WPI-CS-TR-00-06 Mar. 2000

Scalable Visual Hierarchy Exploration

by

Ionel D. Stroe

Elke A. Rundensteiner

Matthew O. Ward

Computer Science

Technical Report

Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department

100 Institute Road, Worcester, Massachusetts 01609-2280

Scalable Visual Hierarchy Exploration *

Ionel D. Stroe, Elke A. Rundensteiner and Matthew O. Ward

Computer Science Department

Worcester Polytechnic Institute

Worcester, MA 01609

daniel | rundenst | matt@cs.wpi.edu

Abstract

More and more modern computer applications, from business decision support to scientific data analysis, utilize visualization techniques to support exploratory activities. Various tools have been proposed in the past decade to help users better interpret data using such display techniques. However, most do not scale well with regard to the size of the dataset upon which they operate. In particular, the level of cluttering on the screen is typically unacceptable and the performance is poor. To solve the problem of cluttering at the interface level, visualization tools have recently been extended to support hierarchical views of the data, with support for focusing and drilling-down using interactive brushes. To solve the scalability problem, we now investigate how best to couple such a visualization tool with a database management system without losing the real-time characteristics. This integration must be done carefully, since visual user interactions implemented as main memory operations do not map directly into efficient database operations. The main efficiency issue when doing this integration is to avoid the recursive processing required for hierarchical data retrieval. For this problem, we have develop a tree labeling method, called MinMax tree, that allows the movement of the on-line recursive processing into an off-line precomputation step. Thus, at run time, the recursive processing operations translate into linear cost range queries. Secondly, we employ a main memory access strategy to support incremental loading of data into the main memory. The techniques have been incorporated into XmdvTool, a multidimensional visual exploration tool, in order to achieve scalability. The tool now successfully scales up to datasets of the order 10^5–10^7 records. Lastly,

*This work is supported under NSF grant IIS-9732897 and NSF CISE Instrumentation grant IRIS 97-29878.

we report experimental results that illustrate the impact of the proposed techniques on the system's overall performance.

Keywords: Hierarchical Structures, Visual Exploration, Recursive Queries, Memory Management, Database Backend

1 Introduction

Visualization is an important process in data mining. 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 the analysis. Humans can detect patterns and trends in the underlying data by just looking at it, *without* being aware in advance about what data model they'll face.

Human perception is of course greatly influenced by the way data is presented. Thus, various techniques for displaying data have been proposed over the years, each of which better emphasizes some of the data characteristics versus others [1, 18, 4, 8, 22]. However, most of these techniques do not scale well with respect to the size of the data. As a generalization, [10] 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.

A new approach has been proposed recently for displaying large datasets [9]. The idea here is to present data at different levels of detail based on applying an aggregation function to a hierarchical structure that results from a proximity clustering process. The problem of cluttering at the interface level is solved by displaying only a limited set of aggregates at a time. However, such hierarchical summarizations result in increasing the size of the input by at least one order of magnitude, making the management of data become an issue.

Storing and retrieving the data sets efficiently has traditionally 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 sets, it becomes unacceptable when scaling to large sets. Visualization applications must thus be integrated with database management systems to enable such a capability. The last couple of decades of research in the area of databases can greatly contribute to increasing the performance of a data intensive application such as exploratory visualization.

Coupling a database with the visualization tool cannot be performed *blindly* though. Techniques used for main memory processing are typically not efficient any more if implemented directly in a database environment. A trivial and well known example is sorting. Internal sorting strategies differ significantly from external sorting ones. Another example is presented in this work. The recursive processing involved when navigating through hierarchies in main memory is no longer appropriate when storing those hierarchies on the disk. Instead, we propose using a technique called *MinMax trees* [28] for this purpose. MinMax labels the hierarchies and transformed the recursive processing into a set of fast range queries.

In general, there are two questions to be answered when doing such a visualization–database integration. The first one is how to most effectively translate the visual exploration operations such as zooming, brushing, etc., into a database understandable language such as SQL. The second one is how to manage storing the results of the database requests into the main memory such that subsequent memory access operations be as efficient as possible. This paper gives solutions for both questions. Furthermore, we applied our proposed solution strategies when coupling a visualization tool called XmdvTool [32] with an Oracle database. We also have evaluated the performance of the system through practical experiments. Although our work has been driven by practical needs, we also sketch a more general class of systems to which the proposed techniques are applicable. These characteristics are formalized in Section 2.3.

The paper is organized as follows. Section 2 presents the hierarchical brushing and navigation as they are implemented in XmdvTool. It also gives an abstraction of the properties and requirements of the system, defining the area of applicability of our method. Section 3 addresses the problem of translating visualization requests into SQL queries by using a technique called MinMax trees. Section 4 presents the memory management as implemented in the system. Experimental results are reported in Section 5. Section 6 introduces the related work. Finally, we present our conclusions as well as directions for further research in Section 7.

2 Hierarchical Multivariate Data Visualization

The work presented in this paper was initially triggered by our goal of adding database support to *XmdvTool* [32]. 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) with interactive selections and linked views. Our recent efforts have produced hierarchical parallel coordinates, that allows multi-resolution data presentation. The main idea is to cluster the datapoints based on a 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 an environment abstraction, 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 [33] 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. He also created a selection calculus that enumerates all possible combinations of actions between a previous selection and a new selection (replace, add, subtract, intersect, and toggle) and attempted to identify configurations of these actions that would be most useful.

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 [2] and applied to high dimensional scatterplots. Ward and Martin [31, 21] extended brushing to permit brushes to have the same dimensionality as the data (N-D instead of 2-D). They also explored the concepts of multiple brushes, composite brushes (formed by logical combinations of brushes), and *fuzzy* brushes, that allow points to be partially contained within a brush. Haslett et al. [11] introduced the ability to show the average value of the points that are currently selected by the brush.

One common method of classifying brushing techniques is to identify in which space the selection is being performed, namely screen or data space. This can then be used to specify a *containment criterion* (whether a particular point is inside or outside the brush). In *screen space* techniques, a brush is completely specified by a 2-D contiguous subspace on the screen. In *data space* techniques, a complete specification consists of either an enumeration of the data elements contained within the brush or the N-D boundaries of a hyper-box that encapsulates the selection.

A third category, namely *structure space* techniques, that allows selection based on structural relationships between data points has been introduced in [10]. The *structure* of a data set specifies relationships between data points. This structure may be explicit (e.g., categorical groupings or time-based orderings) or implicit (e.g., resulting from analytic clustering or partitioning algorithms). Examples of structures include linear orderings, tree hierarchies, and directed acyclic graphs (arbitrary hierarchies). In this work we focus on tree hierarchies. 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. Intuitively, if looking at a tree structure from the point-of-view of its root node (Fig. 1), the extent subrange appears as a *focus region* (with the focus point at its center), while the level-of-detail subrange corresponds to a sampling rate factor or a *density*. In a 2-D representation of the tree (Fig. 2), the subranges correspond to a horizontal and vertical selection, respectively.

2.2 Creation and Manipulation of a Structure-Based Brush

Figure 3 shows the structure-based brushing interface implemented in XmdvTool [10]. The triangular frame depicts the hierarchical tree. The contour near the bottom of the frame delineates the

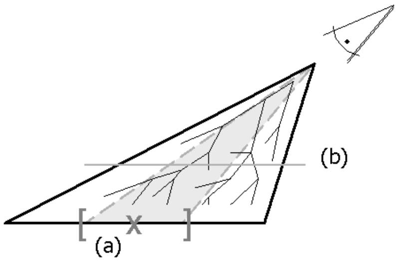

Figure 1: Structure-based brush as combination of a *focus region* (a) and a *density factor* (b).

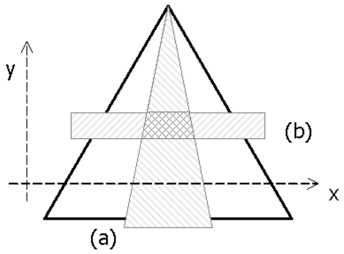

Figure 2: Structure-based brush as combination of a horizontal (a) and a vertical (b) selection.

approximate shape formed by chaining the leaf nodes. The colored bold contour across the tree delineates the tree cut that represents the cluster partition corresponding to the specified level-of-detail. XmdvTool uses a *proximity-based coloring scheme* in assigning colors to the partition nodes [9]. In this scheme, a linear order is imposed on the data clusters gathered for display at a given level-of-detail. This linear order is directly derived from the order in which the tree is traversed when gathering the relevant nodes for a given level-of-detail. Colors are then assigned to each cluster by looking up a linear colormap table. The same colors are used for the display of the nodes in the corresponding data display. The two movable handles on the base of the triangle, together with the apex of the triangle, form a wedge in the hierarchical space.

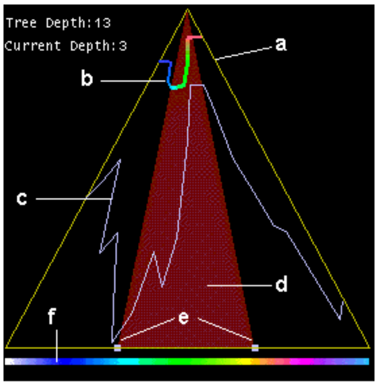

Figure 3: Structure-based brushing interface in XmdvTool. (a) Hierarchical tree frame; (b) Contour corresponding to current level-of-detail; (c) Leaf contour approximates shape of hierarchical tree; (d) Structure-based brush; (e) Interactive brush handles; (f) Color map legend for level-of-detail contour.

Since, as shown before, a structure-based brush is defined as the intersection of two independent selections, it necessarily follows that setting such a brush requires two computational phases as well.

The first one, the horizontal selection, is accomplished in two steps. In the first step a set of leaf nodes is initially selected based on the order property using the two handles indicates by e in Fig. 3. Basically, this step corresponds to "select all leaves between the two extreme values e_1 and e_2". In the second phase, the initial selection is propagated up towards the root based on what we termed an ALL semantic: "select nodes that have *all* their children already selected" (other semantics like ANY or $MOST$ are also possible [28]). The second computation phase, the vertical selection, consists of refining the set of nodes generated in phase one. Basically, the nodes on the desired level-of-detail are only retrieved out of the whole phase one selection.

The brush operations, as described above, are inherently recursive. Recursive processing in relational database systems can be extremely time consuming and thus unsuitable for interactive applications. In Section 3 we develop equivalent but non-recursive computation methods for setting structure-based brushes based on assigning some precomputed values to the nodes that recast retrievals as range queries.

2.3 Model Abstraction

We now introduce a formal model characterizing the salient features of the application domain for the proposed techniques, thus establishing the applicability and scope of our solution. The input space is composed of entries in the extent (x) dimension and in the level (y) dimension. Since the two dimensions are independent, the space is actually a Cartesian product of entries. We can envision this space by overlapping a 2-D grid over the tree hierarchy (Fig. 4). In this representation, a selection is a sequence of consecutive regions with the same level value (Fig. 5). The data points are *spatial* objects whose distribution may be unknown and which can be retrieved by a so called *query mechanism*. Thus, there must be a containment criterion that specifies for each object whether it is included in the current selection window or not. In order for us to be able to implement a structure-based brush as previously specified, an order is enforced on the set of leaf nodes and in general on the nodes with the same level-of-detail value.

An example of transforming a tree structure into a *navigation space* (i.e., the set of spatial objects) is given below. The tree (Fig. 6) is first represented as a tree diagram [28] and then overlapped on a 2-D grid of integers.

The set of characteristics below identifies the requirements that a system needs in order for our

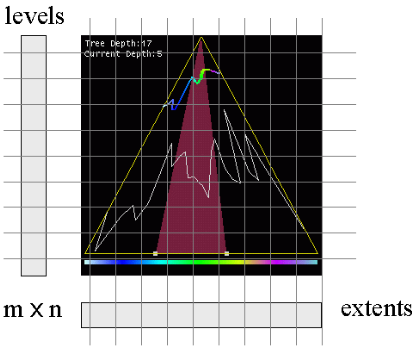

Figure 4: Selection space abstraction.

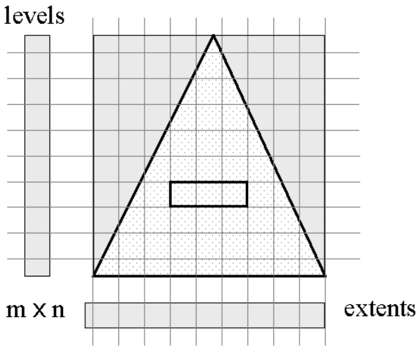

Figure 5: Active window in the selection space.

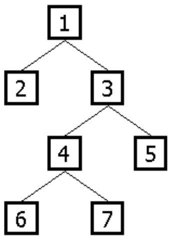

Figure 6: A tree example.

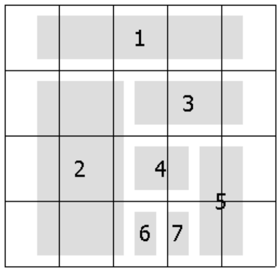

Figure 7: Navigation on a tree support set.

approach to apply (besides providing the naming conventions):

1. *Navigation* consists of continuously changing a selection window (called the *active window*) defined over an $n \times m$ grid of integers (called the *navigation grid*).

2. On both axes of the *navigation grid*, the intervals are indexed rather than the points. Thus, on the x axis (called the extent axis) the intervals are denoted as e_i, while on the y axis (called the level axis) the intervals are denoted as L_k. The grid is therefore a set of rectangle regions of the form (e_i, L_k) (Fig. 8).

   An *active window* is a "compact" selection of points $\{(e_i, L_k), (e_{i+1}, L_k), ..., (e_j, L_k)\}$ on the same level (Fig. 9). Thus, an active set is specified by a triplet of the form (e_i, e_j, L_k).

3. Each *active window* uniquely identifies a set of spatial objects (called the *active set*) being

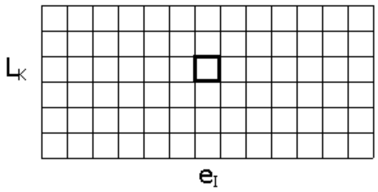

Figure 8: Navigation grid.

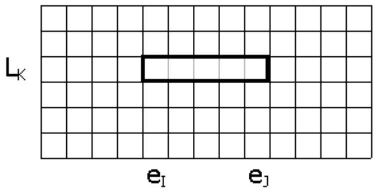

Figure 9: Active window.

selected among the objects of a given set (called the *base set* (Fig. 10)).

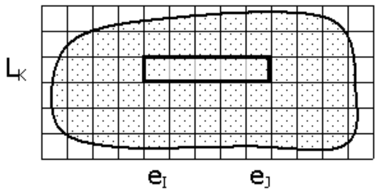

Figure 10: Base set.

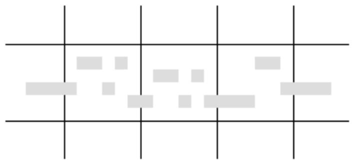

Figure 11: Objects on the same level are totally ordered.

The following properties about the base set are assumed:

- There is a partial order relationship defined on the objects of the base set. However, there is a total order relationship among objects on the same level (Fig. 11).
- Any active set can contain 0, 1 or multiple objects. This number, however, is not known in advance and can change over time.
- An object can spread over multiple regions on the same level and can belong to multiple levels. Thus, the active windows are not additive, i.e., the union of two active sets corresponding to two windows $W1$ and $W2$ is not necessarily the same as (although included in) the active set corresponding to $W1 \cup W2$. Moreover, the active sets for two disjoint active windows are not necessarily disjoint.

4. The *active window* may only change incrementally, i.e., only one of the three parameters can change at a time and only by a single unit. This is an essential property exploited by our memory management strategy, as shown in Section 4.

3 MinMax Trees: Translating the Navigation Operations

The question addressed in this section is "how do we translate the visualization operations into database operations". For this purpose we have developed a technique called *MinMax tree* [28]. The method places the recursive processing in a *precomputation* stage, when labels are assigned to all nodes. The labels provide a containment criterion. Thereafter, simply by looking at the parameters of an active window and at a node's label, we are able to determine whether that node belongs to the active selection or not.

3.1 MinMax Tree Definition

A MinMax tree is a n-ary tree in which nodes correspond to open intervals defined over a totally ordered set, called an *initial set*. The leaf nodes in such a tree form a sequence of non-overlapping intervals. The interior nodes are unions of intervals corresponding to their children.

The initial set can be continuous (such as an interval of real numbers) or discrete (such as a sequence of integers). In either case, the nodes are labeled as pairs of values: the extents of their interval. As the intervals are unions of child intervals, it follows that a node will be labeled with the minimum extent of its first interval and the maximum extent of its last interval. A node n having for example two children $c_1 = (\alpha, \beta)$ and $c_2 = (\gamma, \delta)$ will be labeled as $n = (\alpha, \delta)$. Hence, such a tree is called MinMax. Examples of MinMax trees are depicted for a continuous initial set in Fig. 12 and for a discrete initial set in Fig. 13.

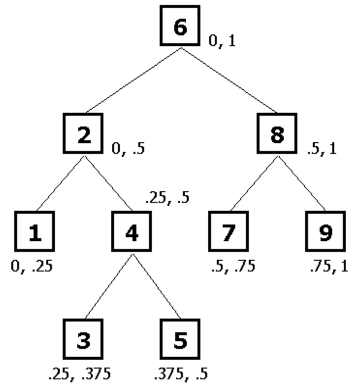

Figure 12: A continuous MinMax tree.

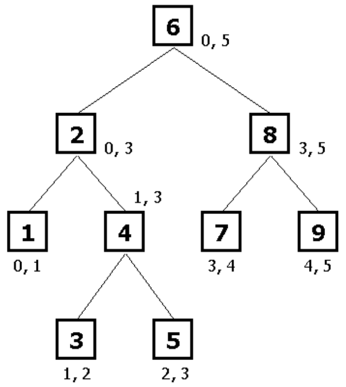

Figure 13: A discrete MinMax tree.

Essentially, the process of labeling the nodes is recursive. The intervals are computed and assigned off-line at the time the hierarchy is created and their value and distribution (as well as the tree structure itself) depend on the technique used to create the hierarchy.

An important property of a MinMax tree is that given a MinMax tree T and two nodes x and y of T whose extent values are $(x_1,\, x_2)$ and $(y_1,\, y_2)$ respectively, node x is an ancestor of node y if and only if $x_1 \leq y_1$ and $y_2 \leq x_2$. The property is based on the intuition that each node in the tree is included in its parent as an interval.

3.2 Using MinMax Trees to Set Structure-Based Brushes

In this work we only consider the case of a static tree hierarchy, i.e., no updates are assumed to occur during navigation, assumption imposed due to the visual tool. However, elsewhere we extended the MinMax technique to also apply for dynamic and arbitrary (acyclic graph) hierarchies [28].

First, we notice that any tree can be labeled as a MinMax tree if, for example, we start with an arbitrary continuous initial interval as the root label and recursively divide it into equal sub-intervals, each sub-interval being assigned to a child (see, for example, the binary tree in Fig. 12). Having the hierarchy labeled as a MinMax tree, we can implement an *ALL* structure-based brush (disregard the level value now) as a non-recursive operation based on the property that given the brush values v_{min} and v_{max}, the brush represents the union of all nodes $n = (n_1, n_2)$ whose extents are fully contained in the brush interval (v_{min}, v_{max}), i.e., $(n_1, n_2) \subseteq (v_{min}, v_{max})$ [28]. The horizontal selection defined by an *ALL* structure-based brush for the example in Fig. 12 and the brush values 0.25 and 0.75 is visually depicted in Fig. 14 and Fig. 15. The selected nodes are covered by a shaded area in the first one and underlined in the latter.

The value corresponding to the level-of-detail of an aggregation (node) can also be precomputed and stored in the node as part of the label. A vertical selection corresponds further to identifying all nodes at a desired level-of-detail that satisfy the extent selection criterion.

3.3 Translating MinMax Tree Brushes into SQL

The MinMax tree technique reduced the containment criterion from an initially recursive semantic to a simple inclusion test (horizontal selection) and an equality test (vertical selection). By using the method, we can consequently decide whether a node should belong to the selection or not

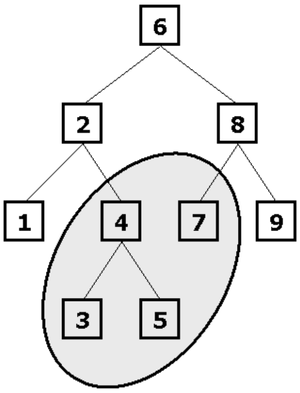

Figure 14: An ALL structure-based brush horizontal selection with the extent values set to 0.25 and 0.75.

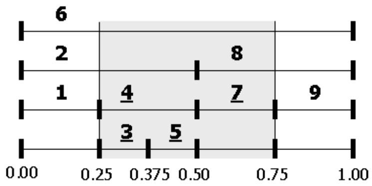

Figure 15: Same ALL structure-based horizontal selection in a tree diagram representation.

independently from the information stored in the other nodes. One scan of the data is hence sufficient to completely form the selection.

Let the relational table that stores the datapoints be H. Besides the node information, H contains also extent and level attributes:

$$\text{H (e\_min, e\_max, level, ...)}$$

An *ALL* structure-based brush for a hierarchy labeled as a MinMax tree and the brush values (min, max, lev) is now a simple range query, expressed in SQL2 as:

```
select  *
from    H
where   e_min >= :min
and     e_max <= :max
and     level  = :lev;
```

The complexity of the query to be answered on-line when setting a structure-based brush reduces considerably. While a recursive processing would require an exponential processing time, a range query requires only a linear (logarithmic if using indices) processing time. Tests to confirm the important improvement achieved by our method are presented in Section 5.

4 Memory Management

The question addressed in this section is "how do we organize the data into memory once it arrives from the database?". The 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 and the current content of the buffer has to be computed fast. Thus, we need to be able to know in each moment what data resides in the memory without fully traversing the buffer.

A significant difference in the buffer management is made by whether the buffer is large enough to store all the objects in the active set or not. We refer to these two cases as database intensive (DBI) and database semi-intensive (DBSI) and treat them separately in what follows.

4.1 DBSI Case

A fixed length buffer B is assumed. We refer to the number of objects that can be stored in B as n. In the DBSI case, n is assumed to be greater than the number of objects in *any* active set.

A characteristic of the objects that are placed in the buffer 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 the object $ID = x$ or $ID = y$, rather instead it passes a query q_{new} to the back-end: "are the objects with these characteristics (within this brush) available?". Thus, although there are object attributes (the extents) that uniquely identify each entry, a classical lookup for a cache key is not possible when testing whether an object is in the buffer or not. Instead, a query $q_{current}$ is associated with the buffer, similar to in *semantic caching* [6]. The query q_{new} is then compared with $q_{current}$ to determine what objects from q_{new} are not in $q_{current}$, and those objects are retrieved next. This difference results in fact in a new query (q_+) that corresponds to those *to be selected next* objects.

Special attention has to be paid in a semantic cache environment to not allow duplicates in the buffer. Thus, when more than one $q_{current}$ query is stored in the buffer, they are *forced* to be disjunct. This means that a new q_{new} query will modify the semantic of the existing $q_{current}$ queries such that they do not refer to any common objects any more. In our model, only one query is allowed at a time. However, when a set of objects is deselected and then selected again, duplicates may occur if the old objects have not been completely removed from the buffer as soon as they were not part of the active set. Thus, when q_{new} contains fewer elements than $q_{current}$, a query q_- will determine what elements need to be removed immediately from the cache.

Consequently, the operations that the buffer needs to support are adding and removing objects that correspond to a given MinMax query. An observation that needs to be made here is that the difference between $q_{current}$ and q_{new} can always be computed due to the *incremental assumption* (Section 2). Moreover, both new possible states $(q_{current} + q_+)$ and $(q_{current} - q_-)$ correspond to an active window and thus to a compact set of objects. In order to make the object additions and subtractions efficient, we store the objects in the buffer 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 3) be retrieved in order by adding an *ORDERED BY* clause to or MinMax-derived SQL queries. This SQL clause will not require extra processing time if we store the objects in the database ordered by their extent values (left, for example). It would require only minimal extra processing if we store the objects unordered but have an index built.

The buffer is implemented as a circular list. In the list, one agent can place objects (the back-end) and one can read them (the front-end). The model is similar to the consumer-producer problem with the only difference being the *consumer* does not actually remove objects (Fig. 16) but only reads them. The synchronization model is however the same: the *producer* cannot place objects in a full buffer (condition ensured by the DBSI assumption) and the *consumer* cannot read elements that are not yet *produced* (i.e., not yet in the buffer). The agents are implemented as separate threads. Thus, there is one pointer i_p where the *producer* places objects and one pointer i_c where the *consumer* consumes objects. In addition, there is a pointer i_0 that points to the first object in the active set. This may not be the first entry in the list. The synchronization conditions are therefore that i_c cannot cross over i_p, and i_p cannot cross over i_0. Since the producer process is faster than the consumer one, i_c is usually one step behind i_p. To differentiate between the cases when the producer finished loading objects versus when it is waiting for an object to come, a new symbol (#) is placed in the list by the producer, when the whole active set was retrieved (Fig. 17).

Given the navigation model defined in Section 2 there are three types of buffer access operations. (1) When the user changes the current level all objects become *old* and are removed from the buffer. The buffer is then flushed with a *setmem()* type of operation. Then, the new active set is loaded

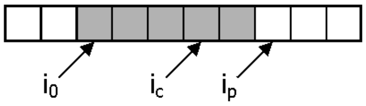

Figure 16: The buffer access model. i_p is the producer pointer; i_c is the consumer pointer.

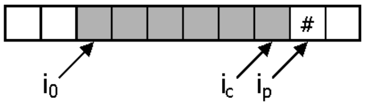

Figure 17: The state of the buffer when the producer thread finishes *producing* elements.

in the memory in the increasing order of the extent value. (2) When a brush is restricted, from let's say (e_1, e_2, L) to (e_1^*, e_2, L), $e_1 \leq e_1^*$, the elements starting with position i_0 are sequentially removed from the buffer until the first element in the new active set, (e_1^*, e_2'), is reached. i_0 is set to the address of (e_1^*, e_2'), i_c is set to i_0 and the consumer process starts (the buffer is in a *consistent* state). If the new brush is (e_1, e_2^*, L), $e_2^* \leq e_2$, then i_p is set to the address of (e_1', e_2^*) and i_c is set to i_0. (3) A similar process occurs when a brush is incremented. Either i_0 or i_p expand, and the space is filled in with new objects. The *ordering assumption* is used here to preserve a consistent state in the buffer. The content of the buffer is always maintained as a triplet (e_1, e_2, L). However, if $(e_1^0, e_2^0)...(e_1^m, e_2^m)$ are the objects in the buffer, then necessarily $e_1 = e_1^0$ and $e_2 = e_2^m$.

4.2 DBI Case

In the DBI case, the number of objects in one active set at a time may be greater than the number n of entries available in the buffer. In this case, as the producer process advances in the circular list, it may attempt to overwrite some of the old objects (Fig. 18). This is acceptable if the objects to be overwritten have already been read by the front-end. However, to assure this functionality our model must slightly change under the DBI assumption. Intuitively, if for instance the data in the buffer needs to be read a second time (as it is the case in a *refresh* process) then the consumer changes the traversal direction. The producer is positioned at the first non-overwritten entry and also changes its traversal direction (Fig. 19). In this *two-way traversal* procedure we keep two pointers i_0 and i_1 positioned at the first and last element, respectively, in the current active set (if those elements exist in the buffer). These pointers do not however take part in the synchronization process any more. The synchronization simply requires that the consumer and the producer do not cross one another. Since the traversal direction is irrelevant, we implement this requirement by not allowing the two pointers to take on the same value, no matter which one follows the other.

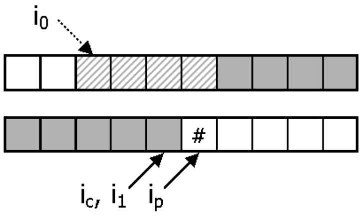

Figure 18: Finishing reading. Current objects are painted solid; the overwritten objects stripped. Dash lines are undefined pointers.

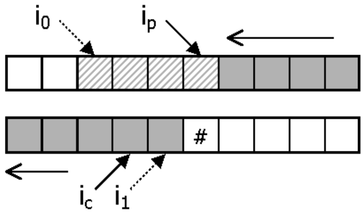

Figure 19: Starting re-reading. The traversal direction changes for both consumer and producer. Buffer in an inconsistent state.

A *consistent* state is one in which the buffer contains a sequence of objects that either starts with the first element of the active set (refered to as α-state) or ends with the last one (refered to as an β-state), i.e., i_0 and i_1 are not both undefined (NULL) at the same time. An α-state usually occurs after a backward traversal, while an β-state occurs after a forward traversal. There are six types of access operations in the DBI case. We will show that all these operations leave the buffer in a consistent state.

1. Refreshing (re-reading the buffer). (a) A forward traversal starts. i_c is set to i_0, i_p to the end of the sequence (always one step before i_0). i_1 is undefined and i_0 is also set to NULL. The loading process continues with the producer behind the consumer, until the end of the active set is reached and a termination character is placed. i_1 is then redefined. The whole process ends when the reader reaches the termination character. The buffer is left in a β-state. (b) A backward traversal symmetric to (1a) begins.

2. Changing the level. Changing the level will flush the current buffer no matter the state and load the new active set in a forward direction. A β-state is reached.

3. Expanding the brush from left. (a) The process corresponds to a continuation of the loading process as if it were temporary interrupted. The termination character is removed and the producer continues to place objects in the buffer. The reader starts reading elements from its previous position. The process ends again in an α-state. A re-read operation of type (1a)

usually follows. (b) Same as (1b).

4. Expanding the brush from right. (a) Same as (1a). Similar with (3a) (symmetric).

5. Restricting the brush from left. (a) The elements that no longer belong to the active set are removed from the buffer starting from i_0. When the first element of the new active set is reached, an (1a) re-read process begins. (b) Same as (1b).

6. Restricting the brush from right. (a) Same as (1a). Similar with (5a) (symmetric).

In conclusion, except for the two cases (3a) and (4b), when a re-read is usually needed after a sequence of writings, the reading and writing time typically overlap (the operations can be completely parallelized).

5 Experimental Results

5.1 System Architecture

The system architecture depicted in Fig. 20 illustrates the key modules and interactions of our proposed solution approach. First, as shown on the top of the figure, an off-line process transforms the hierarchical data into MinMax trees in a labeling process. The prepared data is then loaded into the database. The process implements the MinMax tree approach explained in Section 3. Information about the database schema is used later during exploration.

When exploring, users interact via the graphical interface (GUI) shown of the right side. Details of the visual exploration interface have been given in Section 2. The visual navigation operations correspond to changes in the active set. When a change in the active set is detected, a *producer* thread is created, while the GUI itself acts as *consumer*. Basically, two threads operate now concurrently on the buffer data: the consumer and the producer. The active set information from the GUI is passed to a *rewriter*. The rewriter consults the semantic of the buffer, expressed by a buffer query, and then generates a set of sub-requests to incrementally adjust the data in the buffer. Each sub-request is transformed into an SQL query by the *translator*, based on the database schema. The queries are passed to the *loader*, which fetches the necessary objects from database and places them in the buffer. From here, the *reader* reads them and sends them to the display.

The buffer management as well as the synchronization between the producer and the consumer are described in Section 4.

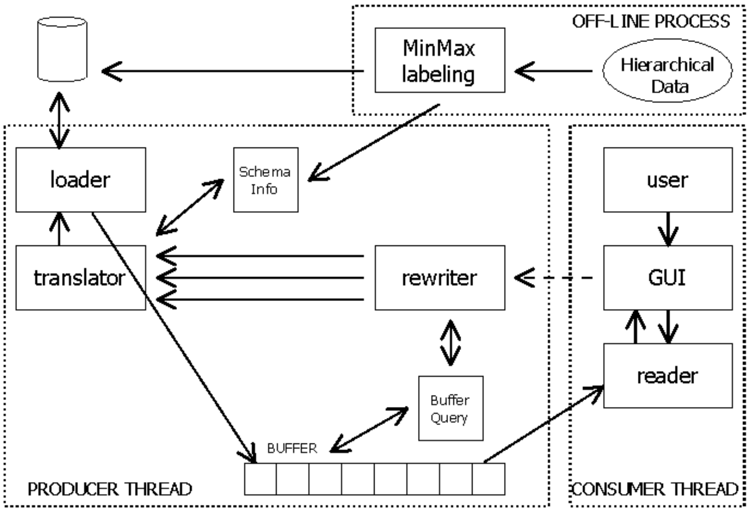

Figure 20: System architecture. The oval depicts the initial data. Dash-lines rectangles represent threads and processes. Solid-line rectangles are agents. Squares represent meta-knowledge. Solid arrows show the control flow. Dash arrows show the creation of child threads.

5.2 Settings

We ran a set of experiments to measure the overall performance of our system on standard commercial technology (Oracle 8i). All the experiments were conducted on an Alpha v4.0 878 DEC station, running Oracle 8.1.5. and having no concurrent clients during the tests. We used C as the host language and embedded SQL statements for accessing the data in the database.

The first experiments were designed to assess the significance of the processing time improvement achieved by our MinMax method versus the direct SQL query translation (recursive approach). Then, we studied the behaviour of the system when scaling the size of the input. Since the size of the input depends on the brush values (extents and level of detail) as well as on the dataset size, we varied all these parameters. We always measured the response time (processing time), since our method was designed to be applied in interactive systems. All the experiments were run six times, the response time shown here being the average of these six runs.

For the experiments we used various datasets, both real and synthetic, with always consistent results. The measurements reported in this paper were obtained using synthetic data. We made this choice to avoid the problems induced by non-homogeneous structures in interpreting the result, since the performance depends to some degree on the shape of the tree, i.e, the distribution of the nodes in the clustering tree. Thus we used complete trees with a constant fan-out (2). The three datasets had D1=2^{12} (4,096), D2=2^{16} (65,536), and D3=2^{17} (131,072) tuples in the hierarchical table. The number of dimensions for the data points was 8. The extent values were set to (0, .2), (0, .4), (0, .6), (0, .8) and (0, 1), while the level-of-detail values were based on the actual level in the tree and varied in two-level increments $(0, 2, 4, ...)$.

5.3 Experiments

Experiment 1: MinMax vs. Recursive. First, we measured the system's performance when implementing the structure-based brushes using both the MinMax tree technique and a recursive technique. As SQL does not support recursive views, the recursive technique used an additional attribute to mark the selected tuples along the recursion steps. In both cases we created simple indices on the join attributes and compound indices on extent and level attributes. Maintaining the indices was not an issue because the trees were static during our experiments. The result of the experiment is presented in Figures 21 – 23. As expected, the MinMax method is substantially faster than the recursive one in all cases. These results show clearly that the application of the MinMax technique in our system made visual exploration over large datasets practically feasible, thus accomplishing our ultimate goal.

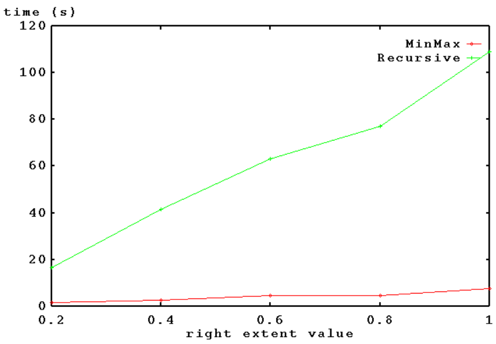

Figure 21: MinMax vs. Recursive. Structure-based brushes for dataset D1.

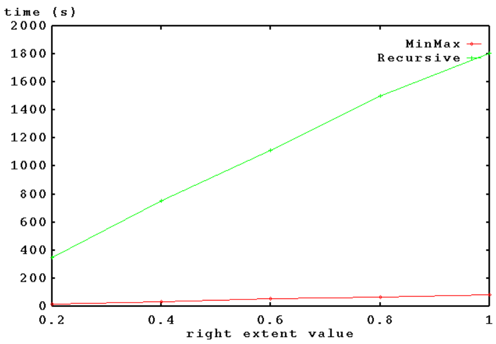

Figure 22: MinMax vs. Recursive. Structure-based brushes for dataset D2.

Experiment 2: Varying brush parameters. In the next three experiments we analyze the behaviour of the system when the size of the brushes changes. Specifically we vary the level value, the extent values and the size of the dataset. For these experiments we use a compound index on $(e1, e2, L)$ and a $NOCACHE$ hint for the Oracle's optimizer (to keep the impact of the Oracle's caching small). Since the brushes are implemented as range queries, we expect that the response time will vary linearly with respect to all these parameters.

Experiment 2a: Varying the level value. In this experiment, the extent values are constant $(0.25, 0.75)$ and the level value varies in two-unit increments. As in a complete binary tree the number of points doubles each level, we expected that the response time would have a similar behaviour and thus the method would be linear in the size of the input. However, as one can see in Fig. 24, the linear behaviour can be observed only for large levels, while for small ones being almost constant.

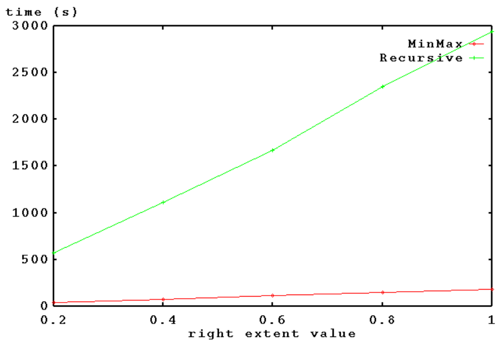

Figure 23: MinMax vs. Recursive. Structure-based brushes for dataset D3.

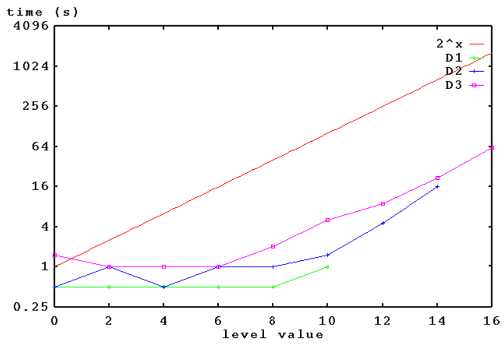

Figure 24: Varying the level value. Functions compared against 2^x. Logarithmic y scale.

Experiment 2b: Varying the extent values. In this experiment, the level value is constant (11 – the leaf level in the smallest dataset) and the extent values varies from $(0.0, 0.0)$ to $(0.0, 1.0)$ in 0.2 increments (captured on the x-axis). The behaviour is again expected to be linear. This experiment is particularly important since it validates our assumption that the incremental computing of brushes is needed. Indeed, as shown in Fig. 25, the increase in the response time is linear, and thus the processing time for a level-based brush is proportionally reduced when computing it incrementally.

Experiment 2c: Varying the size of the dataset. In this experiment we used a new

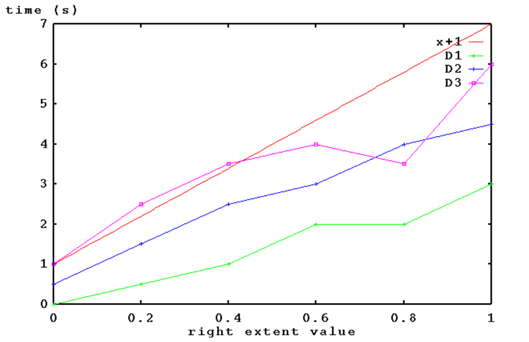

Figure 25: Varying the extent value. Functions compared against $x+1$.

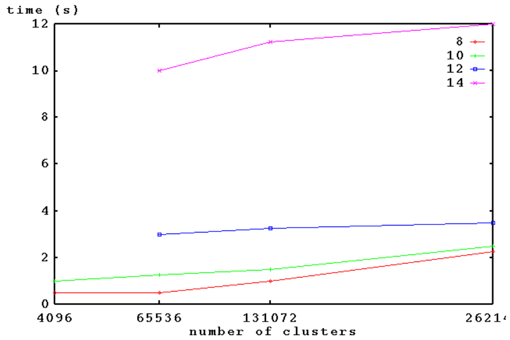

Figure 26: Varying the dataset size. Levels 12 and 14 not defined for D1.

dataset, D4, having $2^{18} = 262144$ tuples. The brush settings are set to values from $(0.25, 0.75; 8)$ to $(0.25, 0.75; 14)$, and the dataset changes from D1 (4096 tuples) to D4 (262144 tuples). The number of nodes doubled from D1 to D2, from D2 to D3, and from D3 to D4 (shown on the x-axis). However, as one could see in Fig. 26, the resulting time is almost constant. This ensures that our system can scale well to large datasets. (Functions labeled “12” and “14” are not defined in 4096, since D1 only has 11 levels.)

Discussion. The experiments have demonstrated that the processing time for a structure-based brush is only proportional to the number of objects being selected and independent from what brush parameter changed. Thus, the time required for computing a sequence of brushes is proportional to the sum of the objects in all these selections. However, due to the incremental computation of the brushes, there is a high probability that small brushes are actually computed rather than large ones. Thus, the amount of the data being read from the database, and implicitly the overall processing, is significantly reduced.

6 Related Work

Visual hierarchy exploration. There have been considerable research efforts in the visualization area for finding effective methods to display and explore hierarchical information, such as Tree-Maps [26], Cone-Trees [23] and Reconfigurable Disc Trees [15]. However, most of these methods provide only modest modes of interactions for navigating the hierarchy. Navigation however plays an important role in aiding users to find their way through the complex structure: to see where they are, what information is available and how to identify information of interest.

On the other hand, techniques for visual exploration of hierarchies [19] have independently been proposed. Hierarchy visualizations are evident for instance in many commercial applications, such as Microsoft Windows Explorer, Norton Commander, and so on. The major disadvantage of such interfaces however is that there is a limited display space for the hierarchy. Hence, they are not suitable for displaying large data. The visualization technique we considered for this integration [9, 10] has both the capability of interactively navigating the hierarchical structures and the capability of displaying large datasets.

Special techniques for hierarchy encoding. There has been also work in the area of hierarchy representation and querying. Various techniques that increase the efficiency of particular classes of hierarchical traversal operations have been proposed. Ciaccia et al. [3], for instance, used the mathematical properties of *simple continued fractions* for encoding tree hierarchies. Basically, each node of the tree has a unique code that encodes the ancestor path from that node up to the root. The sequence gives us the code of the ancestors of a node without any physical access to the data. This information is sufficient for performing some operations, such as getting the first common ancestor of two nodes or testing if a node is the ancestor of another one, without any recursive retrieval of data. However, given a node n, this method cannot for example efficiently provide the list of descendants of n. This limitation reduces the number of operations that can be supported and, moreover, makes updates difficult to handle.

A similar idea was introduced by Teuhola in [30]. He used a so called *signature* for encoding the ancestor path. The important difference of the signature method to the previous approach is that now the code is not unique. Given a node n, the code of n is obtained by applying a hash function to it and by concatenating the resulting value with the code of its parent. The non-unique code can make the quantity of data retrieved be much larger than needed. Moreover, the code obtained by the concatenation of all ancestor codes could exceed the available precision for deep trees. A fragmentation of the initial tree and consequently additional joins would thus need to be performed.

Visualization-database integration. Integrated visualization-database systems such as Tioga [27], IDEA [25], DEVise [20] represent probably the work closest related to ours in terms of problem area. The approaches are however different. Tioga [27] implements a multiple browser architecture for what they call a *recipe*, a visual query. The system is able to buffer the computed data; however, the problem of query translation is not present since database queries are explicitly specified by the graphical interface. IDEA [25] is an integrated set of tools to support interactive data analysis and exploration. Some constraints on the data model are imposed by the application domain, but on-line query translation and memory management are not addressed. In DEVise [20], a set of query and visualization primitives to support data analysis is provided. The relationship between these two classes of primitives is complex since the number of primitives supported is relatively rich. However, caching data is being done at the database level using default mechanisms only; special memory management techniques are not considered.

Other work in the same area includes dynamic query interfaces [29, 14], dynamic query histograms [7] and direct manipulation query interfaces [13, 16, 12]. They all have a visual interface and a database back-end. However, the operations translate differently: to dynamic range queries in [7], to temporal queries in [13] and to 2-D spatial queries in [16]. These works do not deal with hierarchy exploration support.

Caching and buffer management. High level caching systems in which objects are not individually identified, but rather a set of objects together is identified with the query that generated it is called *semantic caching* [6] or *predicate caching* [17]. Our buffer management requirements are in principle similar to those present in semantic cashing. The buffer content is specified by a query, i.e., the parameters of the active window. In our system however only one query is required at a time, thus further influencing the strategy used for preventing duplicate entries in the buffer. Other work in the area of object level caching for database applications have been addressed for example in [5, 24]. Due to not needing to fully consider caching, our buffer management strategy is simple, yet efficient – not requiring any unnecessary data accesses.

7 Conclusions and Future Work

With the increasing amount of data being accumulated nowadays, the need for visually exploring large datasets becomes more and more important. A viable way to achieve scalability in visualization is to integrate visualization applications with database management systems. Such integrations raise however two kind of problems. First, it requires the design of an organization of data and a corresponding query mechanism such that all front-end operations can easily be translated into efficient database operations. Second, a good memory management strategy should be employed in order to reduce the overhead of database accesses and thus make the use of the database transparent to end-users. This paper presents a solution that addresses both aspects. The approach is being used in coupling XmdvTool, a visualization application for interactive exploration of multivariate data, with an Oracle8i database management system. Experiments for assessing the method showed that, despite the recursive nature of the operations at the interface level, the processing time in our integrated system is only proportional to the number of objects in the active selection. Moreover, the system scales linearly with respect to the size of the dataset.

Directions for further research include the design of alternative cache policies for the main memory buffer. Moreover, when the system is idle, new objects can be fetched into the buffer in advance. In this case, the system needs to be able to determine the likelihood that an object is used in the near future. This prediction can be based on the past history of user accesses or on common navigation patterns.

References

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

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

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

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

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

[6] 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.

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

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

[9] 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.

[10] 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.

[11] J. Haslett, R. Bradley, P. Craig, A. Unwin, and G. Wills. Dynamic graphics for exploring spatial data with application to locating global and local anomalies. *Statistical Computing 45(3), p. 234-42, 1991*, 1991.

[12] 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.

[13] 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.

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

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

[16] S. Kaushik and E. A. Rundensteiner. SVIQUEL: A spatial visual query and exploration language. *Lecture Notes in Computer Science*, 1460, 1998.

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

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

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

[20] 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.

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

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

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

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

[25] 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.

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

[27] 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.

[28] I. D. Stroe, E. A. Rundensteiner, and M. O. Ward. Minmax trees: Efficient relational operation support for hierarchy data exploration. Technical Report TR-99-37, Worcester Polytechnic Institute, Computer Science Department, 1999.

[29] 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.

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

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

[32] M. O. Ward. Xmdvtool: Integrating multiple methods for visualizing multivariate data. *Proc. of Visualization*, pages 326–333, 1994.

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