WPI-CS-TR-99-37 Dec. 1999

MinMax Trees: Efficient Relational Operation Support for Hierarchical Data 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

MinMax Trees: Efficient Relational Operation Support for Hierarchical Data Exploration *

Ionel D. Stroe, Elke A. Rundensteiner, Matthew O. Ward
Computer Science Department
Worcester Polytechnic Institute
Worcester, MA 01609
daniel | rundenst | matt@cs.wpi.edu

June 9, 2000

Abstract

This paper presents a new method for implementing hierarchical navigation operations in relational database systems as required for applications such as interactive visual explorations over large datasets. A particular class of recursive unions of joins and divisions that operate on hierarchies has been identified. By using adequate pre-computation (i.e., organizing the hierarchical structure as what we called a MinMax tree), the recursive processing of the operations in this class is shown to be reducible to range queries. Extensions of the method for non-tree as well as for dynamic hierarchies have also been designed. We implemented our technique using Oracle 7 and C as a host language and showed that the MinMax method performed significantly faster than the equivalent recursive ones. Our method has been applied to implement navigation operations in XmdvTool, a visualization system for exploration of multivariate data.

Keywords: Hierarchy Encoding, Hierarchical Drill-Down and Roll-Up, Recursive Query Processing, Relational Databases.

1 Introduction

Recently, we have introduced a new technique for visual exploration and analysis of large datasets based on a hierarchical structuring of the information [FWR99a]. The technique, incorporated in

*This work is supported under NSF grant IIS-9732897.

the XmdvTool visualization system [War94], is part of our on-going efforts in the area of scaling visualization technology to be effective for large datasets. The work presented in this paper has correspondingly been driven by the visualization requirements: it originates in the need of providing efficient database transcriptions to the operations defined in the visualization context, such as visual hierarchical drill-down and roll-up. However, that context is not implicitly or explicitly assumed by the paper. Our method is general and can be applied whenever a hierarchical structure and similar navigation operations are present.

Hierarchical structures occur in a large variety of domains, including CAD/CAM applications, cartography and software management. Most of the time, the operations defined on a hierarchical structure include some ancestor and descendent computation (such as check-in and check-out procedures in CAD [KGM91, JR99]) or variants of the graph transitive closure [JHR98]. Typically, the operations have a recursive definition and require expensive computation for each step of the recursion.

Traditionally, query languages have not offered support for handling recursion. The only means to implement a recursive function in a relational database system has been to use the recursive capabilities of the host languages. However, the number of system calls required by such recursive operations is typically very high with substantial processing involved in each call. Recently, vendors have started to include facilities for recursion in their products. A recursive union operator has been proposed as standard for SQL3 [JCC]. Our method could provide an efficient implementation for the operator.

Compared with alternate approaches for similar problems from the literature [CMT89, Teu96], our method is superior in terms of both efficiency and functionality. Thus, the previous proposed methods either do not support dynamically and arbitrary hierarchies, or were not able to efficiently scale to large datasets. Moreover, neither of them do not provide fast computations for both ascendent and descendent sets of nodes.

The approach that we describe in this paper is based on a new type of a data structure, called a MinMax tree. By formulating hierarchical data as a MinMax tree, the recursive computation gets replaced by equivalent but more efficient range-based operations. For a particular dataset, for example, the processing time required for completing a specific operation on an Alpha DEC station running Oracle 7 decreases from 50 down to 2 minutes. We show that MinMax trees can be applied to static tree hierarchies as well as dynamic or arbitrary (i.e., non-tree) hierarchies.

The paper is organized as follows. The related work is presented in Section 2. Section 3 introduces the clustering and navigation operations defined on hierarchies and motivates the paper from an application point of view. The definition and the use of MinMax trees is further discussed in Section 4. An analytical comparison of the method to competing solutions is presented in Section 5, while an experimental evaluation is described in Section 5. Section 6 concludes the paper and presents some future directions for our work.

2 Related Work

2.1 Optimizing Join Processing

In relational systems, hierarchies (as composite objects) have to be broken-down into multiple fragments that are then stored as tuples in separate relations [VKC86]. Traversing the hierarchical structure in order to gather all fragments together or to find out some specific properties requires thus a large number of joins. Relational joins are unfortunately expensive operations and, therefore, an immediate improvement in handling hierarchical structures would be achieved by improving join efficiency.

Valduriez et al. [Val87] introduce a new access path for processing joins, called a *join index*. The join index is simply a binary relation that contains pairs of surrogates (unique system identifiers) of the tuples that are to be joined. An algorithm that uses join indices is also presented in [Val87]. The join index efficiently supports the computation of joins and particularly the join composition of complex objects in the case of a decomposed storage representation [VKC86].

Another method that speeds up join processing uses hidden pointer fields to link the tuples to be joined. The hidden pointers are special attributes that contain record identifiers. Three pointer-based join algorithms, simple variants of the nested-loops, sort-merge and hybrid-hash join, are presented and analyzed in [SC90].

A hash-based method for large main memory systems is described in [Sha86]. The author concentrates on the improvement of joins based on the traditional strategy of sort and merge. Three algorithms are evaluated: a simple hash, the GRACE hash from the 5th Generation Systems, and a hybrid of the two. When the available main memory exceeds at least the square root of the size of one relation, the hash-based algorithms can successfully be applied for computing joins. Their gain is especially significant when large relations are involved.

The above techniques provide support in implementing joins but they do not limit the recursive processing in any way. The number of system calls is unchanged, i.e., high and unnecessary intermediate tuples are retrieved.

2.2 Implementing Recursive Query Processing

Rosenthal et al. [RHDM86] proposed a new model for defining recursive operations in graph structures. They identified a class of recursion, called *traversal recursion*, for which special algorithms for processing recursive queries were designed. The new model offered more information to the query optimization module that could be used to reduce the number of steps in the recursion. The algorithms, however, do not completely eliminate the unnecessary intermediate processing and therefore the number of calls (SELECTs) continues to be high.

Another way to handle hierarchies has emerged with the development of object-relational systems [Sto86]. Using object extensions a composite object can be represented using nested (non 1-st NF) relations. However, recursive relations do not always have a pre-defined depth and therefore they cannot be represented simply using nesting. A "buffered" recursion can be achieved by dividing and storing the hierarchy as a collection of fixed-depth trees, but this solution only decreases the computation load by a factor.

A novel idea in hierarchical processing was introduced by Ciaccia et al. [CMT89]. They encode tree hierarchies based on the mathematical properties of *simple continued fractions* (SICFs). Basically, each node of the tree has a unique label that encodes the ancestor path from that node up to the root. The trees are assumed to be ordered (i.e., children have order numbers) so that the ancestor path simply corresponds to a sequence of integers. The sequence gives us the code of the ancestors of a node without any physical access to the data. This information is sufficient for performing some operations, such as getting the first common ancestor of 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. Another important limitation of this method is that it can only be applied to tree hierarchies and not to arbitrary hierarchies.

A similar idea was introduced by Teuhola in [Teu96]. He also used a code, called a *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 signature method can be used for arbitrary (non-tree) hierarchies and has good behaviour when used for dynamic hierarchies (i.e., frequent updates). However, although it offers a good base for clustering, the non-unique code can make the quantity of data retrieved be much larger than needed. Moreover, the code is obtained by the concatenation of all ancestor codes. When the depth of the tree is large, the code could exceed the available precision, and then a fragmentation of the initial tree would need to be performed. Each new fragment increases the number of codes that overlap and introduces an additional set of joins to be performed. As indicated earlier, new joins correspond to new system calls.

The new method we describe in this paper also uses some pre-computation in order to replace the recursive queries by a set of simpler non-recursive ones. However, it outperforms previously proposed methods in respect to both functionality and efficiency, as will be shown in Section 5. The method can be applied to arbitrary (non-tree) hierarchies, and to both static and dynamic environments.

3 Definition of Hierarchical Navigation Operations

3.1 XmdvTool: The Motivating Application

We will first describe XmdvTool, the application for which our method was originally designed, as one motivating application for why such operator support is needed. As shown in Section 4.4 and Appendix C.1.3 however, the operations that we will introduce are general and can be applied in other application domains, such as CAD/CAM, as well.

XmdvTool [War94] is a visualization tool designed for exploration and analysis of multivariate data. The tool provides help in detecting interesting patterns and trends in the underlying data by mapping it to different graphical constructs, such as parallel coordinates, glyphs, scatterplots or dimensional stacking. Each of these constructs reveals some characteristics of the data while possibly hiding others. The user can dynamically change the graphical construct to be used during exploration and can filter the data to be displayed. A screen-shot of XmdvTool is presented in Fig. 1.

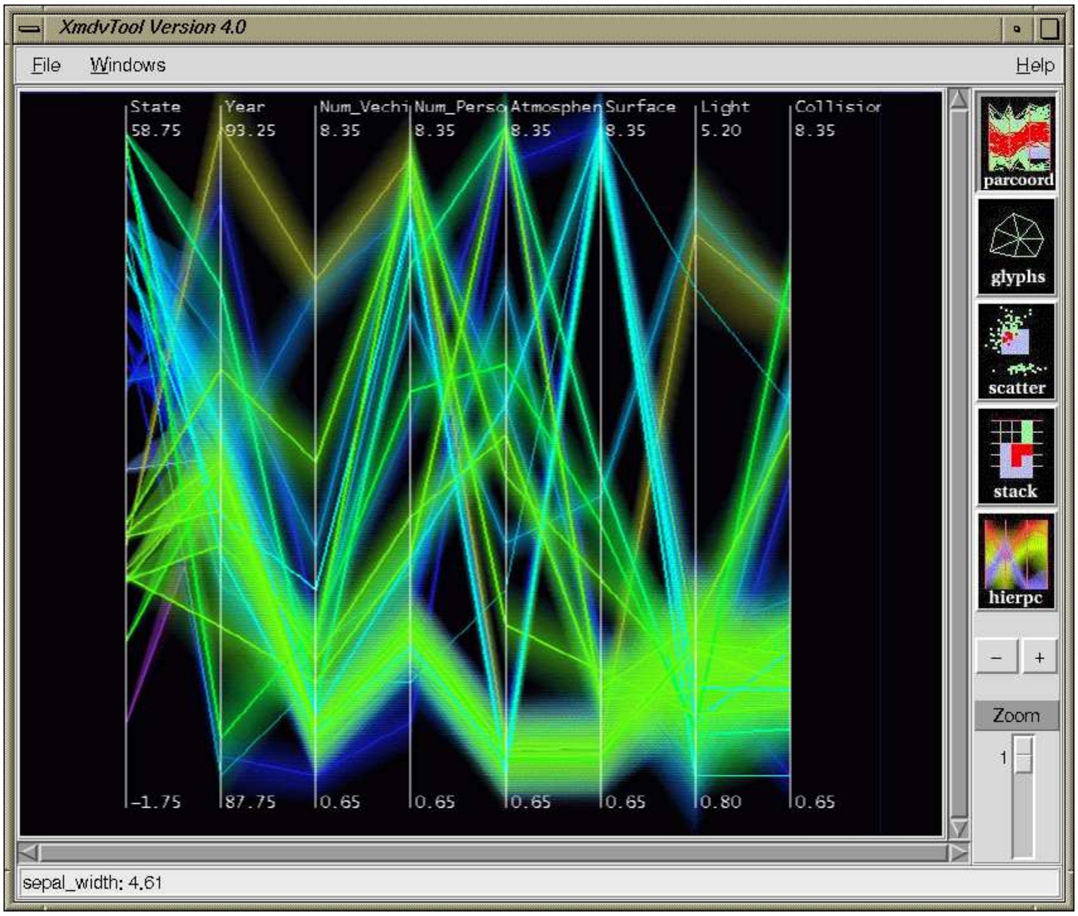

Figure 1: Hierarchical representation of data using *hierarchical parallel coordinates* in XmdvTool.

However, similar to other visualization tools, one of the limitations of XmdvTool has been, so far, scaling to large data sets. This limitation has two aspects: a visualization aspect and a data management aspect. The visualization aspect refers to the high level of clutter that occurs when large data sets are shown on a computer screen which, no matter how advanced the technology, has a limited display capacity. The cluttering seriously reduces the amount of information one can perceive and implicitly the quality of the analysis being performed. The data management aspect refers to the efficiency of storing and accessing large data. The first XmdvTool prototype (`http://davis.wpi.edu/~xmdv/`) used only the main memory to store the necessary information during exploration. However, the assumption that all of this information can fit in the main memory is not valid for large data sets.

To overcome the visualization limitation we proposed a hierarchical model for exploring the data [FWR99a, FWR99b]. The data points are recursively grouped into clusters based on a proximity function. The clustering process generates a hierarchical structure in which each hierarchy level corresponds to a level of abstraction at which the data can be visualized. Each cluster maintains some statistical summarizations, such as the number of data points contained or the average value of them. These aggregate values make a cluster behave like a single data point: the user can see and query over the aggregate values of the clusters instead of the actual data points contained in it. Thus the hierarchy can be thought of as a tree in which the leaves are the raw data points and the interior nodes are aggregate values of the sub-trees below them. In what follows we will use the terms clusters and nodes interchangeably.

To overcome the data management limitation we coupled our visualization tool with a database management system (specifically Oracle 8). Given the clustering trees, the challenge is now how to accommodate the management of the hierarchies with the relational model in order to efficiently support all operations required by the visualization tool.

3.2 Hierarchical Clustering in XmdvTool

In what follows, we describe the clustering process used to organize the data in XmdvTool. The clustering phase generates the hierarchical tree which is further used during exploration, but is not a pre-requisite for our technique. Any other method that generates similar a data structure may be used as well.

Let X be a data set composed of m data points. The elements of X are called *base data points.* A hierarchical clustering is obtained by recursively aggregating elements of X into intermediate groups (clusters). Conceptually, the hierarchical clustering can be thought as an iterative process of successive cluster aggregations that starts with the elements of X (m clusters of one element each) and ends with a large cluster that incorporates all the elements of X. A state of this transitory process can be defined as a partition on the elements of X. The next state is also a partition obtained by grouping some of the sub-sets of the previous partition. Two such successive partitions are called *nested.* Consequently, we can define a hierarchical clustering of X as a sequence of nested partitions, in which the first one is the trivial partition and the last one is the set itself. A formal definition of hierarchical clustering is presented in Appendix B.

A graphical representation of an example of hierarchical clustering is presented in Fig. 2 for a set of five elements $\{a, b, c, d, e\}$. We call this representation a *partition map*. Partition maps provide a good intuition for our work, as the operations introduced in Section 3.3 look "naturally" in such a representation.

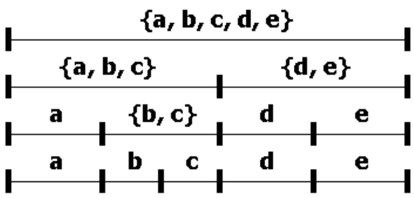

Figure 2: Partition map for a tree hierarchy.

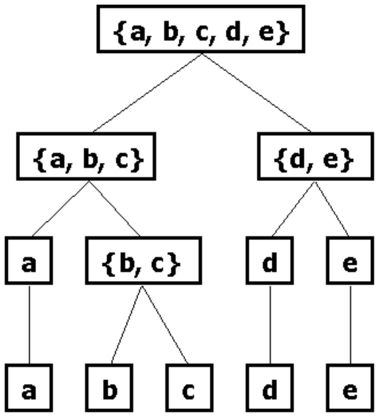

Figure 3: Hierarchical tree obtained by clustering.

A hierarchical clustering may also be organized as a tree structure T, where the root is the whole X and the leaves are base data points. A node of T corresponds to an aggregation whenever it has more than one child. A graphical representation of such a cluster tree, obtained by hierarchical clustering of the same set of five elements, is presented in Fig. 3.

Data can be hierarchically structured either explicitly, based on explicit partitions (such as, for example, in category-driven partitioning) or implicitly, based on the intrinsic values of the data points. In the latter case a clustering algorithm needs to be used to form the hierarchy. Currently, we use BIRCH [ZRL96] as the clustering algorithm in our system, but others would be suitable as well.

3.3 Navigation Operations

Selection is one of the basic operations in exploratory data analysis (EDA). Given a data set, we need to be able to isolate (highlight) some data points of interest for further processing. In case of a flat data, different techniques, based usually on range, have been imagined to define a selection. However, in a hierarchical exploration new types of operations have to be introduced. Below, we present two types of such operations: a structure-based brush and a level-based brush. In XmdvTool these two brushes are combined, so that the current selection is always computed as the intersection of the two brushes. Formal definitions for these operations are given in Appendix C.

3.3.1 Structure-Based Brushes

One of the elementary operations that could be defined over a hierarchical structure is what we term a *structure-based brush* [FWR99b]. A structure-based brush defines a "horizontal" selection on hierarchies based on an initial selection and the structure of the hierarchy. The result of such a brushing operation is a sub-tree of variable size that can be thought of as a "slice" in the original tree.

The process of setting the structure-based brush consists of two phases. In the first phase the leaf nodes that meet a desired property are selected. In the second phase, a number of interior nodes from the upper levels are added to the selection through a mechanism of upward propagation. The process of propagation of phase two is iterated until no more interior nodes can be added. A screen-shot of a structure-based brush in XmdvTool is shown in Fig. 4.

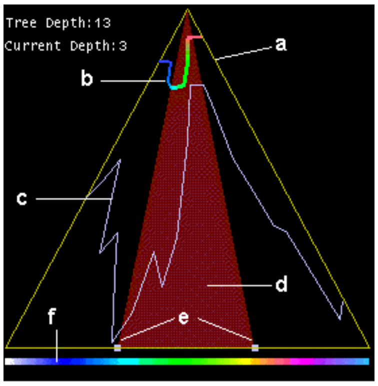

Figure 4: Structure-based brush in XmdvTool. (a) Hierarchical tree frame; (b) Contour corresponding to current level-of-detail; (c) Leaf contour approximates shape of hierarchical tree; (d) Structure-based brush; (e) Interactive brush handles; (f) Color map legend for level-of-detail contour.

Currently, phase one of setting a structure-based brush is determined in XmdvTool by two values. Let us assume that the leaves in the tree are chained (i.e., each leaf has a pointer to the next and to the previous one). The chaining defines a structure of total order on the set of cluster nodes. Given this order, the first phase of setting the structure-based brush consists in selecting all the leaf nodes that fall into the range defined by the two values. Intuitively, the two values represent the left and the right-most leaves. In our system, the values are set by manipulating the two interactive brush handles labeled by e in Fig. 4.

The second phase is a propagation phase. In this phase the initial selection is extended by recursively using a set based operator. The operator determines what are the nodes to be added to the selection next based on the current selection, the structure of the tree, and a predefined semantic. In our system, we have used two propagation operators. They are called ALL and ANY. The ALL operator adds to the selection the nodes that have all of their children currently selected. The ANY operator selects a node if at least one of its children has been selected.

Obviously, the two operators will set different structure-based brushes. Examples of brushes corresponding to ALL and ANY operators are depicted in Fig. 5 and Fig. 6 respectively. The brush values for both examples are nodes 3 and 7. The selected nodes are highlighted by a shaded region. A formal definition of structure-based brushes can be found in Appendix C.1.

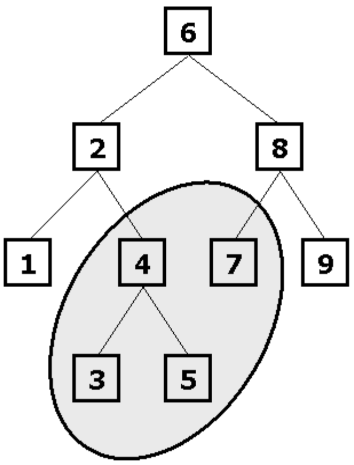

Figure 5: An ALL structure-based brush with the brush values set to 3 and 7.

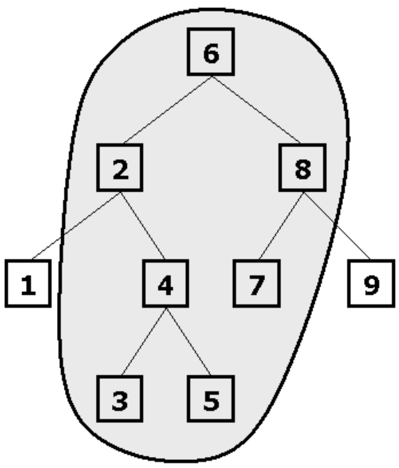

Figure 6: An ANY structure-based brush with the brush values set to 3 and 7.

3.3.2 Level-Based Brushes

Besides the "horizontal" selection, one may also perform a "vertical" selection by using a level-based brush. Level-based brushes allow the user to dynamically change the level of abstraction at which the data is presented. In visualization, this process of increasing and decreasing the level of detail is called drill-down and roll-up, respectively. As we defined clustering as a sequence of nested partitions, choosing a particular level of detail corresponds to choosing a partition. Rolling-up and drilling-down are operations that change the current partition to the previous or to the next one in the sequence.

The number of selected nodes (clusters) varies monotonically when changing the level of detail. It monotonically increases when drilling-down and decreases when rolling-up. Basically, in a drill-down process, one or more clusters are split and their children become new separated clusters, and in a roll-up process two or more clusters merge, forming a larger cluster.

The structure-based brushes define two regions in the space of the tree-nodes, based on whether a node is selected or not (e.g., see shaded and non-shaded regions in Fig. 5 and Fig. 6). XmdvTool allows the user to visualize these two regions at different levels of detail. The result is that rolling-up and drilling down can be performed distinctly for the nodes within or outside the structure brush and therefore one can visualize at the same time two different parts of the hierarchy.

4 The MinMax Technique

4.1 Relational Representation of Data

A standard technique to store hierarchies in a relational system is to use a link-to-parent representation. In such a representation, each tuple has a pointer field that points to its parent except for the root which points to NULL [CMT89, Teu96]. The pointer field is actually a foreign key from the relation to itself. The structure of the table is thus:

HIER (*node_id, parent_id, other information*)

The navigation operations, as defined in Section 3, use iterative traversals of the tree structure based on the parent link. The traversal of the tree can only be performed by repeatedly jumping from one level of the tree to another. For ANY structure-based brushes, this process of jumping between two adjacent levels is basically a join between two instances of table HIER. An ANY structure-based brush can therefore be thought of as a recursive union of joins. For ALL structure-based brushes, the propagation of a selection from one level of the tree to another corresponds to a division between two instances of table HIER. An ALL structure-based brush thus becomes a union of divisions applied to an initial selection. A formal characterization of the relational semantics of the navigation operations is given in Appendix C.1.3.

Joins and especially divisions are expensive operations to be applied repeatedly. Thus, in what follows we will design a technique which replaces these expensive operations by semantically equivalent range query operations. The basic idea is that given some selection parameters and a node of the tree, we want to be able to directly decide whether the node belongs to the selection or not without any supplemental computation (any further join or division traversals). The approach requires that additional information derived from the tree structure be pre-computed and maintained by nodes.

4.2 The MinMax Trees

In this section we introduce the concept of a *MinMax tree*. A MinMax tree is a n-ary tree in which nodes corresponds 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, i.e., a node n having for example two children $c_1 = (\alpha, \beta)$ and $c_2 = (\gamma, \delta)$ will be labeled as $n = (\alpha, \delta)$ (Appendix D). Hence, the trees are called MinMax. Examples of MinMax trees are depicted for a continuous initial set in Fig. 7 and for a discrete initial set in Fig. 8.

Essentially, the process of labeling the nodes is a recursive one. The intervals are computed and assigned off-line at the time the hierarchy is created and their value and distribution (as well as the tree structure itself) depend on the clustering method. Specifically, the interval size and the distribution are influenced by whether the hierarchy is created bottom-up or top-down. In the

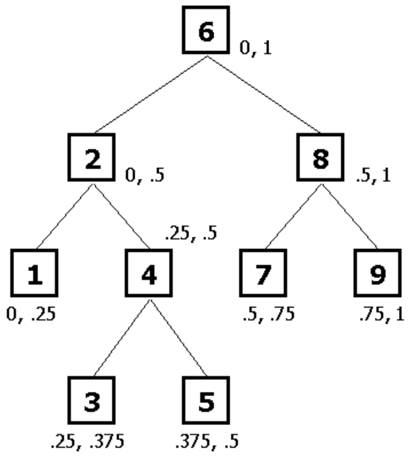

Figure 7: A continuous MinMax tree.

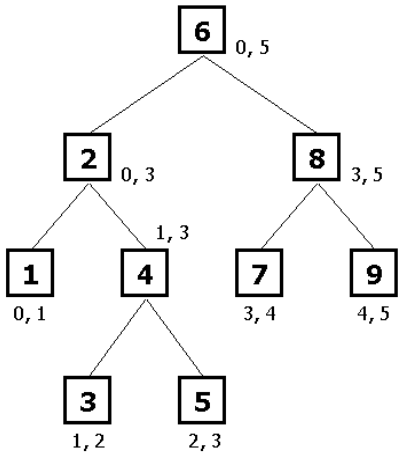

Figure 8: A discrete MinMax tree.

bottom-up case, the leaf intervals have the same size, while in the top-down case, the node intervals on the same level have the same size. Fig. 7 and Fig. 8 presented an example of a top-down tree and an example of a bottom-up tree respectively.

An important property of a MinMax tree is captured in Theorem 1 and will be further exploited when implementing the navigation operations.

Theorem 1 *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 theorem is based on the intuition that each node in the tree is included in its parent as an interval (as constructed). A proof of the theorem is given in Appendix D.1.

4.3 Static Tree Hierarchies

In this section we give an implementation of the navigation operations in the case of a static tree hierarchy, i.e., no updates are present during navigation. 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 and recursively divide it into equal sub-intervals, each sub-interval being assigned to a child (see, for example, the binary tree in Fig. 7).

4.3.1 ALL Structure-Based Brushes

Having the hierarchy labeled as a MinMax tree, we can implement an ALL structure-based brush (as introduced in Section 3.3.1) as a non-recursive operation based on the following property.

Theorem 2 *Given the brush values v_{min} and v_{max}, an ALL structure-based brush generates 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., $(n_1, n_2) \subseteq (v_{min}, v_{max})$.*

The selection defined by an ALL structure-based brush for the example in Fig. 7 and the brush values 3 and 7 is visually depicted in Fig. 9. The selected nodes in the figure are underlined.

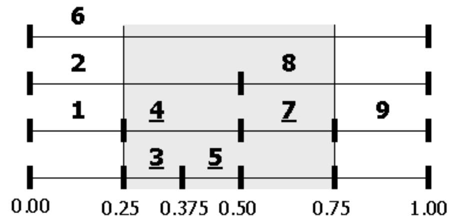

Figure 9: An ALL structure-based brush.

As presented in Section 3.3.2, a level-based brush, which selects nodes on a particular level of detail, can also be specified and manipulated with our tool. Normally, a level brush corresponds to a recursive join. However, if we pre-compute the level value for each node, this level-based brush semantic can also be achieved by a range query.

With this new pre-computed data to be stored in nodes, the structure of the table HIER changes to incorporate *level* (the node level), *e_min* (the minimum extent) and *e_max* (the maximum extent):

HIER (*node_id, parent_id, level, e_min, e_max, other information*)

The ALL structure-based brush for a hierarchy labeled as a MinMax tree is now a simple range query, expressed in SQL2 as:

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

4.3.2 ANY Structure-Based Brushes

An ANY structure-based brush as defined in Section 3.3.1 can also be implemented as a non-recursive operation. The non-recursive computation method is based on Theorem 3.

Theorem 3 *Given the brush values v_{min} and v_{max}, an ANY structure-based brush generates all the nodes $n = (n_1, n_2)$ whose intersection with the brush interval (v_{min}, v_{max}) is not empty, i.e., $(n_1, n_2) \cap (v_{min}, v_{max}) \neq \emptyset$.*

The property states that all nodes that "touch" the brush interval are selected. As shown in Fig. 10, this is intuitively true, all the underlined nodes (that are "touched" by the shaded brush area) are part of the ANY structure-based brush as presented in the example in Section 3.3.1. A proof of Theorem 3 is given in Appendix D.2.

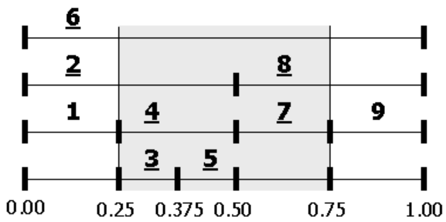

Figure 10: An ANY structure-based brush.

The non-recursive query for an ANY structure-based brush defined over a MinMax tree is therefore of the form:

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

Clearly, the above technique is powerful when the tree structure remains unchanged during exploration. However, in practice nodes often need to be added or removed dynamically. The next subsection addresses the case of a dynamic hierarchy.

4.4 Dynamic Tree Hierarchies

In a dynamic hierarchy, the tree (graph) structure changes during exploration. The type of updates we consider in this section are adding new nodes (as leaves) and deleting existing nodes. If the node to be deleted is an inner node, then we interpret this to mean that the whole sub-tree rooted at that node is removed.

Deleting nodes (sub-trees) in a MinMax tree does not require special computation (such as rearranging the trees or re-labeling the nodes) in order to preserve the properties of the MinMax trees. Deleting a subtree rooted at the node $n = (n_1, n_2)$, for example, is similar to setting an ALL structure based brush with the brush values n_1 and n_2:

```
delete  from hier
where   e_min >= :min
and     e_max <= :max;
```

When inserting a new node n, however, the out-degree of the parent node changes and all siblings of n (and their descendents) need to update their intervals. We say that the node interval "splits". In order to increase the efficiency of this process, we use a two-step method. First, we delay the interval splitting by inserting some "gaps" in the tree nodes (Section 4.4.1). Second, we re-label the affected nodes when splitting by using a fast non-recursive method (Section 4.4.2).

4.4.1 De-Compacting The Tree

Let us consider the case of a node $n = (n_1, n_2)$ that has three children. The method so far divides the (n_1, n_2) interval into three sub-intervals. If a fourth node is inserted, n has to split. If, instead of 3, we first had divided n into more, let's say 5, intervals, the fourth node could have been added without any problem, and thus the splitting would have been delayed.

Based on this idea, we chose the allocation management suggested in [CLR92]. We first label the MinMax tree as an N-ary tree (we say that we "allocate" N positions for each node). Then, when a new node $k + 1$ is inserted in a node n which has only k positions allocated, n just doubles its interval (it expands from k to $2k$ positions) (Fig. 11). By using an amortized analysis, this allocation strategy was proven to be optimal when the maximum number of elements that has to be stored is unknown (and cannot be estimated) ([CLR92]).

4.4.2 Re-Labeling The Nodes

When a node $n = (n_1, n_2)$ splits, a re-labeling process takes place. The extents of all nodes in the sub-tree rooted at n have to be recomputed. But, the sub-tree can be selected based on the n_1 and

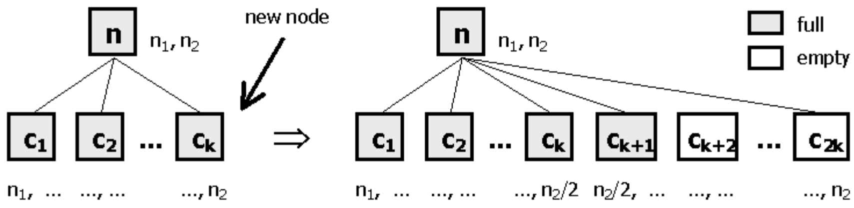

Figure 11: The allocation strategy.

n_2 values. Moreover, for the selected tuples, an affine transformation can be utilized to update the extent values:

```
update  hier
set     e_min = :min+(e_min-:min)/2
        e_max = :max+(e_max-:max)/2
where   e_min >= :min
and     e_max <= :max;
```

4.5 Arbitrary Hierarchies

An arbitrary hierarchy is one in which a node can have more than one parent, i.e., a non-tree acyclic di-graph (Fig. 12). One example application of arbitrary hierarchies is CAD/CAM part hierarchies. In these applications our structure-based brushes have an interesting semantic. Given a set S of basic components (the leaf nodes), an ALL structure-based brush defines the set of super-components that can be manufactured using only parts from S. An ANY structure-based brush gives the super-components that need to use any (at least one) part from S.

One extension of our method can be designed to handle arbitrary hierarchies, too. In an arbitrary hierarchy, more than one interval can be assigned to a node. For example, by using a discrete bottom-up labeling for the tree in Fig. 12, two intervals are assigned to node 5, as shown in Fig. 13.

This case is handled by inserting two copies of node 5 into the HIER table. Thus, the first copy will be assigned the first interval and labeled (0, 1) while the second copy will be assigned the second interval and labeled (2, 3). It is important to notice that the number of additional tuples to be inserted in the HIER table depends on the ordering of the nodes. For example, if nodes 1 and 3 change their position then node 5 will be labeled (1,3) and thus no duplicate copies need to be inserted. However, in this paper we do not address the problem of how to organize the hierarchy nodes in order to decrease the number of stored tuples.

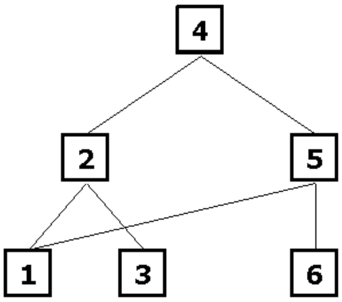

Figure 12: An arbitrary hierarchy.

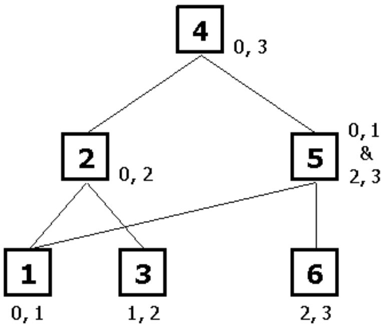

Figure 13: Bottom-up labeling of an arbitrary hierarchy.

If more that one copy of the same node exists in the hierarchy, the (non-recursive) implementation queries for the structure-based brushes change. Thus, because the same nodes may occur multiple times in the table having different interval values, some of them possibly inside and some of them possibly outside the brush interval, the ALL brush becomes "select the nodes that do not have intervals outside the brush interval":

```
select  distinct (everything except node_id)
from    hier
where   level = :lev
except
select  (everything except node_id)
from    hier
where   e_min  > :max
or      e_max  < :min;
```

The ANY structure-based brush query also changes to handle duplicates:

```
select  distinct (everything except node_id)
from    hier
where   e_min < :max
and     e_max > :min
and     level = :lev;
```

While still non-recursive, the new queries are significantly more expensive than those designed for tree hierarchies. Therefore, when no duplicate copies are used, the queries designed for tree hierarchies are preferred.

characteristic being compared	SCF	signature	MinMax
diversity of operations supported	fairly wide	fairly narrow	wide
ascendent computation	good	fairly good	good
descendent computation	fairly poor	fairly good	good
dynamic tree hierarchies	poor	good	fairly good
arbitrary hierarchies	n/a	fairly good	fairly good
large hierarchies handling	good	fairly poor	good

Table 1: Performance comparison of SCF method [CMT89], signature method [Teu96] and our MinMax method.

5 Method Evaluation

In what follows we compare our method against alternate approaches to similar problems from the literature [CMT89, Teu96] in terms of both functionality and efficiency. A summary of our conclusions is presented in Table 1.

5.1 Functionality

We first examine our technique in terms of its ability to support the functions found in [CMT89, Teu96]. Three algorithms that implement functions operating on hierarchies were presented in [CMT89]. The first one computes the *lowest level common ancestor* of two nodes. Given two nodes, the second one tests whether one of them is an *ancestor* of the other or not. The third algorithm generates the keys of the nodes on a given *labeled path* (a sequence of edge order numbers) starting with a specified node. All three algorithms use a main memory computation in order to determine the result tuples. To access the actual data, key based searches are performed.

Our method can perform all three operations above based on the properties of MinMax trees. Given two nodes $n_1 = (min_1, max_1)$ and $n_2 = (min_2, max_2)$, the lowest level common ancestor is simply the node $(min\{min_1, min_2\}, max\{max_1, max_2\})$. The ancestor test reduces to an inclusion test as shown in Theorem 1. Determining the nodes based on a labeled path can be done in our method if we use a top-down node labeling and the same fan-out value for all nodes. In this case only main memory computation is needed. However, [CMT89] cannot non-recursively retrieve the descendents of a node n without a complete scan of the hierarchy table. It also cannot non-recursively support operations similar to the ALL structure-based brushes.

In [Teu96], only one operation is explicitly supported: the computation of descendents of a given node. Conceptually, the ancestors of a node can also be computed. The main problem with this method is the non-uniqueness of the signatures (node codes). Thus, an exact result to queries is not possible. Instead, a super-set of the desired result is obtained. In a MinMax tree, computing the ancestors or the descendents of a node $n = (min, max)$ could be performed by simply testing whether labels include or are included in the node interval (min, max). Finally, operations to support structure-based brushes, i.e. those that compute a recursive union of joins or divisions, cannot be performed with the method described in [Teu96].

5.2 Efficiency

As shown in Section 5.1, retrieving the ancestors of a given node n reduces to a set of key based searches in [CMT89], to a set of key based searches followed by a scan (to filter the duplicates) in [Teu96], and to a single range query in our method. All methods have therefore a logarithmic number of I/O accesses (approximatively) in the size of the hierarchy relation. [Teu96] performs worse than the other two because of the need for eliminating unnecessary tuples. Similar results can be achieved when computing the descendents, except that [CMT89] uses a complete scan of the hierarchy table in this case, which makes it be inefficient.

In case of a dynamic n-ary tree, [CMT89] again performs poorly. The descendent code needs to be recursively updated for a whole sub-tree each time a node is inserted or deleted. [Teu96] performs the best in this case. Because of the hashing method used, the nodes do not change their code value when updating the tree, so no action has to be taken. As shown in Section 4, our method uses in the worst case a single update command to update the tree. The complexity is again logarithmic.

The case of an arbitrary hierarchy is not supported in [CMT89]. In [Teu96], arbitrary hierarchies are handled based on a spanning tree coverage. The efficiency of the method depends significantly on the underlying graph structure, but the method is essentially a recursive one. In our method, arbitrary hierarchies are handled by using range queries with duplicate elimination. The complexity of these operations is $log\ n + m \cdot log\ m$, where n is the cardinality of the hierarchy table and m is the cardinality of the result.

Both our method and [CMT89] are scalable to large hierarchical structures. Neither of them get changed when the tree is too wide or too deep. The performance of [Teu96], however, may decrease drastically for all operations when the depth of the tree increases, due to the tree fragmentation. The process is explained in Section 2.

6 Implementation Results

6.1 Settings

We ran a set of experiments to compare our MinMax method against the traditional recursive approach on standard commercial technology (Oracle 7). These experiments are designed to assess the significance of the processing time improvement achieved by our method when computing structure-based brushes. Then, we studied the functional behaviour of level-based brushes and analyze the extend in which they reduce the result size. Further, we varied the size of the dataset and estimated the feasibility of using an incremental approach for computing the brushes. Testing our MinMax method against the two previous presented ones [CMT89, Teu96] was not possible since the last two methods could not support all the operations we needed.

The performance function we measured during the experiments was the response time (processing time). This is natural considering that our method was primarily designed to be applied in interactive systems. All the experiments were ran ten times and the response time presented is the average of these ten runs.

For the experiments we used various datasets, both real and synthetic, with always consistent results. However, the measurements reported in this paper were obtained on synthetic data. We made this choice to avoid the problems induced by non-homogeneous structures in interpreting the result (the performance depends 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 four datasets had D1=2^{12} (4,096), D2=2^{16} (65,536), D3=2^{17} (131,072), and D4=2^{18} (262,144) tuples in the hierarchical table. The number of dimensions for the data points was 8. The brushes were set to (0, .2), (0, .4), (0, .6), (0, .8) and (0, 1). 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.

6.2 Experiments

Experiment 1: MinMax vs. Recursive. First, we tested our method against two recursive ones. The recursive techniques differed in the way they implemented the mark-up of the intermediary tuples being selected during the recursion steps. While the first technique used an additional attribute to mark the selected tuples, the second one stored the selected tuples each time in a new table (note that SQL does not support recursive views). In both cases we created indexes on all the selection attributes (*node_id*, *parent_id*, *level*,*e_min*, *e_max*). The cost of maintaining the indexes was not an issue because the trees were static during our experiments.

For this experiment we used three datasets D1 – D3 as described in Section 6.1. The result of the experiment is presented in Fig. 14 – Fig. 19. We tested both the ALL structure-based brushes (Fig. 14, Fig. 16, Fig. 18) and the ANY structure-based brushes (Fig. 15, Fig. 17, Fig. 19). As one can see, each of the two recursive approaches (called *RecATT* and *RecTAB* respectively) is more efficient than the other for one particular brush. As expected however the MinMax method is substantially faster than both the recursive ones in all cases.

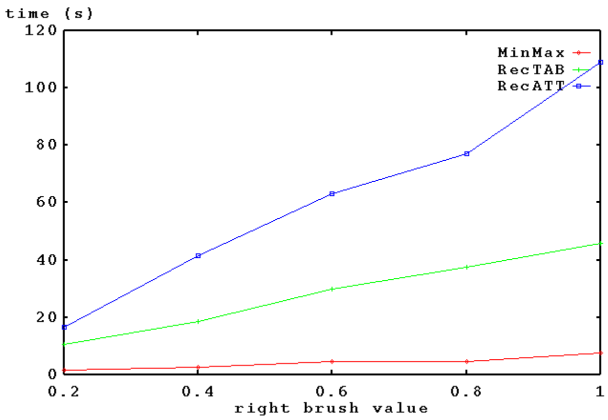

Figure 14: MinMax vs. Recursive. ALL structure-based brushes for dataset D1.

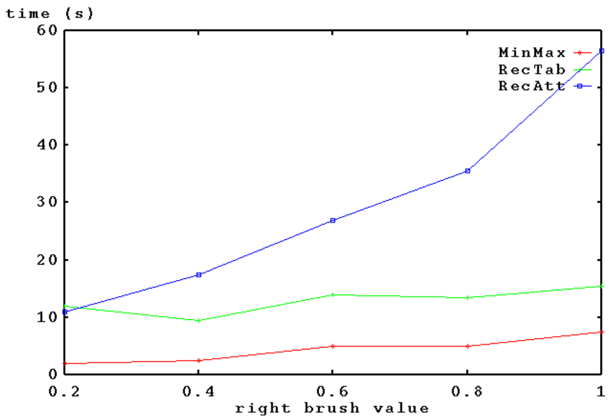

Figure 15: MinMax vs. Recursive. ANY structure-based brushes for dataset D1.

Experiment 2: Structure vs. Level & Structure. In this experiment we tested the impact of applying a level-based brush in addition to a structure-based brush, while the previous experiment tested only the case of setting whole structure-based brushes. Selecting only one level in the structure-based brush reduces the size of the result and therefore the processing time.

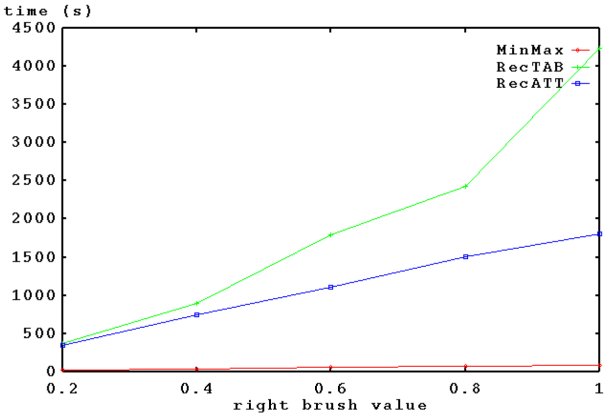

Figure 16: MinMax vs. Recursive. ALL structure-based brushes for dataset D2.

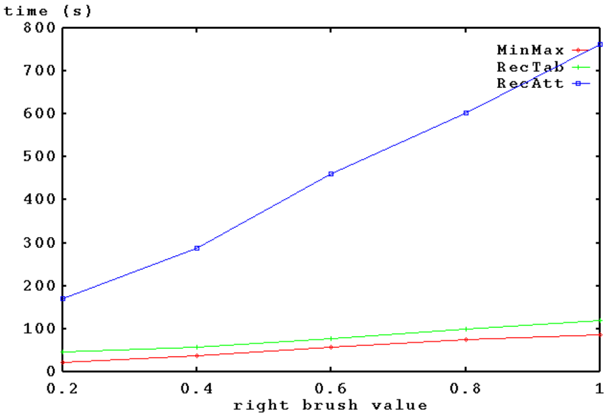

Figure 17: MinMax vs. Recursive. ANY structure-based brushes for dataset D2.

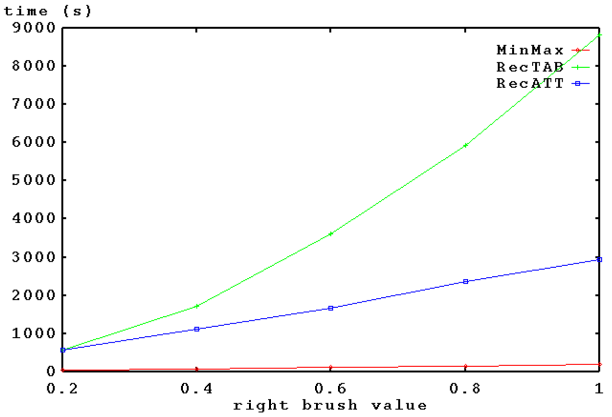

Figure 18: MinMax vs. Recursive. ALL structure-based brushes for dataset D3.

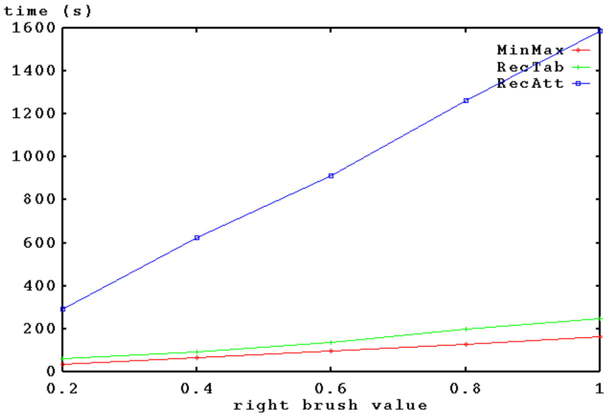

Figure 19: MinMax vs. Recursive. ANY structure-based brushes for dataset D3.

We used four datasets D1 – D4 as characterized in Section 6.1 and the same level for brushing (6) during experiment 2 (level 6 is close to the median level in all the datasets). As shown in Fig. 20 – Fig. 23, the decrease in time due to applying the level-based brushes is considerable (the time of selecting level 6 only in the structure-based brush is systematically low –below 2 seconds– while the time of selecting the whole brush increases more than linearly –up to 1000 seconds for dataset D4).

Experiment 3: Varying the level-based brush. In Experiment 3 we studied the behaviour of the level-based brushes when varying the brush level. Specifically, we exponentially increased the number of selected points and analyzed the increase in the response time.

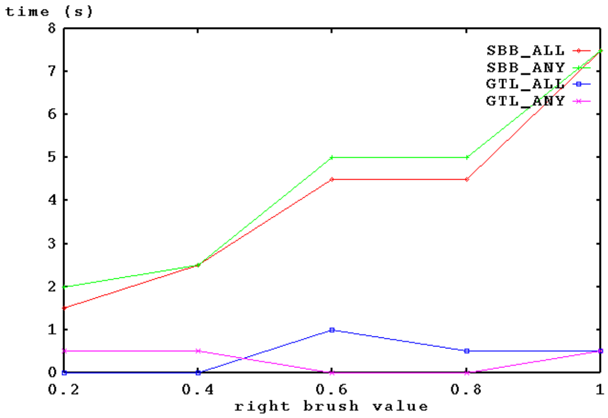

Figure 20: Structure vs. Level & Structure. LEVEL = 6. Dataset D1.

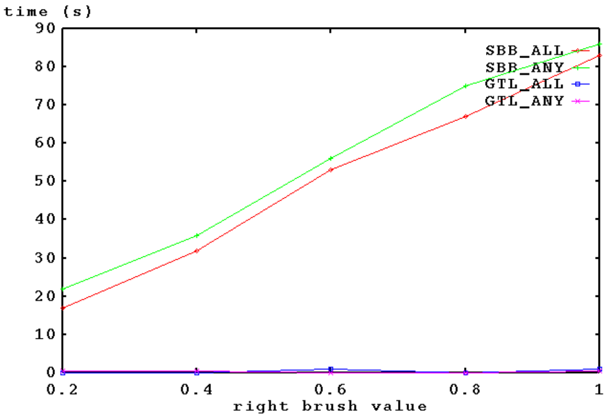

Figure 21: Structure vs. Level & Structure. LEVEL = 6. Dataset D2.

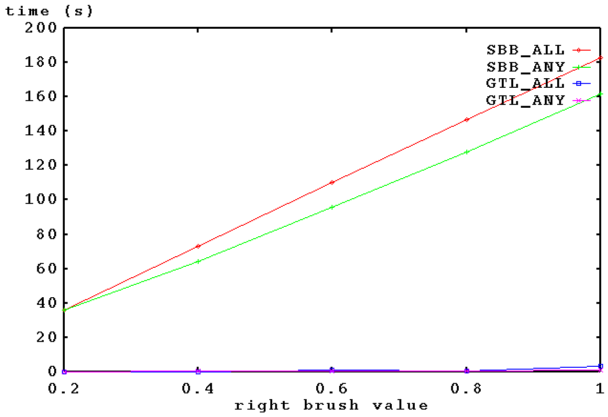

Figure 22: Structure vs. Level & Structure. LEVEL = 6. Dataset D3.

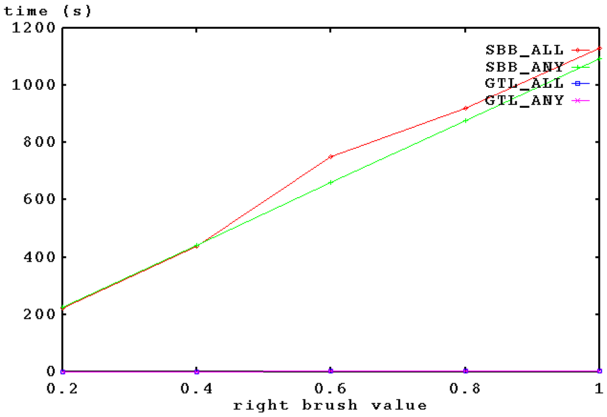

Figure 23: Structure vs. Level & Structure. LEVEL = 6. Dataset D4.

We used for this purpose dataset D3 and measured the response time for setting level-based brushes to levels 4, 8, 12, and 16. The result is that the response time increases also (at least exponentially), except for the low levels. In these cases, the selected tuples are likely to be stored in the same disk page due to our use of a level-based ordering of the tuples. Fig. 24 and Fig. 25 present the experiment result using a normal Y axis, while Fig. 26 and Fig. 27 show the same results on a logarithmic Y scale.

Experiment 4: Varying the dataset size. In the last experiment, we analyzed how the dataset size influences the processing time and whether an incremental computation of the brushes

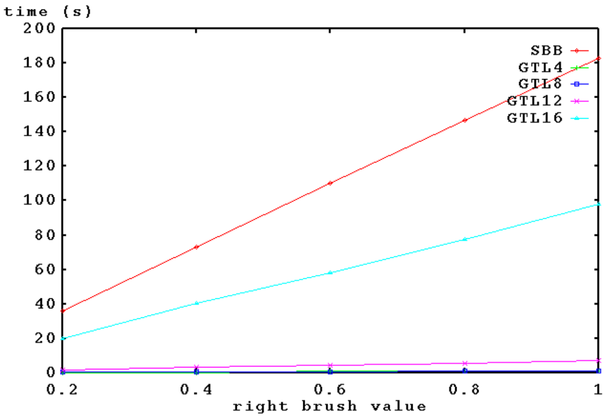

Figure 24: Varying the level-based brush. Dataset D3. ALL structure-based brushes.

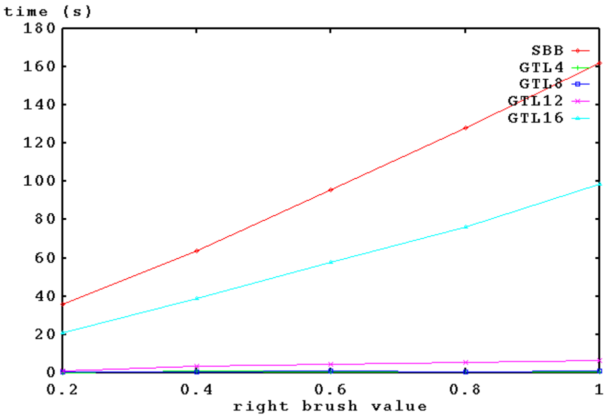

Figure 25: Varying the level-based brush. Dataset D3. ANY structure-based brushes.

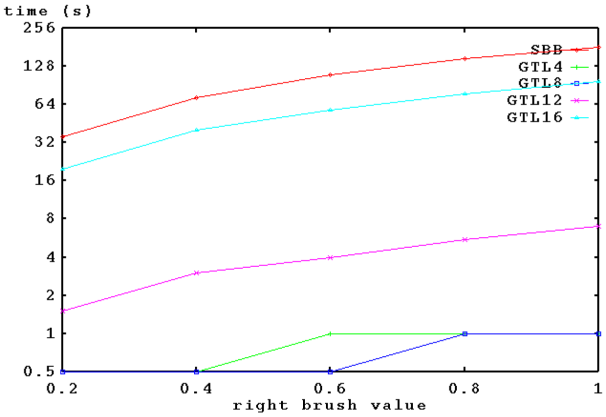

Figure 26: Varying the level-based brush. Dataset D3. ALL structure-based brushes. Logarithmic Y scale.

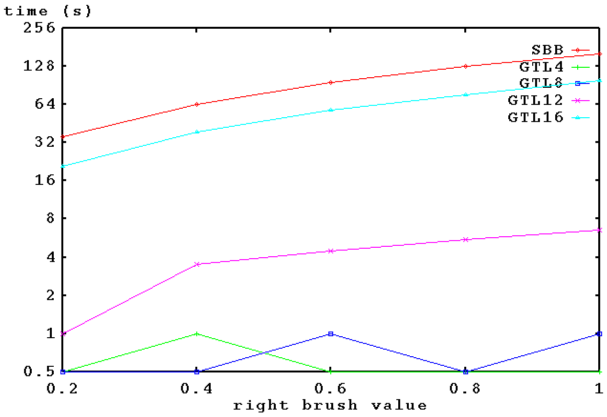

Figure 27: Varying the level-based brush. Dataset D3. ANY structure-based brushes. Logarithmic Y scale.

is useful or not. We tested five different brush values (0-0.2, 0-0.4, 0-0.6, 0-0.8, 0-1.0) for ALL (Fig. 28) as well as for ANY (Fig. 29) structure-based brushes.

The result is that the processing time increases more that linearly when varying the dataset size, and therefore enlarging or restricting the brushes incrementally is very appropriate (i.e., it does improve the response time considerably).

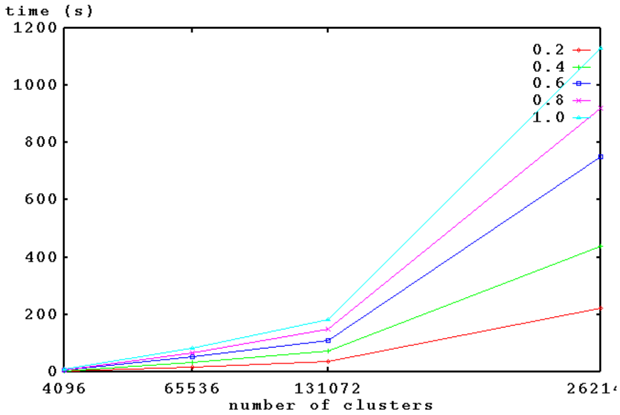

Figure 28: Varying the dataset size. ALL structure-based brushes.

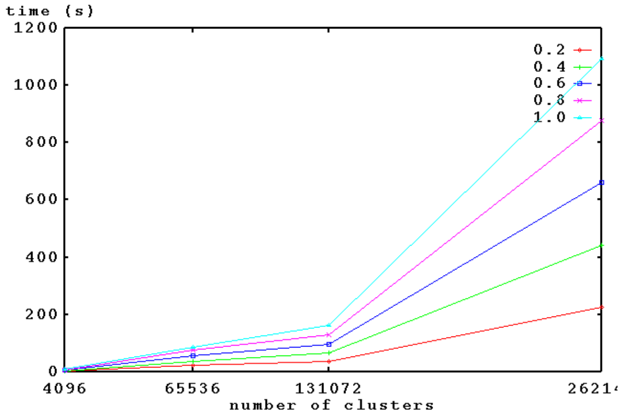

Figure 29: Varying the dataset size. ANY structure-based brushes.

7 Conclusions and Future Work

The MinMax method was developed in order to improve the suitability of relational systems for a class of new applications such as visualization tools or CAD/CAM applications. The method provides important advantages over the traditional recursive approaches when implementing navigation operations on hierarchical structures due to its fast way to compute the ancestors and the descendents of the nodes in the tree. In particular, the computation of recursive unions of joins and divisions in such a hierarchical structure can efficiently be accomplished by organizing the hierarchy as a MinMax tree.

Compared with similar techniques from the literature [CMT89, Teu96] that were also designed to avoid recursive processing, our method has been shown to be superior in terms of both efficiency and functionality. Moreover, unlike the other approaches, the MinMax method can efficiently be applied to dynamic and arbitrary (i.e., non-tree) hierarchies.

Our future research needs to include, as shown in Section 4.5, finding an optimal order of the tree nodes in arbitrary hierarchies such that the number of duplicate nodes minimizes and consequently the efficiency of the operations increases. Another direction for further research is the design of a good strategy for incrementally computing the queries. We estimate that an additional improvement in the efficiency can be gained if using an incremental approach.

References

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[CMT89] P. Ciaccia, D. Maio, and P. Tiberio. A method for hierarchy processing in relational systems. *Information Systems*, 14(2):93–105, 1989.

[FWR99a] Y. H. Fua, M. O. Ward, and E. A. Rundensteiner. Hierarchical parallel coordinates for exploration of large datasets. *IEEE Proc. of Visualization*, pages 58–64, October 1999.

[FWR99b] Y. H. Fua, M. O. Ward, and E. A. Rundensteiner. Navigating hierarchies with structure-based brushes. *Proc. of Information Visualization*, pages 58–64, October 1999.

[JCC] SQL standards. `http://www.jcc.com/SQLPages/jccs_sql.htm`.

[JHR98] 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.

[JR99] M. Jones and E. A. Rundensteiner. Database support for the implicit unfolding of hierarchical structures. *IEEE Transaction on Data and Knowledge Engineering*, 1999. (to appear).

[KGM91] T. Keller, G. Graefe, and D. Maier. Efficient assembly of complex objects. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 20(2):148–157, June 1991.

[RHDM86] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola. Traversal recursion: A practical approach to supporting recursive applications. In C. Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 166–176, Washington, D.C., May 1986.

[SC90] E. J. Shekita and M. J. Carey. A performance evaluation of pointer-based joins. In H. Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 300–311. ACM Press, 1990.

[Sha86] L. D. Shapiro. Join processing in database systems with large main memories. *TODS*, 11(3):239–264, 1986.

[Sto86] M. Stonebraker. Inclusion of new types in relational data base systems. In *Proceedings of the International Conference on Data Engineering,*, volume IEEE Computer Society Order Number 655, pages 262–269, Los Angeles, CA, February 1986. IEEE Computer Society, IEEE Computer Society Press.

[Teu96] 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.

[Val87] P. Valduriez. Join indices. *TODS*, 12(2):218–246, 1987.

[VKC86] P. Valduriez, S. Khoshafian, and G. P. Copeland. Implementation techniques of complex objects. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 101–110. Morgan Kaufmann, 1986.

[War94] M. O. Ward. Xmdvtool: Integrating multiple methods for visualizing multivariate data. *Proc. of Visualization*, pages 326–333, 1994.

[ZRL96] 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.

A Notation Conventions

In what follows, $\mathsf{n_1..n_2}$ *will denote a sequence of integers* $\{n_1, n_1+1, ..., n_2\}$*. A variable* v *that takes all the values from* n_1 *to* n_2 *will be denoted by* $\mathsf{v = n_1..n_2}$*. A variable* v *that takes some values from* n_1 *to* n_2 *(not necessarily all of them) will be denoted by* $\mathsf{some\ v \in n_1..n_2}$*.*

B Hierarchical Clustering

Let $X = \{x_1, x_2, ..., x_m\}$ *be a set of base data points. Then,* $P = \{P_1, P_2, ..., P_n\}$ *is defined to be a partition of* X *iff:*

$$\forall\ 1 \leq j \leq n :\ P_j \subseteq X \tag{1}$$

$$\forall\ 1 \leq j_1 \neq j_2 \leq n :\ P_{j_1} \cap P_{j_2} = \emptyset \tag{2}$$

$$\bigcup_{j=1}^{n} P_j = X \tag{3}$$

We denote the set X *on which partition* P *is defined by* $\mathsf{support}(P)$*. When using more than one partition, we indicate the partition number by a superscript, e.g.:* $P^i = \{P_1^i, P_2^i, ..., P_{n_i}^i\}$*. Two partitions* $P^{i_1} = \{P_1^{i_1}, P_2^{i_1}, ..., P_{n_{i_1}}^{i_1}\}$ *and* $P^{i_2} = \{P_1^{i_2}, P_2^{i_2}, ..., P_{n_{i_2}}^{i_2}\}$ *are called nested and denoted by* $P^{i_1} \prec P^{i_2}$ *(*P^{i_1} *nested in* P^{i_2}*), iff:*

$$\mathsf{support}(P^{i_1}) = \mathsf{support}(P^{i_2}) \tag{4}$$

$$\forall\ 1 \leq j_2 \leq n_{i_2} :\ \exists\ 1 \leq j_1 \leq n_{i_1} :\ P_{j_2}^{i_2} \subseteq P_{j_1}^{i_1} \tag{5}$$

A hierarchical clustering of X *can now be defined as a sequence* $P_X = \{P^1, P^2, ..., P^k\}$ of partitions of X, where:

$$P^1 = X \tag{6}$$

$$P^k = \{\{x_1\}, \{x_2\}, ..., \{x_m\}\} \tag{7}$$

$$P^1 \prec P^2 \prec ... \prec P^k \tag{8}$$

C Navigation Operations

The hierarchical clustering process, as described in Appendix B, results in a tree structure that is formed on the P_j^i partitions. Property (5) above gives us the parent cluster ($P_{j_1}^{i_1}$) for each cluster ($P_{j_2}^{i_2}$) in the tree. In what follows, the parent of a node Z will be denoted by $\theta(Z)$. Moreover, for any set of nodes $S = \bigcup Z_t$, we will denote by $\theta(S)$ the set of the parents of S. Thus, $\theta(S) = \bigcup \theta(Z_t)$.

C.1 Structure Based Brushes

A structure based brush is basically a set-based function $SBB : [1..m] \times [1..m] \rightarrow \bigcup_{i=1}^{k} \bigcup_{j=1}^{n_i} P_j^i$ such that $SBB(v_1, v_2) = I(v_1, v_2) \cup T(I(v_1, v_2)) \cup ... \cup T^{k-1}(I(v_1, v_2))$ and where:

1. $I(v_1, v_2) = \bigcup_{some\ j \in 1..n_k} P_j^k \subseteq \bigcup_{j=1}^{n_k} P_j^k$ *is the initial selection operator that is applied to the leaf nodes, and*

2. $T(\bigcup_{some\ j \in 1..n_i} P_j^i \subseteq \bigcup P_j^i) = \bigcup_{some\ j \in 1..n_{i-1}} P_j^{i-1} \subseteq \bigcup P_j^{i-1}$ *is a propagation operator that propagates the selection from one level to its adjacent level. (T^i is the notation for operator T being applied i times.)*

Any structure-based brush is consequently fully defined by providing a specification for the operators I and T.

C.1.1 The ALL Structure Based Brush

In the ALL structure-based brushes, the I operator is defined as:

$$I(v_1, v_2) = \{x_{v_1}, ..., x_{v_2}\} \subseteq \bigcup_{j=1}^{n_k} P_j^k \tag{9}$$

The T operator for an ALL structure-based brush is defined as:

$$T_{ALL}(S = \bigcup_{some\ j \in 1..n_i} P_j^i) \ = \ \{Z \in \bigcup_{j=1}^{n_i} P_j^{i-1} \mid \neg\exists\ Y \in (\bigcup_{j=1}^{n_i} P_j^i) \setminus S :\ Z = \theta(Y)\} \tag{10}$$

C.1.2 The ANY Structure Based Brush

The I operator for the ANY structure-based brushes is identical with the one for the ALL brushes:

$$I(v_1, v_2) = \{x_{v_1}, ..., x_{v_2}\} \subseteq \bigcup_{j=1}^{n_k} P_j^k \tag{11}$$

The propagation operator T is however different:

$$T_{ANY}(S = \bigcup_{some\ j \in 1..n_i} P_j^i) \ = \ \theta(S) \tag{12}$$

C.1.3 The Relational Semantics of the Structure-Based Brushes

According to the definitions in Section C.1.1 and Section C.1.2, the propagation operator corresponds to a division and to a join respectively.

Given a relation R(x, y, ...), where x is a node id and y is the id of the parent of x, and considering S the initial selection defined by the operator I (same for ALL and ANY structure-based brushes), the ALL and ANY structure-based brushes define the following relational operations:

$$SBB_{ALL} = S \cup (S \div R) \cup (S \div R \div R)... = \bigcup_{i=0}^{k-1} S \div \underbrace{R \div ... \div R}_{i\ times} \tag{13}$$

$$SBB_{ANY} = S \cup (S \bowtie R) \cup (S \bowtie R \bowtie R)... = \bigcup_{i=0}^{k-1} S \bowtie \underbrace{R \bowtie ... \bowtie R}_{i\ times} \tag{14}$$

D MinMax Trees

We first define the anc(x) and desc(x) the sets of ancestors and descendents of a node x in the hierarchy:

$$\begin{aligned} \mathsf{anc}(x) = \{y \mid \quad & \exists\ node_1 = x, node_2, ..., node_{t-1}, node_t = y : \forall i = 2..t : \\ & node_i = \theta(node_{i-1})\} \end{aligned} \tag{15}$$

$$\begin{aligned} \mathsf{desc}(x) = \{y \mid \quad & \exists\ node_1 = x, node_2, ..., node_{t-1}, node_t = y : \forall i = 1..t-1 : \\ & node_i = \theta(node_{i+1})\} \end{aligned} \tag{16}$$

Let now $\mathcal{I}$ be the "initial set" and $\{(a_1, b_1), (a_2, b_2), ..., (a_m, b_m)\}$ be the labels of the leaf nodes in that initial set, $(a_j, b_j) \subseteq \mathcal{I} \times \mathcal{I}$, $a_1 \leq b_1 \leq a_2 \leq b_2 \leq ... \leq a_m \leq b_m$ as shown in Section 4. A MinMax tree is a labeled tree of the form $node_t \leftarrow (a_{\alpha(t)}, b_{\beta(t)})$, where:

$$\alpha(t) = min\{j \mid leaf_j \in \mathsf{desc}(node_t)\} \tag{17}$$

$$\beta(t) = max\{j \mid leaf_j \in \mathsf{desc}(node_t)\} \tag{18}$$

$$\{leaf_{\alpha(t)}, ..., leaf_{\beta(t)}\} \subseteq \mathsf{desc}(node_t) \tag{19}$$

D.1 Proof of Theorem 1

Let $x = (a_{x_1}, a_{x_2})$ and $y = (a_{y_1}, a_{y_2})$ be two nodes in the tree.

We first notice that anc() is a monotonic function. Indeed, relation (15) implies that if $x = \theta(y)$ then $\mathsf{anc}(x) \subseteq \mathsf{anc}(y)$. Using induction, we further get that:

$$if\ x = \mathsf{anc}(y)\ then\ \mathsf{anc}(x) \subseteq \mathsf{anc}(y). \tag{20}$$

Analogously, it can be proven that desc() is monotonically, i.e.:

$$if\ y = \mathsf{desc}(x))\ then\ \mathsf{desc}(y) \subseteq \mathsf{desc}(x). \tag{21}$$

Implication $\Rightarrow$*: If $y \in \mathsf{desc}(x)$ then $x_1 \leq y_1 \leq y_2 \leq x_2$.*
Indeed, $y \in \mathsf{desc}(x) \Rightarrow_{(21)} a_{y_1} \in \mathsf{desc}(y) \subseteq \mathsf{desc}(x) \Rightarrow_{(16)} x_1 \leq y_1$. Similarly, $y_2 \leq x_2$. Since $y_1 \leq y_2$ is true by construction, we have proved that $x_1 \leq y_1 \leq y_2 \leq x_2$ (q.e.d.).

Implication $\Leftarrow$*: If $x_1 \leq y_1 \leq y_2 \leq x_2$ then $y \in \mathsf{desc}(x)$.*
We will prove it by contradiction. Let z be the lowest level common ancestor of x and y. Such a node exists since the root is one common ancestor of x and y. It is also unique since two nodes on the same level can not have common descendents. Now if $x = z$ or $y = z$ then necessarily $x = y = z$ and $x_1 = y_1 \leq y_2 = x_2$ (q.e.d.). If $x \neq z$ and $y \neq z$ then there exists a child $c_x = (a_{cx_1}, a_{cx_2})$ of z such that c_x = anc(x) and a child $c_y = (a_{cy_1}, a_{cy_2})$ of z such that c_y = anc(y). $c_x \neq c_y$ follows from the assumption that z has the lowest level among all the common ancestors. By construction, $(cx_1, cx_2) \cap (cy_1, cy_2) = \emptyset$. However, c_x = anc(x) implies that $(x_1, x_2) \subseteq (cx_1, cx_2)$ and

c_y = anc(y) implies that $(y_1, y_2) \subseteq (cy_1, cy_2)$. From the hypothesis $(y_1, y_2) \subseteq (x_1, x_2)$ and therefore $(cx_1, cx_2) \cap (cy_1, cy_2) \supseteq (y_1, y_2) \neq \emptyset$ (contradiction) (q.e.d.).

D.2 Proof of Theorem 2

Let $x = (a_{x_1}, a_{x_2})$ be a node and $SBB = SBB_{ALL}(v_1, v_2)$ be an ALL structure-based brush.

***Implication** $\subseteq$: If $x \in SBB$ then $(x_1, x_2) \subseteq (v_1, v_2)$.*

$x \in SBB \Rightarrow \exists\ 0 \leq i \leq k : x \in T^i(I(v_1, v_2)) \Rightarrow_{(10)} \neg\exists\ y \notin T^{i-1}(I(v_1, v_2)) : \theta(y) = x \Rightarrow p^{-1}(x) \in T^{i-1}(I(v_1, v_2)) \Rightarrow ... \Rightarrow (p^{-1})^i(x) \in T^0(I(v_1, v_2)) = I(v_1, v_2) \Rightarrow a_{x_1} \in I(v_1, v_2)\ \&\ a_{x_2} \in I(v_1, v_2) \Rightarrow x_1 \in (v_1, v_2)\ \&\ x_2 \in (v_1, v_2) \Rightarrow v_1 \leq x_1 \leq x_2 \leq v_2 \Rightarrow (x_1, x_2) \subseteq (v_1, v_2)$ *(q.e.d.).*

***Implication** $\supseteq$: If $(x_1, x_2) \subseteq (v_1, v_2)$ then $x \in SBB$.*

We will prove it by contradiction. If $x = P_j^i$ then: $x \notin SBB \Rightarrow \exists x^{i+1} \in \bigcup_{j=1}^{n_{i+1}} P_j^{i+1} \backslash T^{k-i-1}(I(v_1, v_2)) : x = \theta(x^{i+1}) \Rightarrow \exists x^{i+2} \in \bigcup_{j=1}^{n_{i+2}} P_j^{i+2} \setminus T^{k-i-2}(\ I(v_1, v_2)) : x^{i+1} = \theta(x^{i+2}) \Rightarrow ... \Rightarrow \exists x^k \in \bigcup_{j=1}^{n_k} P_j^k \setminus I(v_1, v_2) : x^{k-1} = \theta(x^k) \Rightarrow \exists x^k = (a_{a_1}, a_{a_2}) \in \mathsf{desc}(x) : x^k \notin I(v_1, v_2) \Rightarrow_{(T1)} \exists (a_1, a_2) \subseteq (x_1, x_2) : (a_1, a_2) \not\subset (v_1, v_2) \Rightarrow (x_1, x_2) \not\subset (v_1, v_2)$ *(contradiction) (q.e.d.).*

D.3 Proof of Theorem 3

Let $x = (a_{x_1}, a_{x_2})$ be a node and $SBB = SBB_{ANY}(v_1, v_2)$ be an ANY structure-based brush.

***Implication** $\subseteq$: If $x \in SBB$ then $(x_1, x_2) \cap (v_1, v_2) \neq \emptyset$.*

$x \in SBB \Rightarrow \exists\ 0 \leq i \leq k : x \in T^i(I(v_1, v_2)) \Rightarrow \exists x^{k-i} = x \in \mathsf{desc}(x) : x \in T^i(I(v_1, v_2)) \Rightarrow \exists x^{k-i+1} \in \mathsf{desc}(x),\ x \in p^{-1}(x^{k-i}) : x^{i+1} \in T^{i-1}(I(v_1, v_2)) \Rightarrow ... \Rightarrow \exists x^k \in \mathsf{desc}(x),\ x \in p^{-1}(x^{k-1}) : x^k = (a_{a_1}, a_{a_2}) \in I(v_1, v_2) \Rightarrow_{(11)} v_1 \leq a_1 \leq a_2 \leq v_2$. *But,* $x^k = (a_{a_1}, a_{a_2}) \in \mathsf{desc}(x) \Rightarrow_{(T1)} x_1 \leq a_1 \leq a_2 \leq x_2 \Rightarrow (x_1, x_2) \cap (v_1, v_2) \supseteq (a_1, a_2) \neq \emptyset$ *(q.e.d.).*

***Implication** $\supseteq$: If $(x_1, x_2) \cap (v_1, v_2) \neq \emptyset$ then $x \in SBB$.*

$(x_1, x_2) \cap (v_1, v_2) \neq \emptyset \Rightarrow \exists t : t \in (x_1, x_2) \cap (v_1, v_2) \equiv t \in (x_1, x_2)\ \&\ t \in (v_1, v_2)$.

$t \in (x_1, x_2) \Rightarrow a_t \in \mathsf{desc}(x) \Rightarrow \exists$ *a path* $\{a_t = x^k, x^{k-1}, ..., x^i = x\}$ *from a_t to x such that* $x^{k-1} = \theta(x^k), x^{k-2} = \theta(x^{k-1}), ..., x^i = \theta(x^{i-1})$.

$t \in (v_1, v_2) \Rightarrow a_t \in I(v_1, v_2) \Rightarrow x^k \in I(v_1, v_2) \Rightarrow_{(12)} x^{k-1} \in T(I(v_1, v_2)) \Rightarrow_{(12)} x^{k-2} \in T^2(I(v_1, v_2)) \Rightarrow_{(12)} ... \Rightarrow_{(12)} x^i \in T^{k-i}(I(v_1, v_2)) \Rightarrow x \in T^{k-i}(I(v_1, v_2)) \Rightarrow x \in SBB$ *(q.e.d).*