has to select the next prefetching strategy for carrying out prefetch operations.

Whenever the system is idle, the strategy selector looks at these performance measures and depending on the strategy selection policy, it selects one of the prefetching strategies from the list. The selection policy can either be *select-best* or *select-proportionate*.

Chapter 7

Experimental Evaluation

7.1 Experimental Inputs

Performance measurements in systems where human interaction is needed require logging the information about their usages in order to compare the performance for different settings for the same sequence of operations. The same sequence of operations has to be performed multiple times, hence we capture the activities performed by the users in a log file. User input (also referred to as a user script or input script from now onwards) is a sequence of user operations and the time specification of when the operations occur. In XmdvTool, the user input is a sequence of navigation primitives and delays. An example of a user script is presented below:

```
1000  3  0.050000
1200  1  0.100000
1400  1  0.150000
1600  2  0.200000
```

The first column contains the time specification, the second the type of the operation and the third the parameters that the operation requires.

Our experimental results are primarily based on real-user traces. We performed a user-study where we collected traces from a number of users of our system. These traces consisted of 30 minutes each for 20 different users. These users were given real data sets to find patterns and outliers in the data. The people who were unfamiliar with the tool were given a practice session to familiarize them first. These traces were then given as input to the tool and system settings such as prefetching strategies were varied. The values recorded were averaged out for the same settings respectively.

We also generated simulated user traces for simulating the operations performed by the user. The main goal for our input simulation is to provide "data specificity" and "user specificity" to the scripts that we generate. These particularities provide the hints that we exploit when prefetching.

7.2 Settings

All experiments were conducted on an Windows 2000 PC, running Oracle 8.1.7. We used C as the host language and embedded SQL statements for accessing the data in the database.

The dataset we used has 4095 datapoints and 8 dimensions, and 2,048 (2^{11}) objects as the maximum number of points displayed at a time. The dataset was named: D2k. In all experiments we used navigation scripts containing between 300 and 4000 user operations.

The values we measured during the experiments were: mis-classification cost, predicted objects, network load and response time (latency). The *mis-classification cost* is our objective function that we want to optimize by our prefetching framework. Since it represents the prediction errors made by a prefetcher, we would like it to be as low as possible. The *predicted objects* are further divided into three types; namely, *correctly predicted*, *not predicted* and *mis-predicted* objects. These measures are as described in

Section 5.4.2. The *network load* indicates the number of objects received from the server. It is important as the prefetching strategies sometimes would prefetch wrong data that are not required, thus leading to a rise of unnecessary objects from the server. This value should be lowest for an ideal prefetcher. The response time is the total time, expressed in seconds, in which the user had to wait for her requests to be served, i.e., the on-line loading time.

7.3 Experiments Related to Strategy Refinement

In order to adapt the *focus* strategy, we varied the hot-region update frequency mentioned in Section 5.3 to figure out the best speed of updating hot-regions for adaptive focus strategy. Below are the charts that depict the performance of adaptive focus strategy for various different update frequencies.

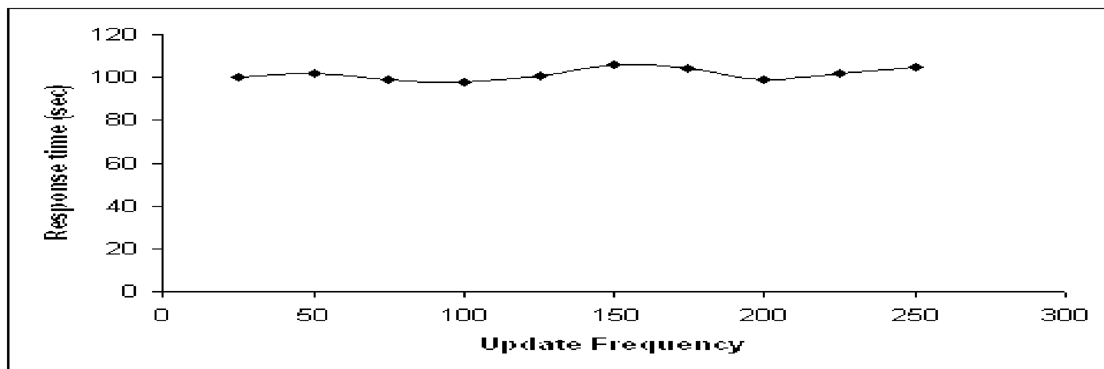

Figure 7.1: Response time vs. hot-region update frequency

As seen from the Figure 7.1, the changes in response time (Y-axis) is minor compared to the change in hot-region update frequency (X-axis indicates the update frequency in terms of number of user operations). Even with higher computation overhead for faster updates (lower values on X-axis), the response time still remains unchanged.

Figure 7.2 depicts the prediction rates for the adaptive focus strategy. Again, these

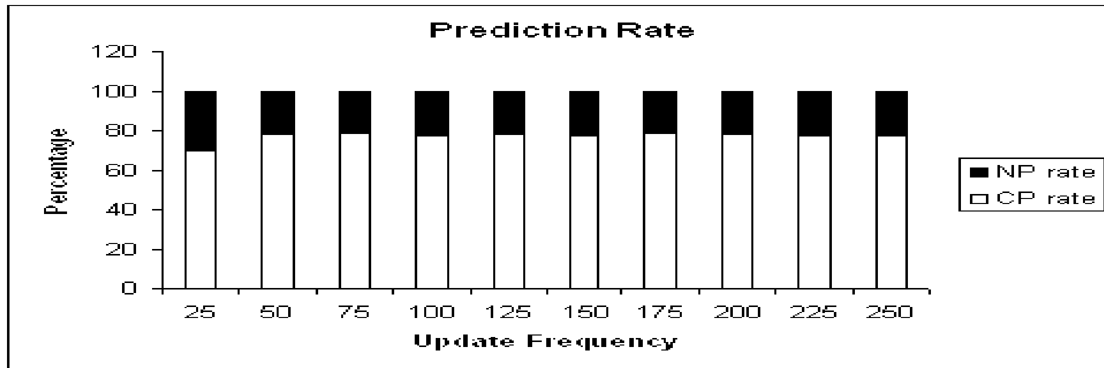

Figure 7.2: Prediction rates vs. hot-region update frequency

rates remain fairly similar with the change in update frequency. These charts show that the performance of athe daptive focus strategy for this user remains fairly constant with the change in update frequency. This suggests that in order to come up with a definite conclusions, we would have to test adaptive focus strategy for variations in update frequency for different users.

7.4 Experiments Related to Strategy Selection

We claim that different users have different ways of navigating because they have their own techniques of finding patterns for the same data. These differences in behaviors also infer that the same prefetching strategy might fail for different users. In addition to this, the user navigation pattern may also change during the same session. This is intuitive, as the user goes on navigating and learning more about the data, he would have different ways of navigation to understand the overall data and some particular characteristics about the data. In what follows, we examine the user traces in our experiments.

Below, we show a detailed analysis and results for two real user traces log files.

7.4.1 Case Study I

To begin with, let's look at user A's characteristics so that we can know details about the navigation pattern of this user.

User A Characteristics

The following four charts present the characteristics of a typical real user. As mentioned in Section 7.1, we gathered the trace from the users who participated in our user study. For anonymity, let's call this user A.

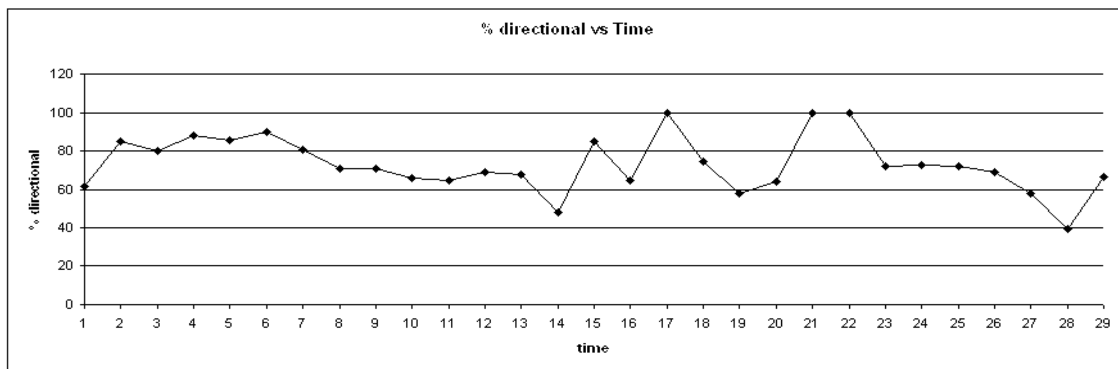

Figure 7.3: Directionality vs. Time for User A

Figure 7.3 shows the directionality for user A at any given time during the session. We define directionality as the percentage of times in the session for which the user continues to move in the same direction as his last movement. This can be achieved by using different extents in the brushing tool we described in Section 2.2. Also, remember that there are only 4 directions permitted for movements in XmdvTool. As can be seen from the charts, the user's directionality changes a good deal, from being 100% directional at times to being 40% directional at times. But, on an average he stays 73% directional. This shows some difference in user navigation characteristics.

As shown in Figure 7.4, the user speed (number of user inputs per minute, in our case) also changes over time. Sometimes, the user is fast enough to generate around 105

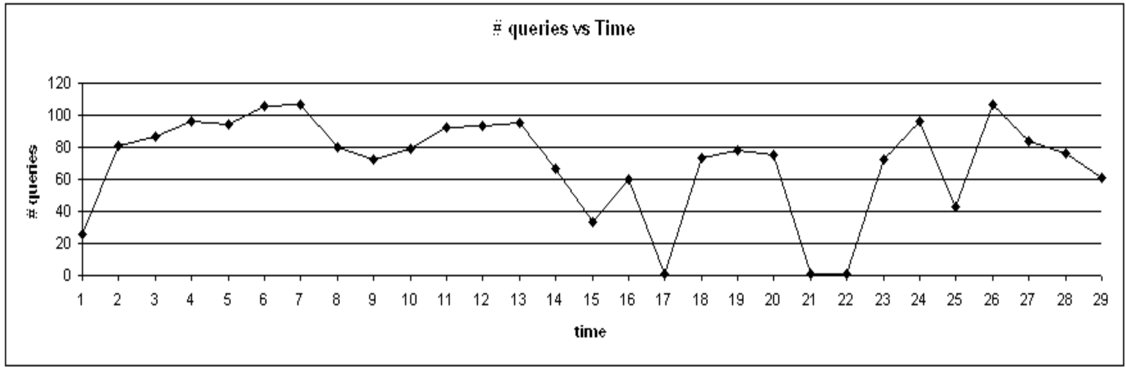

Figure 7.4: Number of Queries vs. Time for User A

movements in a minute and sometimes he stays idle for 2 whole minutes. On an average he has 70 movements in a minute.

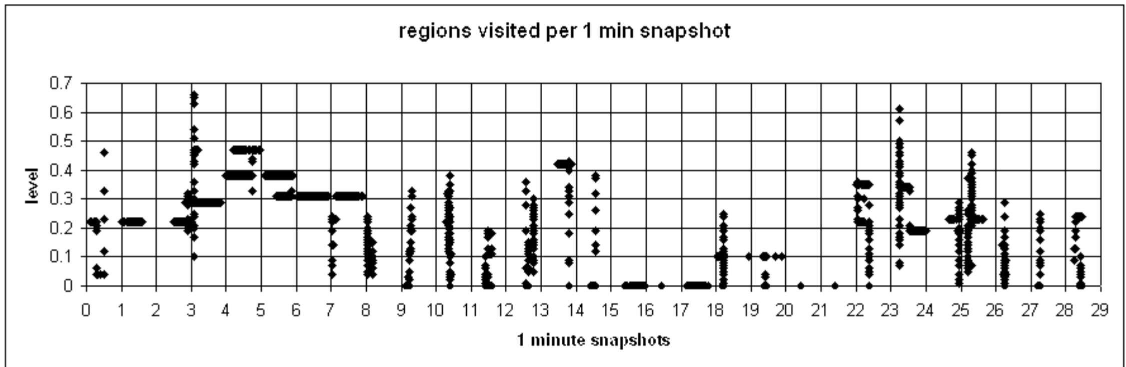

Figure 7.5: Regions visited vs. Time for User A

Figure 7.5 depicts the regions visited by the user per minute. To better understand the graph, consider each vertical column as the structure-based brush in our tool. The spots in those grids indicate the regions visited during that minute. As seen from the chart, the user movements are local as all the navigation points are clubbed together for a given 1 minute time frame, confirming our hypothesis that users tend to explore a particular region before switching to the next location for exploration.

Figure 7.6 illustrates the movements by the user in both horizontal (left/right direction) as well as vertical (up/down direction). The lower line indicates horizontal movements

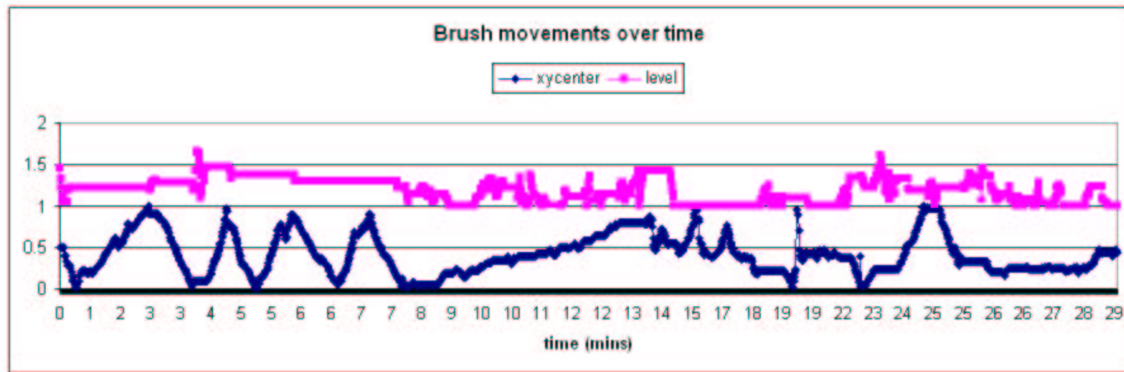

Figure 7.6: Brush movements vs. Time for User A

by the user. The value 0 means the user is at the leftmost extent and value 1 means he's at the rightmost extent. The upper line indicates the vertical movements by the user. Value 1 indicates the user is at the lowest level of detail (drill-down operation) and value 2 indicates the user is at the highest level of detail (roll-up operation). As seen from Figure 7.6, the user analyzes data horizontally most of the time with hardly any drill-down or roll-up operations initially. Then he starts analyzing the same data at different level of detail, and so on.

Now, let's try to analyze the performance of different prefetching strategies for user A's trace. This would give us an idea about the performance for each individual prefetching strategy as well as for adaptively selecting between these strategies.

Performance for User A

As demonstrated in Figure 7.7, the *direction* prefetching strategy gives the best mis-classification cost compared to *no prefetching* as well as *random* strategy. So it is the best performer amongst static prefetching strategies. This result was expected as the user appears to be directional (73% as noted in Figure 7.3). Also note that for adaptive prefetching, the mis-classification cost is even better as it selects the best strategies, namely, *direction* and *no prefetching* strategies, for improving the performance further.

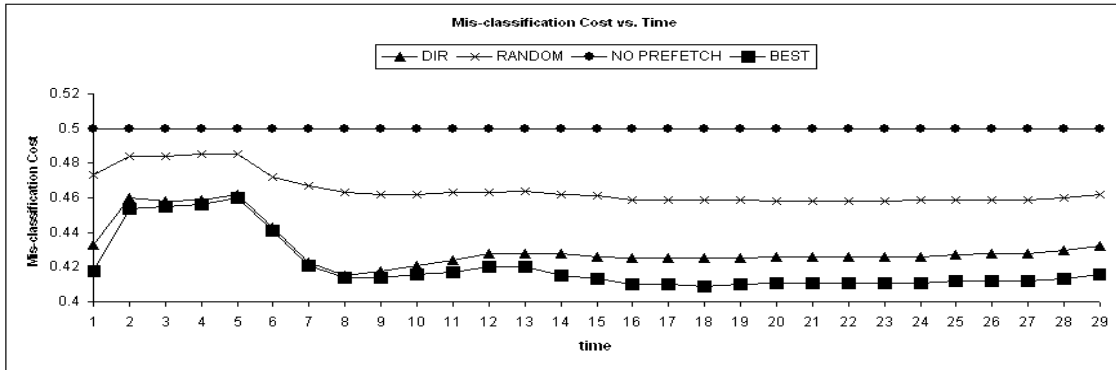

Figure 7.7: Mis-Classification Cost vs. Time for User A

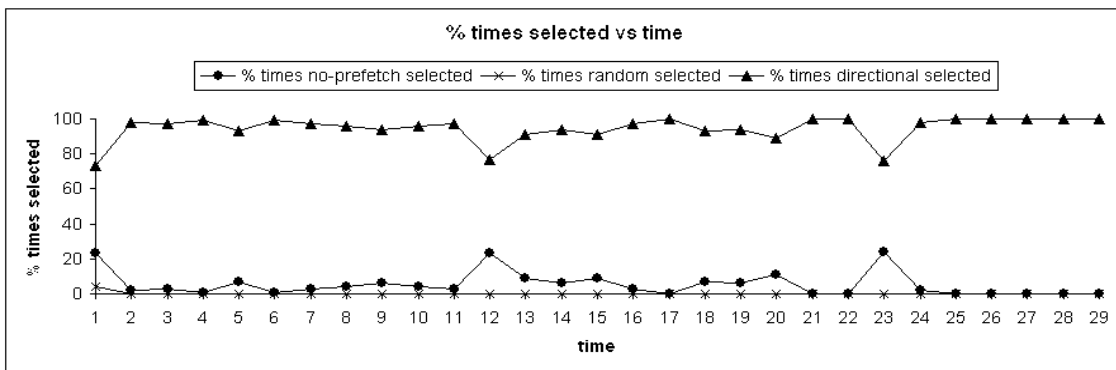

Figure 7.8: Strategy selection vs. Time for User A

Strategy selection over time for user A is exemplified in Figure 7.8. Since the *direction* strategy is best for him, we note that it is the one that gets selected most of the time. Also, sometimes the *no prefetching* strategy is selected as the user appears a bit random at those time instances and the movements are quick, thus indicating not to prefetch in order to reduce the number of objects mispredicted. Note that the *random* strategy is hardly ever selected as its mis-classification cost is high.

As seen from Figure 7.9, *adaptive* prefetching still has the lowest % of not predicted objects. The performance is even better compared to *direction* strategy as that's the strategy selected most of the times by it and *no prefetching* to further improve the performance when *direction* strategy fails.

Figure 7.10 shows that for % of mis-prediction, *adaptive* prefetching has bad perfor-

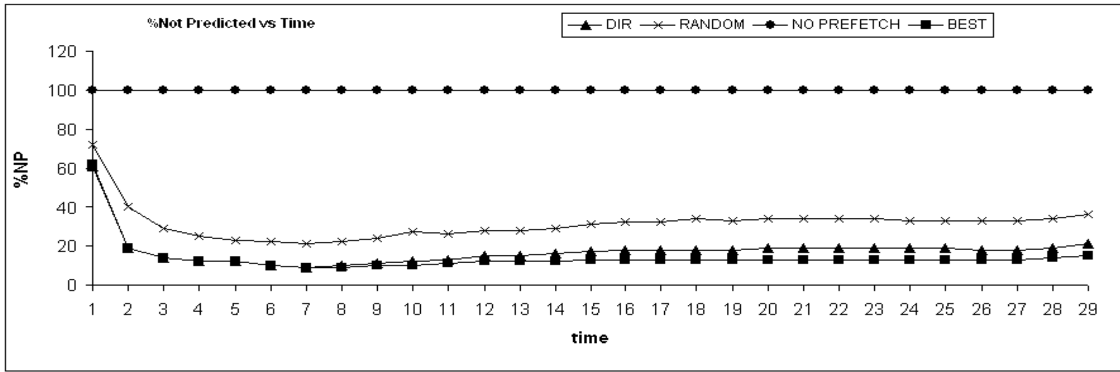

Figure 7.9: % of Not predicted objects vs. Time for User A

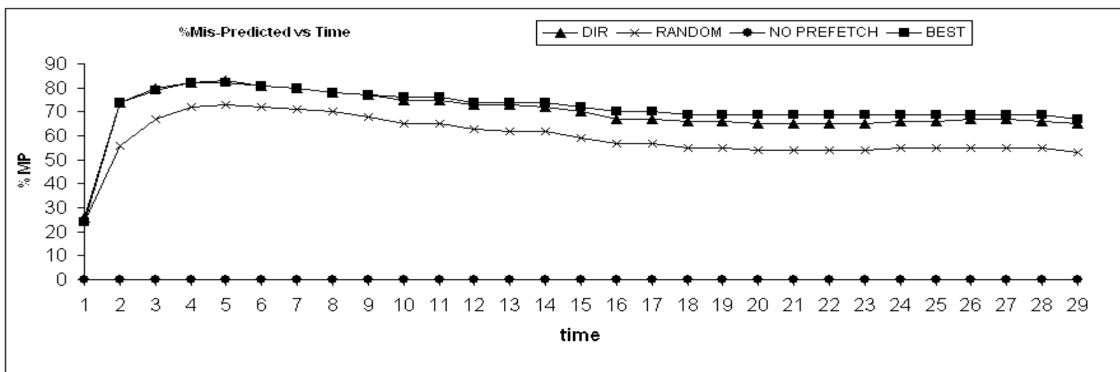

Figure 7.10: % of Mis-predicted objects vs. Time for User A

mance. This says that mis-classification cost is a trade-off between the two measures: *not predicted objects* and *mis-predicted objects*. Depending upon values of C_{NP} and C_{MP}, one is chosen over the other.

As depicted in Figure 7.11, % of correctly predicted objects is also high as an effect of improvement in mis-classification cost. Also from Figure 7.12, response time is better for adaptive prefetching. But as we will see from the next section, this is not always the case for all users.

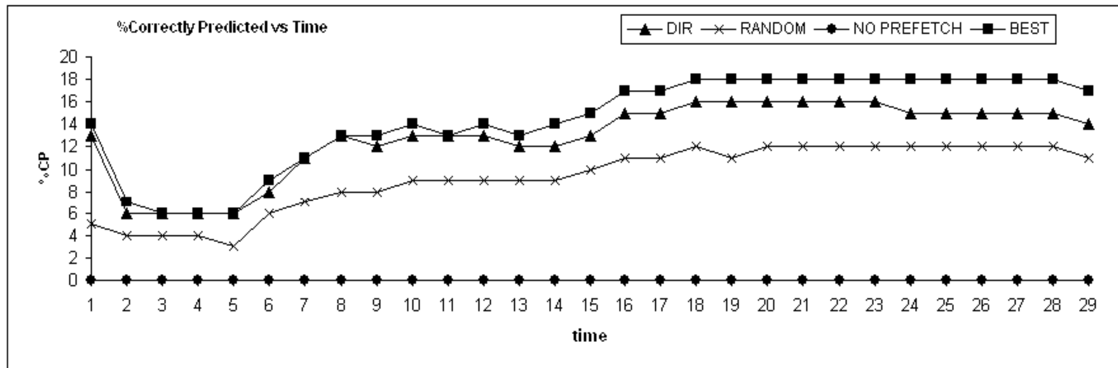

Figure 7.11: % of Correctly predicted objects vs. Time for User A

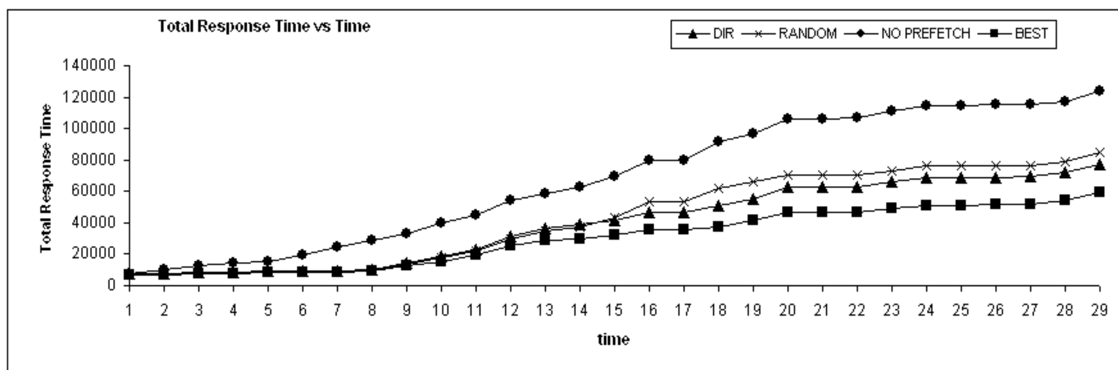

Figure 7.12: Response Time vs. Time for User A

7.4.2 Case Study II

Let's look at the characteristics of user B to compare the behavior of this user with the previous user.

User B Characteristics

Figure 7.13 gives the directionality for user B at any given time during the session. As can be seen from the chart, the user's directionality on average stays around 40%. This is very much different from user A. Also, note that change in directionality for this user is pretty steady throughout.

As shown in Figure 7.14, the user speed again changes with respect to time. Compared

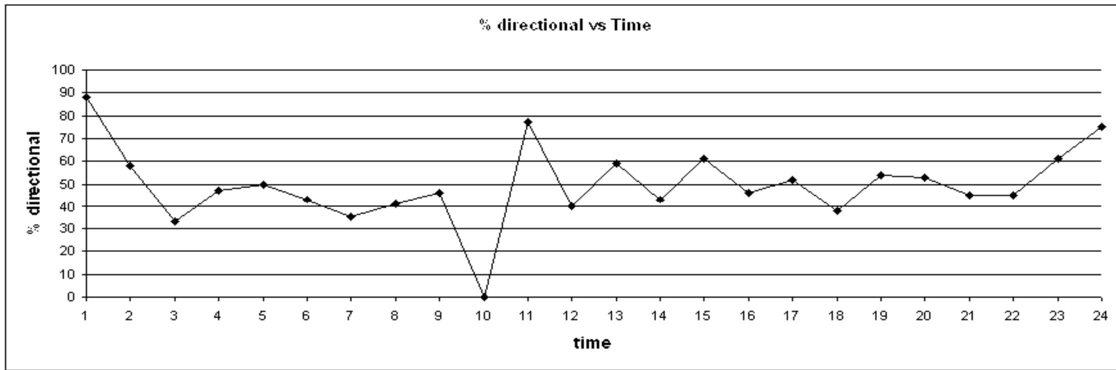

Figure 7.13: Directionality vs. Time for User B

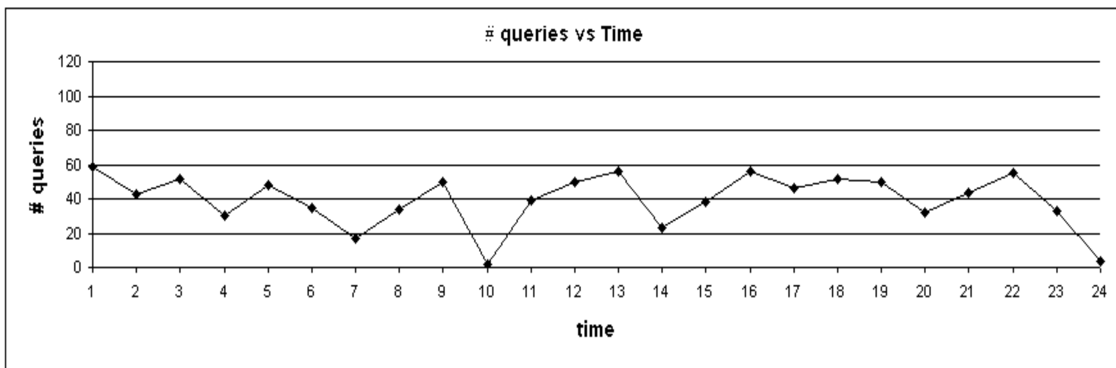

Figure 7.14: Number of Queries vs. Time for User B

to user A, user B is pretty slow and steady with an average of 40 movements per minute.

Figure 7.15 depicts the regions visited by the user per minute. Note that unlike user A, he navigates vertically most of the time, viewing the data at various levels of abstraction for every chosen subset of data.

Figure 7.16 illustrates the movements by user B in horizontal and vertical directions. As seen from the figure, the user analyzes data vertically most of the time, very slowly selecting different subsets and visualizing them at different level of details.

Performance for User B

As demonstrated from Figure 7.17, among the static prefetching techniques, the *random* strategy gives the best mis-classification cost compared to *no prefetching* as well as *direc-*

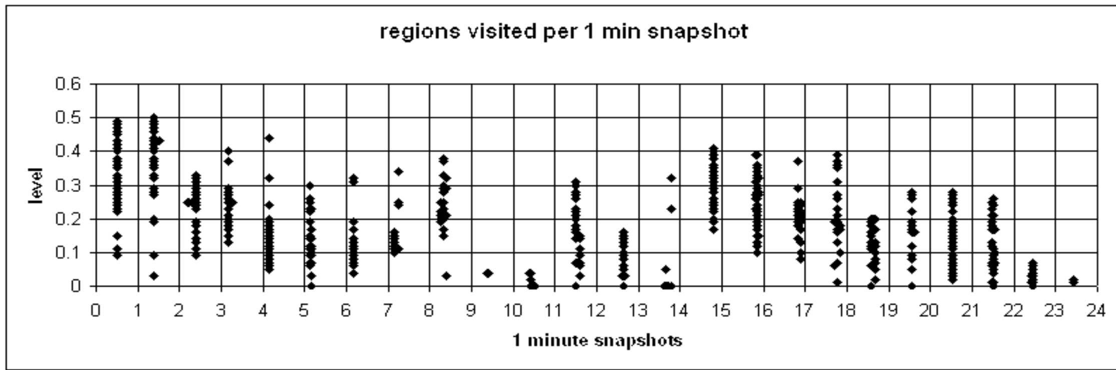

Figure 7.15: Regions visited vs. Time for User B

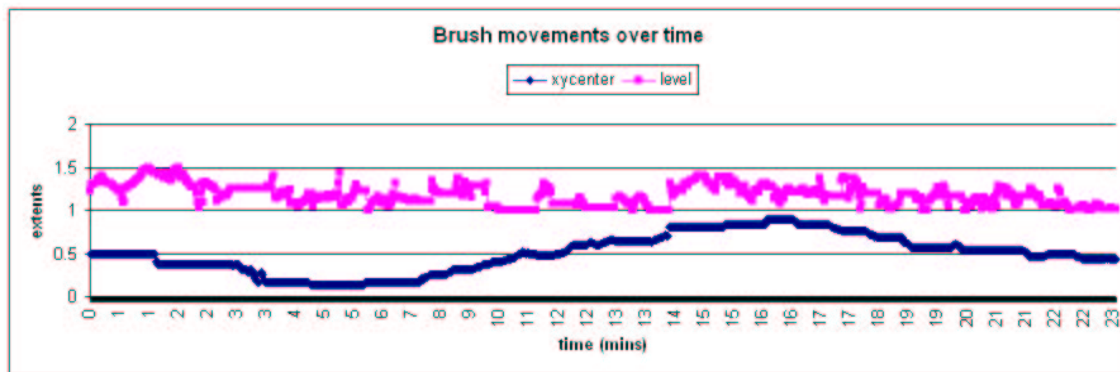

Figure 7.16: Brush movements vs. Time for User B

tion strategy. This is different from what we observed for user A where *direction* strategy had better performance. This confirms our claim that no single prefetching strategy wins for all the users. Also note that for adaptive prefetching, the mis-classification cost is even better as it selects the best strategies for improving the performance further.

Strategy selection over time for user B is shown in Figure 7.18. Since *random* strategy is best for him, it is the one that gets selected most of the time. Also, sometimes *no prefetching* strategy is selected when the movements are quick, thus indicating not to prefetch in order to reduce the number of objects mispredicted. Note that the *direction* strategy is hardly ever selected as its mis-classification cost is high.

As seen from Figure 7.19, *adaptive* prefetching still has the best % not predicted objects, though the difference is not great compared to *direction* strategy as that's the

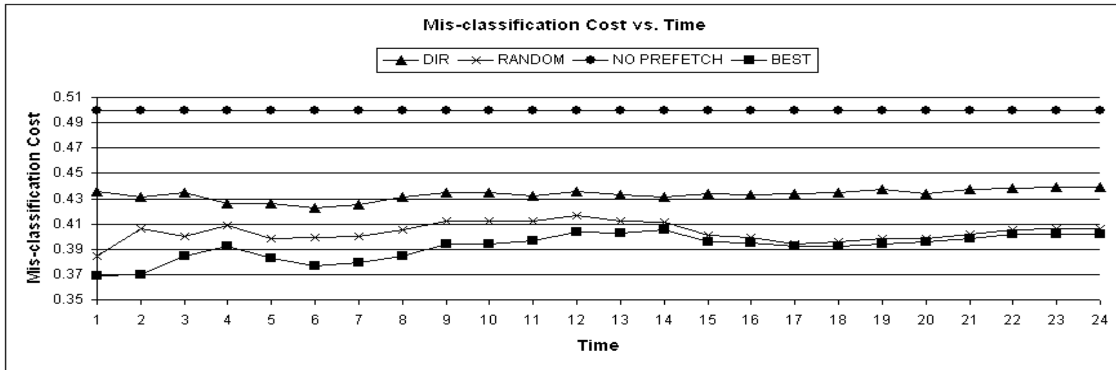

Figure 7.17: Mis-Classification Cost vs. Time for User B

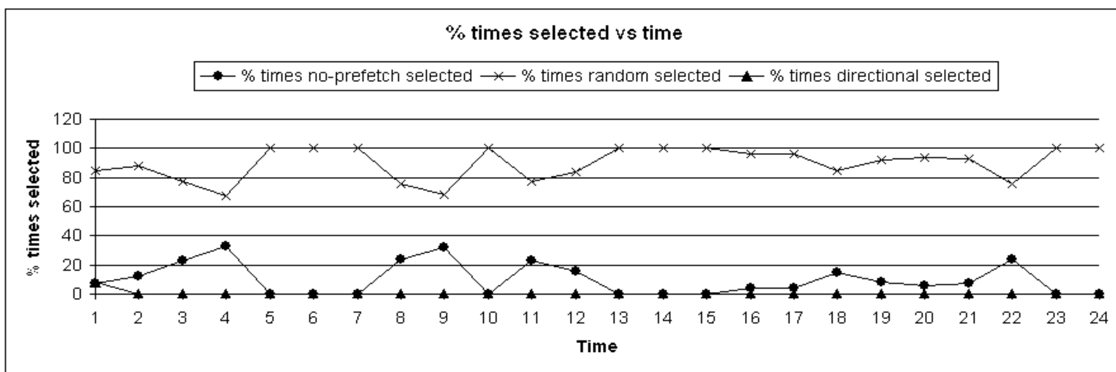

Figure 7.18: Strategy selection vs. Time for User B

strategy selected most of the times by it.

As seen from Figures 7.19 and 7.20 again, mis-classification cost is a trade-off between *not predicted objects* and *mis-predicted objects*. These charts are analogous to the same for user A (Figures 7.19 and 7.20).

As depicted in Figure 7.21, % of correctly predicted objects for *adaptive* prefetching is nearly the same as for the *random* strategy. Also from Figure 7.22, the response time is similar for all the strategies. This indicates that adaptive prefetching might not help to improve the performance to a good extent if individual prefetching strategies do not have good improvements.

The strategy selection done for user A resulted in the combination of direction strategy with no prefetching. On the other hand, the strategy selection done for user B resulted

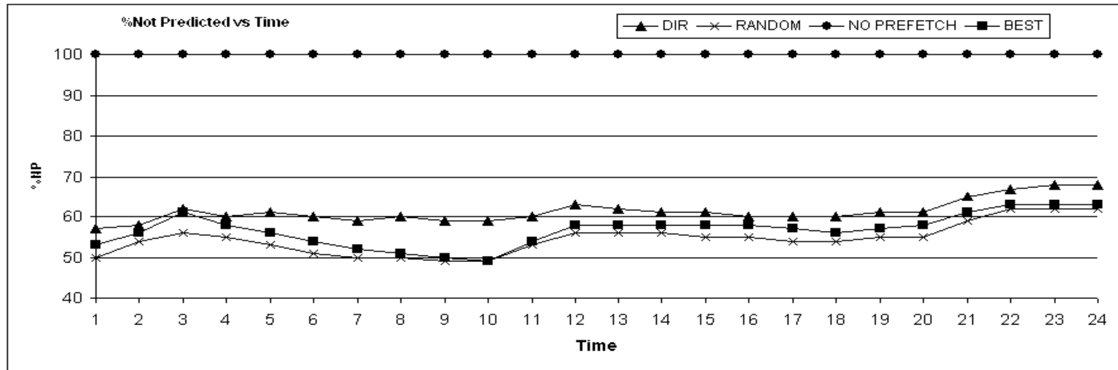

Figure 7.19: % of Not predicted objects vs. Time for User B

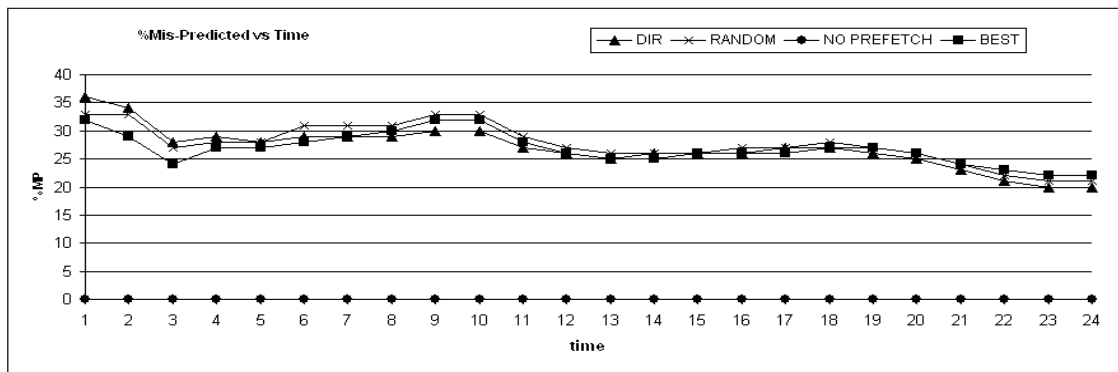

Figure 7.20: % of Mis-predicted objects vs. Time for User B

in the combination of random strategy with no prefetching. In both cases, no prefetching was chosen to compensate for the high mis-prediction done by direction and random strategies. This suggests that one could also adjust the amount of data prefetched (i.e. prefetch step size) to achieve less % of mis-predicted objects for both direction and random strategies.

7.4.3 Summary Charts

The following experiments were performed in collaboration with Rosario. To summarize the performance of our adaptive prefetcher, we investigate its effect on 14 real user traces. We did experiments on all the user traces listed in Section 7.1. To remove the possible

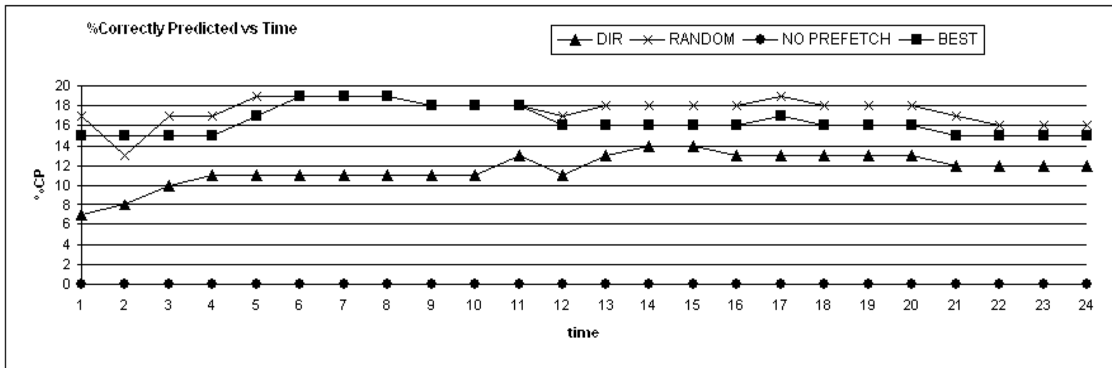

Figure 7.21: % of Correctly predicted objects vs. Time for User B

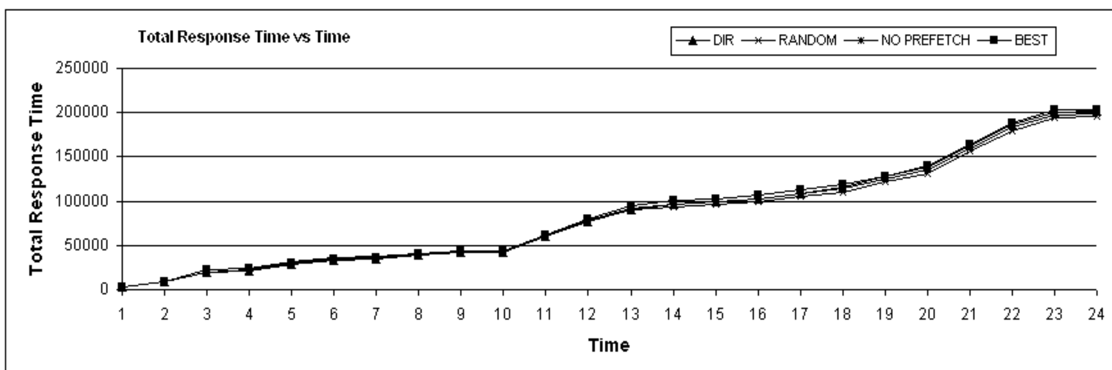

Figure 7.22: Response Time vs. Time for User B

effect of network traffic on the performance measures, especially response time, each experiment was repeated 3 times and then the measures were averaged. To get an overall picture of the performances on each type of user trace, the results are summarized for each user cluster.

Figures 7.23 and 7.24 show the global average mis-classification costs and normalized response times for different prefetching strategies, summarized for the 3 user clusters. Recall that cluster 1 users are the random-starers, cluster 3 users are the directional-movers, and cluster 2 users are the indeterminates.

Figure 7.23 shows that, on average, strategy selection improves the misclassification cost for the random-starers and directional-movers, and only improves slightly for the indeterminates. This could be due to fact that the random and directional prefetchers

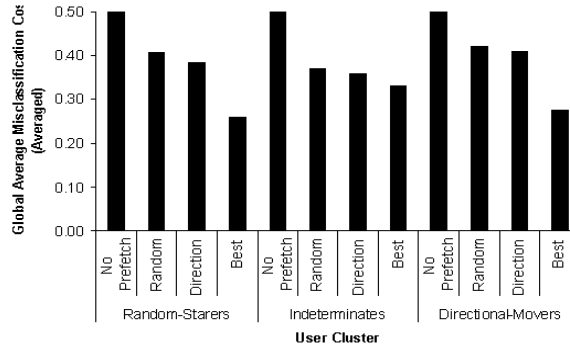

Figure 7.23: Global Average Mis-classification Cost (Averaged) For Different User Clusters

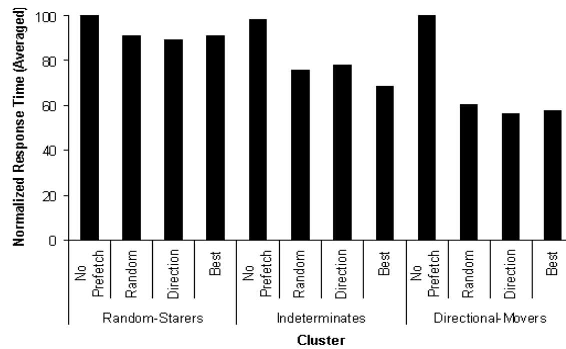

Figure 7.24: Normalized Response Time (Averaged) For Different User Clusters

do not have direct control over how much they prefetch and thus have large number of mispredictions. With strategy selection, there is the option to switch to 'no prefetching' when the movements become more random and more frequent, thus minimizing the number of mispredictions which in turn minimizes misclassification cost. This observation leads us to consider refining the random and directional prefetchers to allow the prefetch step size to change over time. For indeterminate users, strategy selection improved the misclassification cost only slightly compared to the static prefetching strategies.

Figure 7.24 shows that, on average, strategy selection does not improve the response time compared to the static prefetchers for the random-starers and directional-movers. Recall that response time is affected by several external factors (listed in Section 5.4.2).

As such, the exact reason behind these summary patterns is hard to pinpoint.

We also tried the SelectProp (fitness-proportionate) strategy selection policy (described in Section 5.4.4) to investigate if a more exploratory approach (SelectProp) is better than a greedy approach (SelectBest). Our experiments showed that SelectProp yielded worst results compared to SelectBest.

Chapter 8

Related Work

8.1 Adaptive Prefetching

There has been much research performed on adaptive prefetching for different applications. We list some of this research here and point out both how these have influenced our approach and differences to our approach.

Davison et al. [14] proposed a solution for predicting the next user command in the Unix shell prompt by using simple Markov chain predictors. These statistics, collected for each user, are aged over time in order to emphasize the recent commands by the user. This solution utilizes the concepts of strategy refinement and information aging. It has provided us with inspiration for extending the static focus strategy into an adaptive focus strategy.

Tcheun et al. [46] proposed an adaptive sequential prefetching scheme for hardware, which is similar to our static *direction* strategy (analogous to sequential prefetching in many of the I/O prefetching strategies). It adapts to a user's step size to make sure that only the best data gets in the memory. This solution utilizes strategy refinement but not strategy selection. We note that the user's step size can be one of the base parameters that

can be adapted in our approach; however, we did not get a chance to experiment with it so we list it in the Future Work section.

Some research efforts utilize the concept of learning (instead of strategy selection nor strategy refinement) in deciding the next adaptive action to take. Srikant et al. [1] present a data mining approach to gathering sequential patterns about time-series data. This algorithm can also be used to predict the next user movement. [44] discusses a model for capturing user behavior that may be necessary to adapt to the changes in the user patterns. Learning is one potential enhancement for adaptation that we currently do not apply in our system.

We note that these research efforts have focused on taking one strategy and adapting it (i.e., strategy refinement). In our approach, we utilize strategy selection in addition to strategy refinement.

[7] describes statistics collection for database management systems that keeps track of data access characteristics (such as data pages) of the client machine. This is analogous to our model of user trace collection to keep track of the access patterns by the user.

Several strategy selection policies already exist in various fields, including Genetic Algorithms (fitness-proportionate selection, sigma scaling, elitism, rank selection, tournament selection, steady-state selection) [36] and Operating Systems (lottery scheduling [47]). For our approach, we chose fitness-proportionate selection because it allows for exploration (vs. exploitation), and it is quick and easy to calculate (minimal overhead). However, other selection methods may need to be experimented in the future to assess this trade-off in overhead vs. potential performance gain.

For performance evaluation, we extracted the idea of using mis-predicted/not predicted/correctly predicted statistics from [28]. The idea of a fitness function in strategy selection is inspired by fitness functions in Genetic Algorithms [36]. Furthermore, since prefetching boils down to a statistical classification problem, we looked into the Statistics

field for ideas for fitness function and found misclassification cost [50].

8.2 Caching

Semantic caching is used for client-side caching and replacement in a client-server database system. It is aimed largely at providing support for navigational access to data (such as visualization applications). We have implemented a caching structure inspired by [30, 12] as it provides efficient support for access to data. In particular, we have developed a hash-based look-up structure that allows replacement at the object-granularity level [43]. Though caching is necessary for visualization applications and necessarily a prerequisite for support of prefetching, our research reported here concentrates on the trade-offs between different prefetching strategies.

8.3 Database Support for Interactive Applications

Much work has been done in recent years on visual interaction tools, including [40, 45, 26, 15, 25, 29, 27, 24]. Integrated visualization-database systems such as Polaris [40], Tioga [41], IDEA [39] and DEVise [32] represent the work most closely related to ours in terms of developing tools for visual data exploration support. The specific approaches taken are however different.

Polaris [40] is an interface for exploring large multi-dimensional databases that extends the well-known Pivot Table interface first popularized by Microsoft Excel. 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 from the visual specifications. But Polaris does not include caching and prefetching to improve the performance.

Tioga [41, 2] implements a multiple browser architecture for a *recipe*, a visual query. The problem of translating front-end operations into database queries is not present since database queries are explicitly specified by the graphical interface. Also, they do not cache the queries in their system. VIDA [52] is a visualization tool (an extension of DataSplash [2]) that is developed in an attempt to solve the cluttering problem by providing *goal-directed zooming*.

Infostill [10] is a data analysis application that focuses on assisting users with all stages of data analysis. It does not take client-side caching into consideration for improving the performance of the tool. IDEA [39] is an integrated set of tools that supports interactive data analysis and exploration. This tool focuses on multiple display views like XmdvTool, but on-line query translation and memory management are not addressed in that work. In DEVise [32], a set of query and visualization primitives to support data analysis is provided. The number of primitives supported is relatively large. However, caching data is done at the database level using the default mechanisms only; special memory management techniques as in our work are not studied.

Unlike prior work, we aim to focus on the interactions between the two areas: Visualization and Databases. In particular, we worked on adaptive prefetching using strategy selection in this context.

Chapter 9

Conclusions and Future Work

9.1 Summary and Conclusions

We first designed and implemented simple static prefetching strategies that improve the performance of the system in terms of response time. We utilized a high level caching [42] that reduces the system response time by incrementally loading the data into the cache. We developed a framework for adaptation for prefetching in any interactive visualization system.

From our experiments, we have shown that prefetching is always better than no prefetching, as it helps in improving the response time. But prefetching also comes with mispredicted objects that lead to a rise in network traffic.

Our case studies showed that different users have different navigation patterns in our system. In addition to that, we have also shown that for the same user, the navigation pattern vary within the same session. These claims have been analyzed by our case studies. We have also seen that different prefetching strategies work well for different users. For this reason, different prefetching strategies are better at different instances of time during the same user session.

The choice of fitness function is important in evaluating the success of a strategy selection approach. In prefetching, the choice of fitness function is tricky because (1) it requires a trade-off between several competing performance measures, and (2) it begs the question of how much weight should be given to more recent performance information. We have experimented with response time as a fitness function and also figured out the drawbacks of using it as explained in Section 5.4.3.

Overall, we have shown that allowing the choice of prefetching strategy to change over time is better than hard-coding the choice upfront, even when there is some overhead inherent due to maintaining statistics for strategy evaluation in the selection process.

9.2 Future Work

In the experimental section, we have noted possible next steps in improving the quality of prefetchers based on the results we saw. In this section, we summarize all these next steps as well as list other possible options.

For both direction and random strategies, we can adjust the amount of data prefetched (e.g., by adjusting the prefetch step size or having a fixed prefetch buffer size).

We can explore other strategy selection policies and fitness functions. There are several other strategy selection policies as listed in the Related Work section, as well as several other ways to combine the performance measures and produce a fitness function.

Comparison of the performance of strategy selection against strategy refinement can be one of the works that might bring to light some new points. Other research on adaptive prefetching have mostly focused on refining a single strategy. Which adaptation approach works better for adaptive prefetching for visual data exploration tools?

In our current implementation, we attempt to prefetch after every user query. This may not be good especially when the user is moving very fast and hence user queries

are issued one after then other quickly. We can defer prefetching in order to improve the performance of the system.

In situations when the data sets are very large and the cache size is limited, one could measure the performance of a prefetcher based on the accuracy of the prediction and not based on the actual number of predicted objects that got loaded into the limited cache.

Bibliography

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In P. S. Yu and A. L. P. Chen, editors, *Proc. 11th Int. Conf. Data Engineering, ICDE*, pages 3–14. IEEE Press, 6–10 1995.

[2] A. Aiken, J. Chen, M. Lin, and M. Spalding. The Tioga-2 database visualization environment. *Lecture Notes in Computer Science*, 1183:181–190, 1996.

[3] P. Andreae, B. Dawkins, and P. O'Connor. Dysect: An incremental clustering algorithm. *Document included with public-domain version of the software, retrieved from Statlib at CMU*, 1990.

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

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

[6] D. Calvanese, G. D. Giacomo, and M. Lenzerini. On the decidability of query containment under constraints. In *Proc of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symp on Princ of Database Systems, Seattle, Washington*, pages 149–158. ACM Press, 1998.

[7] C.-L. Chee, H. Lu, H. Tang, and C. V. Ramamoorthy. Adaptive prefetching and storage reorganization in a log-structured storage system. *IEEE*, 10(5):824–838, 1998.

[8] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In F. N. Afrati and P. Kolaitis, editors, *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece*, volume 1186 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 1997.

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

[10] K. Cox, S. Hibino, L. Hong, A. Mockus, and G. Wills. Infostill:a task-oriented framework for analyzing data through information visualization. In *IEEE Information Visualization Symposium 1999, Late Breaking Hot Topics*, pages 19–22. ACM Press, Jan. 9–11 1999.

[11] F. Dahlgren, M. Dubois, and P. Stenström. Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors. In *1993 International Conference on Parallel Processing, August 1993*, volume 1, pages 56–63, Aug. 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, Sept 3-6, 1996, Mumbai (Bombay), India*, pages 330–341. Morgan Kaufmann, 1996.

[13] What is exponential smoothing? http://cne.gmu.edu/modules/dau/stat/expsmoothg/def_frm.html.

[14] B. D. Davison and H. Hirsh. Predicting sequences of user actions. In *Proceedings of AAAI-98/ICML-98 Workshop*, pages 5–12, Madison, WI, July 1998. AAAI Press.

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

[16] P. R. Doshi, E. A. Rundensteiner, and M. O. Ward. Prefetching for visual data exploration. In *Database Systems for Advanced Applications (DASFAA)*, 2002.

[17] P. R. Doshi, E. A. Rundensteiner, M. O. Ward, and I. D. Stroe. Prefetching for visual data exploration. Technical Report TR-02-07, Worcester Polytechnic Institute, Computer Science Department, 2002.

[18] S. Feiner and C. Beshers. Worlds within worlds: Metaphors for exploring n-dimensional virtual worlds. *Proc. UIST'90, p. 76-83*, 1990.

[19] D. Florescu, A. Levy, D. Suciu, and K. Yagoub. Run-time management of data intensive web-sites. Technical report, Inria, Institut National de Recherche en Informatique et en Automatique, 1999.

[20] Y. Fua, M. Ward, and E. Rundensteiner. Structure-based brushes: A mechanism for navigating hierarchically organized data and information spaces. *IEEE Visualization and Computer Graphics, Vol. 6, No. 2, p. 150-159*, 2000.

[21] 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.

[22] 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.

[23] J. Han and M. Kamber. *Data Mining Concepts and Techniques*, chapter Chapter 9: Mining Text Databases, page 429. Morgan Kaufmann, 2001.

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

[25] S. Hibino and E. Rundersteiner. User interface evaluation of a direct manipulation temporal visual query language. In *Proc of The Fifth ACM Intl Multimedia Conf (MULTIMEDIA '97)*, pages 99–108, New York/Reading, Nov. 1998. ACM Press/Addison-Wesley.

[26] Y. Ioannidis. Dynamic information visualization. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(4):16–16, Dec. 1996.

[27] N. Jing, Y. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *IEEE Transaction on Data and Knowledge Engineering*, 10(3):409–432, May 1998.

[28] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *Proc of the 24th Annual Intl Symposium on Computer Architecture (ISCA-97)*, Computer Architecture News, pages 252–263, New York, June 1997. ACM Press.

[29] S. Kaushik and E. Rundensteiner. SVIQUEL: A spatial visual query and exploration language. *DEXA*, 1460:290–299, 1998.

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

[31] T. Kohonen. The self-organizing map. *Proc. of IEEE, p. 1464-80*, 1978.

[32] M. Livny, R. Ramakrishnan, K. S. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and R. K. Wenger. DEVise: Integrated querying and visualization of large datasets. In *Proc ACM SIGMOD Intl Conf on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 301–312. ACM Press, 1997.

[33] S. Manoharan and C. R. Yavasani. Experiments with sequential prefetching. *Lecture Notes in Computer Science*, 2110:322–331, 2001.

[34] A. Martin and M. Ward. High dimensional brushing for interactive exploration of multivariate data. *Proc. of Visualization '95, p. 271-8*, 1995.

[35] A. Mead. Review of the development of multidimensional scaling methods. *The Statistician, Vol. 33, p. 27-35*, 1992.

[36] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1999.

[37] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[38] X. Qian. Query folding. In S. Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 48–55. IEEE Computer Society, 1996.

[39] 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, Montreal, Quebec, Canada, June 4-6, 1996*, pages 24–34. ACM Press, 1996.

[40] 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.

[41] 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.

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

[43] 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.

[44] N. Swaminathan and S. Raghavan. Intelligent prefetching in www using client behavior characterization. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 13–19, Sept. 2000.

[45] E. Tanin, R. Beigel, and B. Shneiderman. Incremental data structures and algorithms for dynamic query interfaces. *ACM Special Interest Group on Management of Data*, 25(4), Dec. 1996.

[46] M. K. Tcheun, H. Yoon, and S. R. Maeng. An adaptive sequential prefetching scheme in shared-memory multiprocessors. In *Proceedings of IEEE ICPP-97*, Bloomingdale, IL, Aug. 1997. IEEE Press.

[47] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource mangement. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 1–11, California, Nov. 1994.

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

[49] M. O. Ward, J. Yang, and E. A. Rundensteiner. Hierarchical exploration of large multivariate data sets. *Proceedings Dagstuhl '00: Scientific Visualization*, May 2001.

[50] S. Weiss and C. Kulikowski. *Computer Systems That Learn. Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems*. Morgan Kaufmann Publishers, San Mateo, CA., 1991.

[51] G. Wills. Selection:524,288 ways to say this is interesting. *Proc. of Information Visualization '96, p. 54-9*, 1996.

[52] A. Woodruff and M. Stonebraker. Visual information density adjuster (VIDA). Technical Report CSD-97-968, University of California, Berkeley, Nov. 24, 1997.

[53] Xmdvtool home page. http://davis.wpi.edu/˜xmdv.

[54] J. Yang, M. O. Ward, and E. A. Rundensteiner. Interactive hierarchical displays: A general framework for visualization and exploration of large multivariate data sets. *Computers and Graphics journal*, 2002.

[55] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: an efficient data clustering method for very large databases. *SIGMOD Record, vol.25(2), p. 103-14*, pages 103–114, June 1996.