VIRTUAL FORCED SPLITTING IN MULTIDIMENSIONAL ACCESS METHODS

by

RICHARD SWINBANK

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
The University of Birmingham
April 2008

Abstract

External, tree-based, multidimensional access methods typically attempt to provide B+ tree like behaviour and performance in the organisation of large collections of multidimensional data. The B+ tree's efficiency comes directly from the fact that it organises data occupying a single dimension, which can be linearly ordered, and partitioned at arbitrary points in that order. Using a multiway tree to partition a multidimensional space becomes increasingly difficult with increasing dimensionality, often leading to the loss of desirable properties like high fanout and low internode overlap.

The K-D-B tree [49] is an example of a structure in which one property, that of zero internode overlap, is provided at the expense of another, high fanout. Its approach to doing this, by *forced splitting*, is shared by a collection of other structures, and in 1995 Freeston suggested a novel approach to mitigate the effects of forced splits, by executing them *virtually.* This approach has not been taken up widely, but we believe it shows a great deal of promise.

In the thesis, we examine the virtual forced splitting approach in depth. We identify a number of problems presented by the approach, and propose solutions to them, allowing us to characterise a general class of virtual forced splitting structures that we call VFS-trees. The efficacy of our approach is demonstrated by our implementation of a new VFS structure, and by what we believe to be the first implementation of a BV-tree, together with new algorithms for region and K Nearest Neighbour search. We further report experimental results on construction, exact-match search and K-NN search of BV-trees, and show how they compare, very favourably, with the corresponding operations on the currently most popular multidimensional file access method, the R*-tree [3].

Acknowledgements

I would like to thank my supervisor, Alan Sexton, for his unstinting support throughout my PhD, and the many friends I have made during my time in Birmingham — you know who you are.

Contents

List of Figures

Chapter 1

Introduction

Providing the means for efficient search of large, disk-resident collections of multidimensional data is a research area of long standing. Multidimensional analogues of the B+ tree, of which Guttman's R-tree [25] was an early example, attempt to do so dynamically; that is, to index a dataset in such a way that its contents can change without causing the query performance of the access method to deteriorate unacceptably. A host of dynamic, tree-based access methods have appeared since the R-tree, and the question remains very much open.

1.1 Motivation

Much of the difficulty in indexing multidimensional datasets has its origins in the tension between a number of competing objectives, and very many access methods can be described as achieving some of these objectives at the expense of others. To return to the R-tree example, the structure is dynamic and provides a guaranteed fanout ratio, but its performance is apt to deteriorate as a result of overlap between the regions of space indexed by individual nodes.

In chapter 3 we characterise the ways in which a number of access methods attempt to achieve these competing objectives, and note that while some approaches — like that of the R-tree — are very common, others are less so. This leads us to an examination of some of these, perhaps underexplored approaches, in particular, that of the BV-tree [19]. The BV-tree appears to move away from some very fundamental properties of tree structures, but does so in such a way that conventional tree search not only remains possible, but becomes more efficient.

1.2 Summary of contributions

The contributions of this thesis are as follows.

1. We describe desirable properties for multidimensional, tree-based, external access methods, and provide a characterisation of existing access methods, based specifically on their ability to provide such properties (or otherwise). Existing surveys, for example [24], essentially catalogue the field at their time of writing, relating structures in terms of their evolution from one to another. We take a more property-based approach: in general, what properties are required of an access method, and how well are these provided by specific structures?

2. We describe, in depth, the concept of *virtual forced splitting*, taking an approach to the description of regions that we make independent of their on-disk representation. This makes clear what virtual forced splitting attempts to achieve, and allows a deep understanding of the often complex issues involved in making it work in practice. This allows us to describe algorithms for a number of essential tree operations across a class of structures employing virtual forced splitting.
3. Clear characterisation of virtual forced splitting has permitted us to produce what we believe to be the first working implementation of a BV-tree. We demonstrate that the BV-tree is a practical, well-behaved structure with performance that can exceed that of the *de facto* industry standard multidimensional access method, the R*-tree [3].
4. We present algorithms using an abstract-machine based approach, which we found useful because it operates at a level of abstraction natural for implementing access methods. Operations are specified as a collection of state transition rules, each of which, individually, describes the behaviour of an algorithm in a single node.

A brief summary of 2 and some results from 3 appear in [55].

1.3 Thesis outline

The thesis is organised as follows.

- In chapter 2, we describe properties required of a multidimensional access method, as a basis for a broad taxonomy of structures. We describe our approach to understanding subtrees in terms of predicates based on spatial relationships, at a potentially higher level than a direct decoding of their on-disk representation. Our abstract-machine approach to algorithm description is introduced.
- In chapter 3, we review significant contributions to the field, and classify them according to their provision of four structural characteristics described in chapter 2.
- In chapter 4, we motivate and introduce the concept of virtual forced splitting, describing issues with the approach that affect the implementation of tree operations.
- In chapter 5, we provide descriptions of, and algorithms for, a number of core tree operations, based on the considerations of chapter 4.
- In chapter 6, we describe three implementations of two VFS-tree structures, focussing on representation-specific issues that fall outside the scope of chapter 5.
- In chapter 7, we present and discuss comparative experimental results from our VFS-tree implementations and other structures.
- In chapter 8, we conclude our discussion, and identify and discuss briefly a number of avenues for future research.

Chapter 2

Preliminaries

In this chapter we present a high level discussion of dynamic, hierarchical, external, multidimensional, point access methods. We begin by describing the fundamental characteristics of this class of structures to establish the scope of our discussion, before outlining some additional properties that, if present, are of benefit.

The second half of the chapter describes our approach to description of structures and algorithms.

2.1 Fundamental characteristics

We characterise members of the class of dynamic, hierarchical, external, multidimensional, point index structures explicitly as follows:

- They are **hierarchical**; almost all structures in this class are trees, although examples of directed acyclic graphs (DAGs) exist.
- They are used to index **multidimensional** datasets; which we take to mean not only datasets of explicitly more than one dimension, but also datasets of unknown or unspecified dimensionality, such as those occupying general metric spaces.
- Elements of datasets organised by these structures are **points**; they have zero spatial extent.
- Structures of the class are **external**, that is to say disk-resident. A tree node consists of an integer number of disk pages, usually one, and has potentially many children. Trees are thus described as n-ary or *multiway.*
- Tree development (growth or contraction) is **dynamic**; development is self-managed, such that subsequent offline reorganisation should not be required for minimum performance guarantees to be met.

Items of data are stored only in the leaf level of the tree; higher levels contain information to direct search to the correct leaf or leaves. For consistency in the discussion that follows, we assume that entries in trees' leaf nodes are always $\langle$*Key*, *Value*$\rangle$ pairs, where *Value* is the data item in storage and *Key* is an attribute that identifies the data item.

2.2 Desirable characteristics

In addition to the fundamental characteristics of the class of structures introduced in section 2.1, we describe four further desirable properties.

2.2.1 IO-balance

An equal IO cost for reading any path from the root of the tree to the leaf level is desirable in order to permit guarantees to be made concerning query execution cost. We draw a distinction between a path's *node-length* (the number of nodes found on the path) and its *IO-length*, the number of pages that must be retrieved from disk in order to read that path. Post-and-grow behaviour, as exhibited by the B+ tree [15], produces trees in which all paths are of equal node-length, and which are described as being *height-balanced*. We refer to trees in which all paths are of equal IO-length as being *IO-balanced*, and observe that not every height-balanced tree is also IO-balanced; if nodes are permitted to be of variable size then the IO cost of reading a path may vary.

The vast majority of hierarchical external access methods are both height-balanced and IO-balanced, but we will later describe structures that, while height-balanced, are not IO-balanced.

2.2.2 The single path property (SPP)

Ideally, it should be possible to identify, unambiguously, a path from the root of a tree to a single leaf, for any point in the data space. In combination with IO-balance, this property ensures that the IO cost of an exact-match query is logarithmic with respect to the number of pages in the tree, and in most cases equal to the tree's height. We refer to this as the *single path property* (SPP), and to structures exhibiting it as SPP structures. The SPP results from insisting that all subtree choices in an internal node are deterministic and mutually exclusive: if a point might be found in a given subtree of a node, it will certainly not be found in any other subtree of that node.

We note that structures possessing the SPP are required to handle the case of duplicate entries rather carefully. If we permit multiple instances of a given key to be stored in the tree, a time may come when those instances cannot be accommodated inside a single disk page. However, splitting a leaf filled with many copies of a single value will produce two leaves that must both be explored when searching for that value: the SPP would then no longer hold. As such, we must ensure that no one value appears in more than one leaf node.

2.2.3 Guaranteed minimum fanout ratio

A guaranteed minimum fanout ratio is essential is order to preserve a true tree structure; *i.e.* a structure with height that scales logarithmically with the number of data (leaf) pages. Lower fanout ratios produce taller trees, and in the limit (fanout ratio = 1) cause the tree to degenerate to a list. Minimum fanout is achieved by specifying a minimum node occupancy, generally specified as a parameter of post-and-grow trees, and enforced (except in the root node) by permitting overflowing nodes to split only within acceptable balance constraints. Maintaining minimum occupancy also ensures better utilisation of disk space, with disk pages being, in general, more heavily occupied.

Quadtree [50]-based approaches, amongst others, do not handle externalisation well; their top-down growth produces unbalanced trees, and quadtree nodes have a constant four children. Variants to ameliorate this behaviour have been proposed, for example the PK-tree [51, 59], a quadtree variant that uses path compression to raise node occupancy and limit tree height. This allows it to claim a probabilistic *expected* maximum height, although it remains unbalanced. Height balance is only guaranteed in structures exhibiting bottom-up growth, effectively limiting it to the class of structures whose nodes undergo binary splits; a quaternary split and post in a hypothetical 'external quadtree' would result in poor node occupancy and low fanout, because even a split in which a node's entries were perfectly evenly distributed would result in four nodes that were each only one quarter full.

We confine our interest to height-balanced structures exhibiting post-and-grow behaviour after binary node splits; trees such as the quadtree family fall outside the scope of our discussion.

2.2.4 Locality preservation

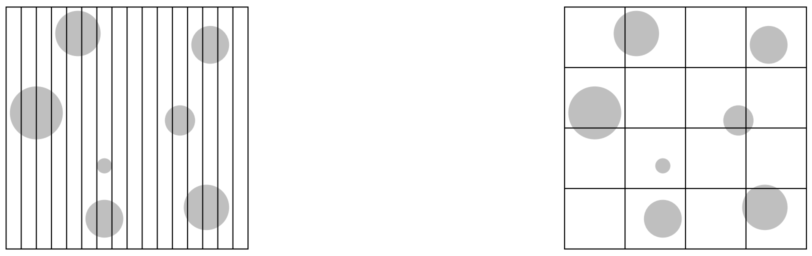
For non-exact-match queries, *i.e.* those with spatial extent, for example Range or K Nearest Neighbour (K-NN) queries, it is not possible to impose an absolute upper bound on execution cost (beyond brute force search of the entire tree), but we still wish to limit it as far as possible. One way to do this is to ensure that the spatial relationships between areas of the data space are preserved in the index structure: put simply, that points that are close together in space are also stored in close proximity in the index structure. This means that queries examining a restricted region of the space will also be confined to a limited portion of the index structure.

As a simple example, consider a database of health records including details of a patient's height and weight, from which the records of all patients of given ranges of both height and weight must be retrieved. A one-dimensional index might be built over height, allowing the query to be processed using that index, but on identifying the disk location of records of qualifying height, *every* record must be retrieved and inspected to test its satisfaction, or otherwise, of the weight criterion. Building an index over both height and weight together, in such a way that records of patients of similar height and weight appear on the same (or nearby) disk pages, would greatly improve the query's efficiency.

Figure 2.1 shows two decompositions of a two-dimensional data space containing clusters of data represented as grey circles. Figure 2.1a gives a decomposition into 16 equal subregions; only one cluster is contained fully in a single subregion, and many clusters occupy several subregions. By contrast, figure 2.1b illustrates a decomposition in simultaneously two dimensions, and in which the likelihood that a cluster will lie across several subregion boundaries is reduced. It is this effect that we seek to emulate in indexing data spaces of two and more dimensions; in general, preservation of locality in multidimensional spaces is achieved only by the partitioning of the space in all dimensions (albeit not necessarily simultaneously).

2.2.5 Summary

Dynamic trees develop from an initial single, root node, so provision of the global properties described above relies on their being induced by a local property of a tree's principal means of structural modification: the node split. We summarise below the global properties and the local split properties that are typically used to induce them:



(a) Spatial decomposition in one dimension only (b) Decomposition in two dimensions simultaneously

Figure 2.1: Splitting in two dimensions better preserves notions of locality in a two-dimensional space.

Global property	**Induced by**
Equal path IO-lengths	Height balance and limited node capacity
Guaranteed minimum fanout ratio	Balanced node split ratios
The single path property (SPP)	Splitting a node's region disjointly
Locality preservation	Partitioning data in all dimensions of the space

Despite the desirability of these properties, it has long been recognised that provision of them all simultaneously is rather difficult (with the exception of special cases like the one-dimensional B+ tree). Very many hierarchical access methods have been proposed for the indexing of multidimensional spaces, all of which, either explicitly or inadvertently, relax the requirement for at least one of them. We use this as the basis of a broad taxonomy of structures in chapter 3, but cite a few examples here by way of illustration.

IO-balance is a feature of almost all post-and-grow structures. The X-tree [8], however, observes that there exist cases where split quality, in terms of overlap between regions resulting from a split, is so poor that efficiency of data access is better served by choosing not to split at all. As a result, the X-tree remains height-balanced but loses its IO-balance. Members of the class of VFS-trees that we will introduce in chapter 4, while dynamically height-balanced (see section 4.5) have varying degrees of IO-balance.

The requirement for a unique, deterministic path for exact-match queries is that most commonly relaxed in index structure implementations, by permitting the existence of overlap between node region descriptors, and thus requiring backtracking search algorithms. The R-tree [25], described in sections 2.4.3 and 3.2.2, is an example of such a structure. Performance improvements in structures of this kind are made typically by applying heuristics for overlap reduction, usually in one or both of two places:

- at a node split (when new entries are created), using some heuristic to govern the separation of node entries into two sets, often referred to as a *split* (or *splitting*) *policy*[1];
- at insertion of an entry into a subtree, when a subtree is selected to accommodate a point, based on some measure of the effect that doing so will have on its region descriptor (or, more commonly, is selected because the deleterious effect on its region descriptor is least).

Such structures are effectively parametric in their split policies and subtree selection algorithms, and indeed are treated formally as such by the Generalized Search Tree (GiST) [26] in the definitions

[1]We believe this expression to have been coined by Ciaccia *et al.* when introducing the M-tree [14].

of **PickSplit** and **Penalty**.

The K-D-B tree [49] enforces disjointness amongst region descriptors by the relaxation of node occupancy requirements. Node splits are selected on the basis of split balance, but any entry whose region is not contained entirely on one side of the split or the other undergoes a forced split that must be propagated throughout the entry's subtree. Imposing a pre-defined split boundary on a descendant node, irrespective of its contents, has the effect of producing poorly balanced splits, forcing the creation of under-occupied nodes.

Locality preservation is of importance only when the notion of locality is useful, such as when support is required for region or nearest neighbour queries. For a structure required to support only exact-match queries it is unnecessary, and may be meaningless in high numbers of dimensions (see section 2.3). Relaxation of the requirement for locality preservation allows mapping of the multidimensional data space into a one-dimensional index space which can then be indexed by an efficient one-dimensional structure, for example a B+ tree.

2.3 The curse of dimensionality

Spaces of higher dimensionality are associated with rapid deterioration in index structure performance with increasing dimensionality. This effect was termed the *curse of dimensionality* by Bellman [4], and has been described as being caused by the exponential growth of the volume of a space with linear growth in its dimensionality [10]. We can illustrate this with an example adapted from Bellman: if we are required to populate the one-dimensional unit space with points such that each point is at most 0.01 away from another, we require 100 points at (say) 0.00, 0.01, . . . , 0.99. To achieve the same in two dimensions, however, we require 10000 points, and in twenty dimensions 10^{40}. In practice, therefore, for a collection of datasets of a common size n and increasing dimensionalities, the space will become rapidly more sparsely occupied. In such a situation the distance between points not only increases, but has been shown (in [10]) probabilistically to converge to a single value, an observation linked by Pestov [47] to the *concentration of measure* in high-dimensional structures.

The practical effect of this is to cause the selectivity between objects on the basis of distance to collapse, indeed Beyer *et al.* [9] argue that the familiar notion of 'nearest neighbour' loses its meaning in high numbers of dimensions, because the requisite notion of 'nearness' has been lost. Nearest neighbour and other distance-based queries typically become more expensive to evaluate (because the region defined around a point by a distance intersects many index partitions which must all be explored), while access methods that rely on distance-based partitioning techniques to manage growth exhibit increased node region overlap and performance deterioration across the board. Similarly, the sparse occupancy of the space requires high-volume partitions in order to enclose sufficient data to meet node minimum occupancy constraints; this in turn makes heuristic reduction of region descriptor overlap very difficult.

A factor often cited as an effect of dimensionality is reduced fanout caused by the number of bytes of storage required to represent a higher dimensional object on disk. While this is certainly the case (and indeed there exist structures, for example the TV-tree [37], that attempt to tackle the problem directly), we contend that this is an artefact of representation rather than a topological feature of the space *per se*.

2.4 Node predicates

2.4.1 Global and local predicates

Hitherto we have described a node's 'region' or 'region descriptor', referring to information, stored in the node's parent entry, indicating the region of space indexed by the node. A region descriptor, R, forms the basis of a predicate, P, used to decide whether or not a point might be found in the entry's subtree, *i.e.* $P = \lambda x.(x \in R)$. This is a notion familiar from the GiST ([26]; see also section 3.2.1). When searching for a point, p, if $P(p)$ evaluates to `False`, the subtree can be pruned from the search. If $P(p)$ evaluates to `True`, the subtree must be explored, although this does not indicate that the point will be found in the subtree, as it may not be stored in the tree at all (indeed the GiST's authors describe node predicates as evaluating to either `False` or '`Maybe`'). We discuss tree operations in a language of node predicates, and introduce some terminology for this here with a simple example.

Consider searching for a point $p = (x_p, y_p, z_p)$ in the kd-tree [5] fragment shown in figure 2.2. Progress of the search into node D implies, given the values in the tree's non-leaf nodes, that $x_p \geqslant 20$, $y_p < 10$ and $z_p \geqslant 15$; to put it another way, progress into node D of a search for p depends on satisfaction of the predicate $x_p \geqslant 20 \wedge y_p < 10 \wedge z_p \geqslant 15$; we refer to this conjunction as node D's *global predicate*:

$$\lambda p.\text{match } p \text{ as } \langle x, y, z\rangle \text{ in } (x \geqslant 20 \wedge y < 10 \wedge z \geqslant 15)$$

where 'match' is used to pattern-match p against the structure $\langle x, y, z\rangle$. We use the term global predicate to distinguish it from what we shall call a tree node's *local predicate*; the predicate used to describe the node in its parent. Node D's local predicate is

$$\lambda p.\text{match } p \text{ as } \langle x, y, z\rangle \text{ in } z \geqslant 15$$

A node's local predicate does not alone characterise the data found at or below that node; a point (x_d, y_d, z_d) at or below node D does not merely satisfy $z_d \geqslant 15$ but satisfies D's global predicate, $x_d \geqslant 20 \wedge y_d < 10 \wedge z_d \geqslant 15$.

We therefore describe the process of tree descent as the satisfaction of a sequence of local predicates, implicitly constructing a node's global predicate as the conjunction of all the local predicates satisfied in order to reach that node. We draw this distinction because algorithms for tree search or manipulation typically operate on local predicates directly, assuming the implicit construction of global predicates; in particular, satisfaction of a node's local predicate is generally treated as being sufficient evidence that its global predicate is also satisfied. We will later discuss structures, however, for which this is not the case.

2.4.2 Notation

We denote a node's local predicate by P_{node} and its global predicate by $\mathscr{P}_{\text{node}}$. Continuing with the example in figure 2.2, we have:

$$\begin{aligned} P_{\text{D}} &= \lambda p.\text{match } p \text{ as } \langle x, y, z\rangle \text{ in } z \geqslant 15 \\ \mathscr{P}_{\text{D}} &= \lambda p.\text{match } p \text{ as } \langle x, y, z\rangle \text{ in } x \geqslant 20 \wedge y < 10 \wedge z \geqslant 15 \end{aligned}$$

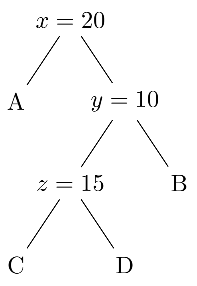


Figure 2.2: kd-tree fragment.

We extend spatial notions like containment and disjointness to our description of predicates and introduce some associated notation:

- **Containment**. If predicate P_2 implies predicate P_1, we say that P_1 contains P_2:

$$P_1 \supseteq P_2$$

- **Disjointness**. If predicates P_1 and P_2 cannot both be satisfied, *i.e.* $P_1 \wedge P_2 \leftrightarrow \texttt{False}$, we say that they are disjoint:

$$P_1 \parallel P_2$$

- **Non-disjointness**. We write non-disjointness of predicates P_1 and P_2 as follows:

$$P_1 \nparallel P_2$$

  and may also say that P_1 *intersects* or *overlaps* P_2.

2.4.3 Predicates in external access methods

We describe non-leaf, multiway node structures as a list of pairs of local predicates and child page addresses:

$$[\langle P_1 : \mathit{LocalPredicate}, a_1 : \mathit{PageId}\rangle, \ldots, \langle P_n : \mathit{LocalPredicate}, a_n : \mathit{PageId}\rangle]$$

where P_i is the local predicate of the node at address a_i. As in some functional programming languages, we use the notation $\langle\cdots\rangle$ to indicate a tuple of some fixed number of values, of possibly different types, and $[\cdots]$ to indicate a list of objects of a common type.

Because external index structures are stored on disk, the information required to construct the local predicate associated with each of a node's children must first be decoded from some external, byte-encoded representation. Many access methods, for example the R-tree, encode this information as an explicit list of pairs of regions and child page addresses, but this is not always the case; the hB-tree [39], for example, stores region information using an intranode kd-tree. We refer to the process of translating the decoded region representation into a list of $\langle \mathit{LocalPredicate}, \mathit{PageId}\rangle$ pairs as *interpretation*. We make no assumptions about the 'raw' decoded representation, other than that the information stored is sufficient to reconstruct the interpreted node structure.

Given our description, at the conclusion of section 2.4.1, of tree descent as the construction of nodes' global predicates, it might seem more appropriate to describe an interpreted node entry as having the form $\langle \mathit{GlobalPredicate}, \mathit{PageId} \rangle$, but to do so is cumbersome and unnecessary. As described in section 2.4.1, global predicates are constructed implicitly, merely by virtue of selecting a path through more than one node, while local predicates are interpreted when selecting a subtree from within a node. Describing an interpreted node structure consisting of global predicates does not, therefore, reflect the true situation in practice. In fact, the construction of the global predicate as a series of conjunctions strictly limits the local predicate's contribution to the global predicate.

Let $\mathscr{P}_{\mathrm{N}}$ be the global predicate of a node N with internal structure $[\langle P_1, a_1 \rangle, \ldots, \langle P_n, a_n \rangle]$, and $\mathrm{N}_1, \ldots, \mathrm{N}_n$ the nodes addressed by child pointers $a_1, \ldots, a_n$. Because we define a global predicate as the conjunction of local predicates, we can, in general, define a node's global predicate as the conjunction of its local predicate and the global predicate of its parent:

$$\begin{aligned}\mathscr{P}_{\mathrm{N}_1} &= \mathscr{P}_{\mathrm{N}} \wedge P_{\mathrm{N}_1} \\ &\vdots \\ \mathscr{P}_{\mathrm{N}_n} &= \mathscr{P}_{\mathrm{N}} \wedge P_{\mathrm{N}_n}\end{aligned}$$

This has the effect that the difference between $\mathscr{P}_{\mathrm{N}_1}$ and $\mathscr{P}_{\mathrm{N}_n}$ is attributable entirely to the difference between P_{N_1} and P_{N_n}; in particular, disjointness of P_{N_1} and P_{N_n} is sufficient to ensure that $\mathscr{P}_{\mathrm{N}_1}$ and $\mathscr{P}_{\mathrm{N}_n}$ are disjoint.

For very many structures, the 'interpretation' of local predicates amounts to little more than the decoding of their on-disk representation. The R-tree's regions are described directly as rectangles, and we interpret a rectangle R directly to give the predicate $\lambda x.(x \in R)$. In structures such as these, the separation of predicates and their decoded representation may seem unnecessary. There are, however, a number of more complex structures in which further interpretation is required to yield local predicates. A shared feature of the development of the hB-tree and the BANG file [22] (and, we believe, the BV-tree [19]) is that treatment of nodes as collections of low-level data, rather than as higher-level, interpreted predicates, has led to misconceptions and errors. It is for this reason that we introduce the notion of local predicates explicitly.

In section 2.4.5 we provide more detailed examples of local predicate interpretation in the R-tree, the B+ tree and the BANG file.

2.4.4 Diagram conventions

Having introduced our approach to tree structure description, we describe some diagrammatic conventions with the aid of figure 2.3, which shows a small fragment of an example external multiway tree.

We have described an interpreted node structure as a list of $\langle \mathit{LocalPredicate}, \mathit{PageId} \rangle$ pairs. When illustrating a tree structure it will usually be using this interpreted structure; if otherwise this will be made clear in the text. The root node of the tree in figure 2.3 is thus the interpreted node $[\langle \mathrm{A}, a \rangle, \langle \mathrm{B}, b \rangle, \langle \mathrm{C}, c \rangle]$ (assuming a, b and c to be the child node addresses associated with predicates A, B and C respectively). We place the local predicate component of each interpreted entry within the node; the associated node address is represented by the line connecting the entry to its child.

Each block of entries (*e.g.* [A,B,C] or [D,E,F]) represents a non-leaf node; leaf pages will usually be denoted with a small circle (as in the children of entries D, E and F). Nodes may be referred to either as the child of a named entry or by using an explicit node label, placed to the left of the node

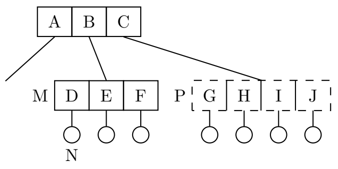


Figure 2.3: Tree diagram conventions.

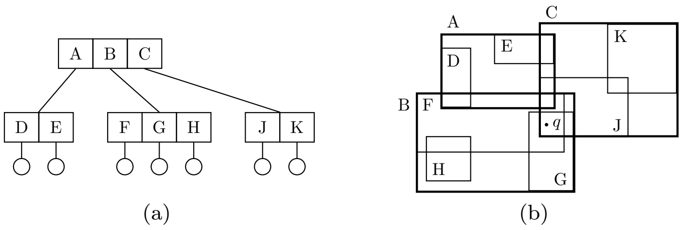


Figure 2.4: Example of a simple R-tree and the two-dimensional data space it indexes.

(or sometimes beneath it, in the case of leaf nodes); for example node M is the child of entry B while node N is the child of D. In section 2.2.1 we described nodes as having a maximum capacity; nodes that exceed that capacity must split, and, while not realised on disk, we represent overflowing nodes as a block of entries bounded by a dashed line. In the figure, node P is overflowing.

Unterminated child pointers from entries in internal nodes (*e.g.* from entry A) indicate that a subtree is present but not relevant to the issue under consideration.

2.4.5 Examples of local predicate interpretation

R-tree

The R-tree has an explicit pairwise entry structure consisting of $\langle R : \mathit{Rectangle}, a : \mathit{PageId}\rangle$ pairs, with each rectangle R_i providing directly the local predicate of the node at address a_i. Evaluating predicate satisfaction consists merely of choosing the correct spatial relationship between a query and the rectangle, R. For an exact-match query point, q, the predicate is $\lambda x.(x \in R)$. For a region query, Q, to have to explore the subtree associated with R, we require $\exists x.\ x \in R \wedge x \in Q$; *i.e.* $R \not\parallel Q$. Given a node containing entries $[\langle R_1, a_1\rangle, \ldots, \langle R_n, a_n\rangle]$ with parent entry $\langle R_p, a_p\rangle$, we have that $\forall i \in \{1, \ldots, n\}, R_p \supseteq R_i$, providing the convenient result that every node's local predicate directly implies (in fact, equals) its global predicate, *i.e.* $\mathscr{P}_{\mathrm{N}} = P_{\mathrm{N}}$. Predicates in the R-tree may or may not be disjoint; if multiple local predicates are satisfied, then each associated subtree must be explored.

An example of an R-tree and the data space it indexes is given in figure 2.4. The point q is contained by rectangles B and C in the root, and in rectangles F, G and J at the leaf level (although it may not appear in all, or indeed any, of the children of F, G and J). A recursive, depth-first search of the tree would first visit the root node, then the child of B and the children of entries F and G, before *backtracking* to the root and descending the children of C and J.

B+ tree

The B+ tree [15] can be viewed as an R-tree optimised for the indexing of one-dimensional spaces. One could imagine permitting overlap between local predicates in a one-dimensional R-tree (comparable to that between the predicates associated with entries A, B and C in figure 2.4), but for

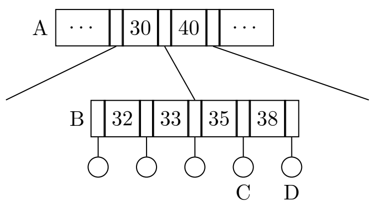


Figure 2.5: Consideration of B+ tree node B in isolation suggests that the interval associated with node E is $[38, +\infty)$, but in the context of node A is correctly calculated to be [38,40).

point data in one dimension this is never necessary. This being so, we observe that, if we insist that the entire data space is represented, the end of one interval (predicate) in the structure coincides with the start of the next, allowing us to store that value only once for both intervals. This optimisation provides us with the familiar B+ tree internal node structure:

$$[a_0 : \mathit{PageId},\ k_1 : \mathit{Key},\ a_1 : \mathit{PageId}, \ldots,\ k_n : \mathit{Key},\ a_n : \mathit{PageId}]$$

where *Key* values are the interval start/end points. This has two consequences:

- As the structure is no longer a list of $\langle I : \mathit{Interval}, a : \mathit{PageId}\rangle$ pairs, a node's local predicate is not dependent on data associated directly with its page pointer, but must be interpreted from either one or two *Key* values.

- Unlike the R-tree, but in common with the kd-tree example discussed in section 2.4.1, some nodes' local predicates are no longer contained in their global predicates.

Local predicates for a node at address a_i are interpreted from surrounding key values as follows:

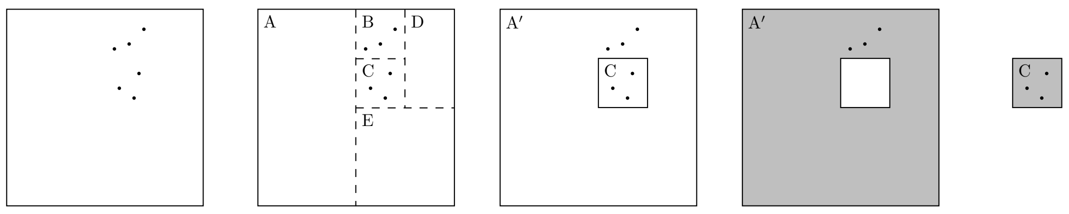
$\boldsymbol{i}$	$\boldsymbol{P}$
$i = 0$	$\lambda x.(x < k_1)$
$0 < i < n$	$\lambda x.(x \geqslant k_i \ \wedge x < k_{i+1})$
$i = n$	$\lambda x.(x \geqslant k_n)$

with the result that the intervals associated with the children of a_0 and a_n can only be interpreted to be $(-\infty, k_1)$ and $[k_n, +\infty)$ respectively. Figure 2.5 shows a B+ tree fragment (in its decoded rather than interpreted representation) in which examination of node B indicates that

$$\begin{aligned} P_{\mathrm{C}} &= \lambda x.35 \leqslant x < 38 \\ &= \lambda x.x \in [35, 38) \end{aligned}$$

Similar consideration for node D gives $P_{\mathrm{D}} = \lambda x.x \in [38, +\infty)$, but D's global predicate is

$$\begin{aligned} \mathscr{P}_{\mathrm{D}} &= \lambda x.x \in [30, 40) \wedge x \in [38, +\infty) \\ &= \lambda x.x \in [38, 40) \end{aligned}$$



(a) Skewed data distribution (b) Space partitioning using DYOP scheme (c) Space partitioning using BANG scheme (d) Exploded BANG scheme decomposition

Figure 2.6: Partitioning of a skewed data set in the DYOP and BANG files.

BANG file

The BANG file [22] is a post-and-grow tree that uses a prescriptive binary partitioning scheme with a cycling splitting dimension, similar to that of the DYOP file [45], but with the important difference that empty partitions need not be represented.

Figure 2.6a illustrates a region containing 6 points. Assuming that this region is represented on disk by a single page, and that the maximum page capacity is 5 points, the set of points must be split into two. The DYOP partitioning scheme produces the partitioning shown in figure 2.6b; note that in partitioning the data points, three empty regions have been created, those labelled A, D and E. The critical boundary that separates the points as required is that between regions B and C, but the strict partitioning scheme requires the creation of the other regions before that boundary can be created. In the DYOP file, each of these regions is represented explicitly, even if empty.

The BANG file cycles through the same partitioning scheme to produce region C, but omits to store empty regions created in the process, leaving the decomposition shown in figure 2.6c. This amounts to the removal of a ‘core’ (region C) from the region, leaving a ‘doughnut’ (region A′); these two regions are shown, separated, in 2.6d. We will use the set difference symbol, $\setminus$, to describe the removal of holes from a region, *i.e.* $\mathrm{A}' = \mathrm{A} \setminus \mathrm{C}$. Further holes might be removed from region A′ in subsequent splits of the region, producing local predicate descriptions that consist of an outer boundary and a potentially long list of holes.

The immediate question arising from this is how a holey region is to be represented on disk, since explicit recording of the geometry of A′ would become increasingly expensive as more holes were made therein. Note, however, that (in a way similar to that of the B+ tree intervals’ coincident end points) the outer boundary of each core region coincides with the boundary of the hole it leaves behind, so the local predicate corresponding to a holey region can be reconstructed using the descriptions of the region’s outer boundary, and the outer boundaries of any hole regions removed from it. Decoded BANG file node entries thus have the structure $\langle R : \mathit{RegionOuterBoundary}, a : \mathit{PageId} \rangle$, requiring the local predicate of the node at page address a to be interpreted from R and between 0 and $n-1$ other boundaries (where n is the number of entries in its parent node). Figure 2.7a shows a region partitioned into five subregions; the holey regions are given explicitly in the exploded decomposition in figure 2.7b. Figure 2.7c shows the entries representing those regions occupying a node; for node N (the child of entry F), we have $P_{\mathrm{N}} = \lambda x.(x \in \mathrm{F} \setminus (\mathrm{G} \cup \mathrm{H} \cup \mathrm{J}))$. (It is also true that $P_{\mathrm{N}}(x) \rightarrow x \notin \mathrm{K}$, but we need not state this explicitly since $\mathrm{K} \subset \mathrm{J}$).

In figure 2.7, and in later diagrams of BANG file type holey decompositions (such as in section 3.3.5, section 4.2.1 and chapters 6 and beyond), we draw holes as being clearly contained within their outer boundaries. It will be apparent, however, from the mode of partition formation,

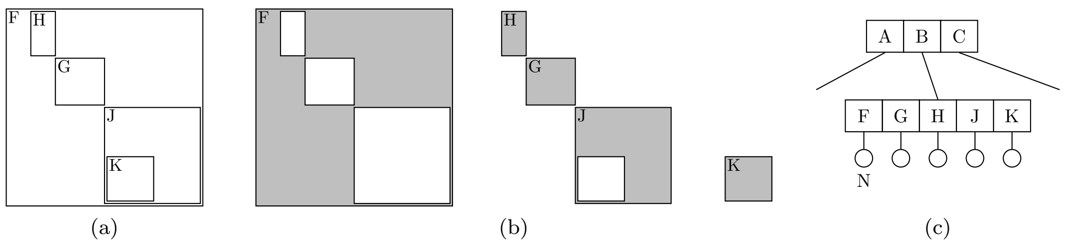


Figure 2.7: A BANG file style holey region decomposition and the associated tree fragment.

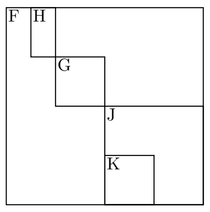


Figure 2.8: Depiction of BANG file type decompositions with coincident boundaries may cloud the detail of containment features.

that holes' boundaries often coincide partially with those of their containers. The spatial decomposition of figure 2.7a might therefore be more accurately represented as shown in figure 2.8, but, for clarity, we will continue to illustrate containment of this kind as in figure 2.7a.

2.5 An abstract state machine for algorithm specification

We will describe algorithms (here for query and insertion operations) using transitions between abstract machine [34] states. An individual state is described as a *configuration*, and an operation consists of a series of transformations between configurations. The exact sequence of transformations performed during any given operation is determined by an ordered set of *rules*, each specified as an input and output configuration, with associated local definitions and conditions for its execution. Collectively, the set of rules used to determine transformations during an operation specifies the algorithm for that operation.

We introduce the syntax in sections 2.5.1–2.5.4, leaving a discussion of the motivation for this approach until section 2.5.6.

2.5.1 Configurations

An abstract machine state is specified using a *configuration*, which consists of a *command* (including a label and zero or more *command parameters*) followed by zero or more *operational parameters*:

$$\Big\langle \texttt{CmdLabel}\,(\mathit{CmdArg},\ \ldots)\,,\ \mathit{OpArg1}\,,\ \ldots \Big\rangle$$

The command label, `CmdLabel`, indicates a subset of rules that could possibly be applied to make the next transition from this configuration; a configuration with a given command label can only be rewritten by a rule with that command label in its input configuration. Operational parameters provide resources that are required throughout an operation, for example in-memory data structures like stacks. Information required by a command that is not part of the operational state is encoded in the command's parameters; an example would be a new entry undergoing insertion into a tree, and which is no longer required in the configuration after being written into a leaf (although further transitions may be required to handle node overflow). Each of an operation's configurations thus has the same structure: a command followed by the same number of consistently-typed operational parameters.

Commands, and tree nodes, are specified as terms using a BNF grammar extended with sequence constructors: thus $[X]$ means a sequence of terms of type X. Tuple types are specified as a list of components within angle brackets, for example $\langle a, b, c\rangle$, but are distinguishable from configurations by their context: a configuration may only appear in a rule as its sole input or output.

2.5.2 Operations

An operation is described in full by an ordered set of conditional transition rules on configurations. Rules are written in the form:

$$\begin{aligned}
&\textit{input configuration} \leadsto \textit{output configuration}\\
&\qquad \text{if } \textit{condition1} \wedge \textit{condition2} \ldots\\
&\qquad \text{where } \textit{definition1}\\
&\qquad \text{and } \textit{definition2}\\
&\qquad \text{and } \ldots
\end{aligned}$$

Definitions are of the form '*LHS* = *RHS*', and bind the variables on the left hand side to the values computed by the expression on the right. In the rules for the transition relation $\leadsto$, the distinction between *conditions* and *definitions* is this: conditions must be tested before a transition can be triggered, but only those definitions used in the input configuration or in the condition being tested should be evaluated before the condition test succeeds. The delayed evaluation of 'where' clauses is similar to the use of `let` in functional programming languages, but the variables defined by 'where' have wider scope than an individual expression, so can be reused by other definitions and in the output configuration. In addition to the explicit conditions and definitions in 'if' and 'where' clauses, the input configuration may encapsulate instances of both. For example, [] in an input configuration is regarded as a test for an empty list, while $[a]$ specifies a list containing a single value, bound to the variable a.

The rules are ordered so that if multiple combinations of input configuration and rule conditions are found to match a machine state, only the first is triggered. When satisfaction of a condition is required only for the first rule in a group, ordering permits omission of an explicit negation of the condition in subsequent rules; for example, in the following rule pair we can omit the clause "if $a \neq 1$" from the second rule, because the rule can only be considered if the condition "if $a = 1$"

is not met in the first:

$$\begin{aligned}
&\big\langle \texttt{CmdI}\,(a)\,,\, b\,,\, c\big\rangle \leadsto \big\langle \texttt{CmdF}\,,\, d\,,\, c\big\rangle\\
&\qquad \text{if } a = 1\\
&\qquad \text{where } d = f(b)\\
&\big\langle \texttt{CmdI}\,(a)\,,\, b\,,\, c\big\rangle \leadsto \big\langle \texttt{CmdF}\,,\, b\,,\, d\big\rangle\\
&\qquad \text{if } a \neq 1 \quad \textbf{[this clause is omitted]}\\
&\qquad \text{where } d = f(c)
\end{aligned}$$

Note that the specification of 'if' clauses may include implicit definitions. If this is the case, although the ordering of rules may permit an 'if' clause to be omitted, the definitions within the omitted clause may require explicit restatement in an appropriate 'where' clause:

$$\begin{aligned}
&\big\langle \texttt{CmdI}\,(a)\,,\, \langle p, q\rangle\,,\, c\big\rangle \leadsto \big\langle \texttt{CmdF}\,,\, \langle p, q\rangle\,,\, c\big\rangle\\
&\qquad \text{if } g(c) \neq \langle r, s\rangle\\
&\big\langle \texttt{CmdI}\,(a)\,,\, \langle p, q\rangle\,,\, c\big\rangle \leadsto \big\langle \texttt{CmdF}\,,\, \langle r, s\rangle\,,\, c\big\rangle\\
&\qquad \text{where } \langle r, s\rangle = g(c)
\end{aligned}$$

(This example is rather contrived in that, in this case, the need to restate the definition in the second rule would be avoided by reversing the rule ordering). We describe tuples structurally, as in the above example, rather than by naming their components, but when a tuple variable is not required for a definition we may denote it with an underscore, _. This is not a variable name, but rather indicates the presence of a variable whose name we have omitted, because we will not be making reference to it. In the above example, had we been interested solely in the s component of the result of $g(c)$, we might have written:

$$\begin{gathered}
\vdots\\
\text{where } \langle \_, s\rangle = g(c)
\end{gathered}$$

We indicate, in the same way, cases in which variables in input configurations are not used in the body of a given state transition.

Initialisation (or 'bootstrapping') and termination of operations involves two further configurations, not of the structure specified for the operation, but consisting of a single command and no operational parameters. An initial configuration encapsulates the resources passed into the operation from the external environment and undergoes a transition initialising the first operational configuration. Similarly, an operational configuration that satisfies some terminating condition undergoes a transition into a terminal configuration. For example:

$$\left\langle \texttt{init}\,(p, q) \right\rangle \rightsquigarrow \left\langle \texttt{CmdI}\,(p)\,,\, q\,,\, [\,] \right\rangle$$

$$\left\langle \texttt{CmdF}\,,\, \langle r, s\rangle\,,\, c \right\rangle \rightsquigarrow \left\langle \texttt{term}\,(c) \right\rangle \quad \text{if } r = s$$

(The actual commands in the initial and terminal configurations, in this case `init` and `term`, are specific to each operation).

2.5.3 The store

A particular feature of our abstract machine approach is our use of a store, σ, to capture destructive in-place updates on disk pages. The store is usually specified as the final operational parameter of a configuration, and is defined formally as a finite partial map from page identifiers (disk addresses) to nodes; $\sigma : \mathit{PageId} \rightarrow \mathit{Node}$, where *Node* is the type 'external tree node'. For a store σ, we define the domain of σ to be the set of all page addresses of nodes in the store:

$$\operatorname{dom}\sigma \;::=\; \{\; p \mid \exists n \in \mathit{Node}.\; p \mapsto n \in \sigma \,\}$$

and the store update notation $\sigma[p \mapsto v]$ to mean

$$\{q \mapsto n \mid q \mapsto n \in \sigma \wedge q \neq p\} \cup \{p \mapsto v\}$$

We further define $\sigma[p_0 \mapsto v_0, p_1 \mapsto v_1, \ldots]$ to mean $\sigma[p_0 \mapsto v_0][p_1 \mapsto v_1]\ldots$. New, unallocated page addresses are obtained from the store through the function $\texttt{fresh}(\sigma)$. Finally, to accommodate release of a location r in a store σ, where $r \in \operatorname{dom}\sigma$, we define $\sigma \setminus r$ to mean the restriction of σ with r removed, $\sigma\,|_{\operatorname{dom}\sigma\setminus\{r\}}$.

The store is an abstraction of an external storage device of some kind, and for the purposes of operations that take place within a database management system may be considered to be a buffer pool. Notice that the detail of decoding from and encoding into a node's external, byte-encoded representation is hidden inside the store, however (for now) the detail of predicate interpretation is not.

2.5.4 Ancillary definitions

A number of ancillary definitions and notations are presented below. Most of these are fairly standard functions for sequence manipulation, but, for clarity, we make them precise here. We use $\mathbf{s}_i$ to mean "the ith element of sequence $\mathbf{s}$"; the index of the first element in a sequence is 1. For sequences $\mathbf{s}$ and $\mathbf{t}$, element a, and for $1 \leqslant i, j \leqslant |\mathbf{s}|$ and predicate P, we define:

$$\begin{aligned}
\operatorname{dom}\mathbf{s} &::= \{i \in \mathbb{N} \mid 1 \leqslant i \leqslant |\mathbf{s}|\} \\
\operatorname{elems}\mathbf{s} &::= \{\mathbf{s}_i \mid i \in \operatorname{dom}\mathbf{s}\} \\
\mathbf{s} \oplus \mathbf{t} &::= \begin{cases} |\mathbf{s}| = 0 & \mathbf{t} \\ |\mathbf{t}| = 0 & \mathbf{s} \\ |\mathbf{s}| \neq 0 \wedge |\mathbf{t}| \neq 0 & [\mathbf{s}_1, \ldots, \mathbf{s}_{|\mathbf{s}|}, \mathbf{t}_1, \ldots, \mathbf{t}_{|\mathbf{t}|}] \end{cases} \\
a :: \mathbf{t} &::= [a] \oplus \mathbf{t} \\
\mathbf{s}_{i \cdot\cdot j} &::= \begin{cases} i > j & [] \\ i \leqslant j & [\mathbf{s}_i, \mathbf{s}_{i+1}, \ldots, \mathbf{s}_j] \end{cases} \\
\texttt{ins}\,(a, i, \mathbf{s}) &::= \mathbf{s}_{1 \cdot\cdot i-1} \oplus [a] \oplus \mathbf{s}_{i \cdot\cdot |\mathbf{s}|} \\
\texttt{del}\,(i, \mathbf{s}) &::= \mathbf{s}_{1 \cdot\cdot i-1} \oplus \mathbf{s}_{i+1 \cdot\cdot |\mathbf{s}|} \\
\texttt{repl}\,(\mathbf{t}, i, \mathbf{s}) &::= \mathbf{s}_{1 \cdot\cdot i-1} \oplus \mathbf{t} \oplus \mathbf{s}_{i+1 \cdot\cdot |\mathbf{s}|} \\
\texttt{append}\,(a, \mathbf{s}) &::= \mathbf{s} \oplus [a] \\
\texttt{first}\,(\mathbf{s}, P) &::= \begin{cases} \nexists x \in \operatorname{elems}\mathbf{s}.\ P(x) & |\mathbf{s}| + 1 \\ \exists x \in \operatorname{elems}\mathbf{s}.\ P(x) & \min\{i \in \operatorname{dom}\mathbf{s} \mid P(\mathbf{s}_i)\} \end{cases} \\
\texttt{uniq}\,(\mathbf{s}, P) &::= \imath\, x \in \operatorname{elems}\mathbf{s}.\ P(x)
\end{aligned}$$

The definition of `first` is such that it has two purposes, the first of which is existential quantification:

$$\exists x \in \operatorname{elems}\mathbf{s}.\ P(x) \leftrightarrow \texttt{first}\,(\mathbf{s}, P) \leqslant |\mathbf{s}|$$

If existential quantification holds, `first` returns the index in the sequence of the first element therein that satisfies P. The operator `uniq` returns the unique element in the sequence that satisfies P; here we use the definite quantifier, $\imath$, which denotes 'the unique x...'. If $\exists! x \in \operatorname{elems}\mathbf{s}.\ P(x)$, then $\texttt{uniq}\,(\mathbf{s}, P) = \mathbf{s}_i$, where $i = \texttt{first}\,(\mathbf{s}, P)$; `uniq` is otherwise undefined.

2.5.5 Example: The B+ tree

In this section we describe the B+ tree insertion algorithm using our abstract machine formalism. Our choice of B+ tree for this example has been made for clarity; the structure itself is so well-understood that we can use the example to provide an uncluttered illustration of our approach. For simplicity, we specify node occupancy limits in terms of number of entries, rather than explicitly by their physical size, and will continue to do so throughout the thesis. Node occupancy limits are specified using the constants Max_L, which corresponds to the maximum number of entries permitted in a leaf page, and Max_I, which corresponds to the maximum number of entries (*i.e.* the maximum number of children) in an internal page.

The grammar for a node of a B+ tree is as follows:

$$\begin{aligned}
\mathit{Node} &::= \mathit{INode} \mid \mathit{LNode} \\
\mathit{INode} &::= \texttt{I}\,([\mathit{Key}], [\mathit{PageID}]) \\
\mathit{LNode} &::= \texttt{L}\,([\langle \mathit{Key}, \mathit{Value} \rangle], \mathit{PageID})
\end{aligned}$$

The intention is that an *INode* represents an internal node of the tree, implemented as a disk page containing a sequence of keys and a sequence of child page pointers. For convenience, we describe the node as containing two separate lists, to simplify issues of type presented by a single list of interleaved page pointers and keys. An *LNode* corresponds to a leaf node; a disk page containing entries of the form $\langle \mathit{Key}, \mathit{Value}\rangle$ as described earlier, and, in this B+ tree variant, a forward pointer to the next leaf page. Leaf forward pointers join the tree's leaf pages into a singly linked list, permitting forward range searching along the leaf level. A newly-created B+ tree is represented by a pair $\langle r, \sigma\rangle$, where r is the disk address of the root of the tree and σ is a store that maps r to a leaf node containing an empty sequence of entries and a null forward pointer. The store representing a new B+ tree thus has the form:

$$\{r \mapsto \mathtt{L}\,([\,], \mathtt{null})\}$$

B+ tree insertion configurations are tuples of the form:

$$\left\langle \mathtt{C}\,,\, r\,,\, \pi\,,\, \sigma \right\rangle$$

where $\mathtt{C}$ is a command, r is a page identifier, π is a stack of page identifiers and σ is a page store. r and π are used to record the path taken down the tree to permit us to step back up it. The initial configuration for insertion of an entry a into some B+ tree $\langle r, \sigma\rangle$ is $\left\langle \mathtt{btInsert}\,(a, r, \sigma)\right\rangle$, where r is the page identifier of the root page of the B+ tree. A transition into the terminal configuration $\left\langle \mathtt{bt}\,(r', \sigma')\right\rangle$ occurs when insertion is complete; the resulting B+ tree is $\langle r', \sigma'\rangle$. The grammar for the B+ tree insertion command terms is as follows:

$$\mathit{btreeInsertCommand} \ ::= \ \mathtt{ins}\,(\mathit{Entry}) \mid \mathtt{S} \mid \mathtt{D}\,(\mathit{Key}, \mathit{PageID})$$

The `ins` command is used during the descent of a B+ tree during entry insertion. If insertion causes overflow, requiring a node to be split and entries to be promoted, those entries are carried in a `D` command; return from insertion without overflow is described using the `S` command.

We described in section 2.2.5 the use of a split policy in non-SPP structures, but we can also use the term to describe the way in which splits are chosen in B+ tree nodes. We specify this here with two ancillary definitions that define the policy for splitting the contents of leaf and internal nodes when such a split in necessary. When an entry a is to be inserted into a sequence of entries $\mathbf{e}$, and $|\mathbf{e}| = \mathit{Max}_L$, the resulting list is split into two using $\mathtt{splitL}\,(i, a, \mathbf{e})$ (where i is the position in $\mathbf{e}$ into which a is inserted). The components of the result are $\langle \mathbf{e}', k, \mathbf{e}''\rangle$, and have the following properties:

$\mathbf{e}' \oplus \mathbf{e}'' = \mathtt{ins}\,(a, i, \mathbf{e})$	Insertion of entry a into sequence $\mathbf{e}$ at position i, then splitting the resulting sequence, yields subsequences $\mathbf{e}'$ and $\mathbf{e}''$.
$\langle k, \_\rangle = \mathbf{e}''_1$	k is the *Key* component of the $\langle \mathit{Key}, \mathit{Value}\rangle$ pair at position 1 of sequence $\mathbf{e}''$. This value will be posted into the parent node.
$\lvert\lvert\mathbf{e}'\rvert - \lvert\mathbf{e}''\rvert\rvert \leqslant 1$	The split of the list resulting from $\mathtt{ins}\,(a, i, \mathbf{e})$ is optimally balanced; the lengths of the resulting subsequences differ, at most, by 1.

When a `D` command is used to post a key value, k, and page identifier, q, into the paired key and page identifier sequences $\langle \mathbf{d}, \mathbf{p} \rangle$ of a full internal node, then that node must split. This is handled using $\texttt{splitI}\,(i, k, q, \mathbf{d}, \mathbf{p})$, in which i indicates the positions in the node's respective key and page address sequences at which k and q are to be inserted. The result of `splitI` is $\langle \mathbf{d}', \mathbf{p}', k', \mathbf{d}'', \mathbf{p}'' \rangle$; $\mathbf{d}'$ and $\mathbf{p}'$ form the list of keys and pointers for the 'left-hand' node resulting from the split, $\mathbf{d}''$ and $\mathbf{p}''$ form those for the right-hand node, and key value k' provides a discriminator between the two nodes.

Prior to incorporating k and q into the full internal node, we have $|\mathbf{p}| = |\mathbf{d}| + 1$; the components of the result of `splitI` satisfy the conditions:

$\mathbf{d}' \oplus [k'] \oplus \mathbf{d}'' = \texttt{ins}\,(k, i, \mathbf{d})$	In the sequence resulting from the insertion of key value k into sequence $\mathbf{d}$ at position i, the subsequences to either side of key value k' are $\mathbf{d}'$ and $\mathbf{d}''$. k' will be posted into the parent node.
$\mathbf{p}' \oplus \mathbf{p}'' = \texttt{ins}\,(q, i+1, \mathbf{p})$	Insertion of page pointer q into sequence $\mathbf{p}$ at position $i+1$, then splitting the resulting sequence, yields subsequences $\mathbf{p}'$ and $\mathbf{p}''$.
$\lvert\mathbf{p}'\rvert = \lvert\mathbf{d}'\rvert + 1$	The list of page pointers to be stored in the left-hand node is one element longer than its list of key values, as was the case in the original node.
$\lvert\mathbf{p}''\rvert = \lvert\mathbf{d}''\rvert + 1$	The list of page pointers to be stored in the right-hand node is also one element longer than its list of key values.
$\lvert\lvert\mathbf{p}'\rvert - \lvert\mathbf{p}''\rvert\rvert \leqslant 1$	The split of the list resulting from $\texttt{ins}\,(q, i+1, \mathbf{p})$ is optimally balanced; the lengths of the resulting subsequences differ, at most, by 1.

The abstract machine transition rules for B+ tree insertion are described in figure 2.9. They are split into five groups and each rule is numbered for reference. The first group, containing single transition 1.1, initialises the insertion from the external command `btInsert`. The second group also contains only one rule, 1.2, for descending down the correct path of internal nodes in the tree, while pushing path location information onto the stack at each step. The third section describes the three cases that can occur when a leaf page is encountered:

- Transition 1.3: The entry to be inserted has the same key value as an existing entry in the node and so replaces the existing entry. No further changes to the tree are necessary because a single node result has occurred.

- Transition 1.4: The entry's key value is not found in the node, but the node is not full, so the entry is added. Again, no further changes to the tree are required because the insertion has resulted in a single node.

- Transition 1.5: The leaf node is full and must be split between the original node and a new one. The forward pointer of the original leaf is updated to point to the new leaf and, as a double node results, a new key value and page pointer must be posted into the parent level.

$$\left\langle \texttt{btInsert}\,(\langle k,v\rangle,r,\sigma)\right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{ins}\,(\langle k,v\rangle)\,,\,r\,,\,[\,]\,,\,\sigma\right\rangle \tag{1.1}$$

$$\left\langle \texttt{ins}\,(\langle k,v\rangle)\,,\,r\,,\,\pi\,,\,\sigma\right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{ins}\,(\langle k,v\rangle)\,,\,\mathbf{p}_i\,,\,r::\pi\,,\,\sigma\right\rangle \tag{1.2}$$

if $\sigma(r) = \texttt{I}\,(\mathbf{d},\mathbf{p})$

where $i = \texttt{first}\,(\mathbf{d},P)$

and $P = \lambda x.x > k$

$$\left\langle \texttt{ins}\,(\langle k,v\rangle)\,,\,r\,,\,\pi\,,\,\sigma\right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{S}\,,\,r\,,\,\pi\,,\,\sigma[r\mapsto \texttt{L}\,(\texttt{repl}\,(\langle k,v\rangle,i,\mathbf{e})\,,f)]\right\rangle \tag{1.3}$$

if $i \leqslant |\mathbf{e}|$

where $\sigma(r) = \texttt{L}\,(\mathbf{e},f)$

and $i = \texttt{first}\,(\mathbf{d},P)$

and $P = \lambda x.\text{match } x \text{ as } \langle j,w\rangle \text{ in } j = k$

$$\left\langle \texttt{ins}\,(\langle k,v\rangle)\,,\,r\,,\,\pi\,,\,\sigma\right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{S}\,,\,r\,,\,\pi\,,\,\sigma[r\mapsto \texttt{L}\,(\texttt{ins}\,(\langle k,v\rangle,i,\mathbf{e})\,,f)]\right\rangle \tag{1.4}$$

if $|\mathbf{e}| < \mathit{Max}_L$

where $\sigma(r) = \texttt{L}\,(\mathbf{e},f)$

and $i = \texttt{first}\,(\mathbf{d},P)$

and $P = \lambda x.\text{match } x \text{ as } \langle j,w\rangle \text{ in } j > k$

$$\left\langle \texttt{ins}\,(\langle k,v\rangle)\,,\,r\,,\,\pi\,,\,\sigma\right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{D}\,(d,t)\,,\,r\,,\,\pi\,,\,\sigma[r\mapsto \texttt{L}\,(\mathbf{e}',t)\,,t\mapsto \texttt{L}\,(\mathbf{e}'',f)]\right\rangle \tag{1.5}$$

where $\sigma(r) = \texttt{L}\,(\mathbf{e},f)$

and $i = \texttt{first}\,(\mathbf{d},P)$

and $P = \lambda x.\text{match } x \text{ as } \langle j,w\rangle \text{ in } j > k$

and $\langle \mathbf{e}',d,\mathbf{e}''\rangle = \texttt{splitL}\,(i,\langle k,v\rangle,\mathbf{e})$

and $t = \texttt{fresh}(\sigma)$

$$\left\langle \texttt{S}\,,\,r\,,\,s::\pi\,,\,\sigma\right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{S}\,,\,s\,,\,\pi\,,\,\sigma\right\rangle \tag{1.6}$$

$$\left\langle \texttt{D}\,(d,p)\,,\,r\,,\,s::\pi\,,\,\sigma\right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{S}\,,\,s\,,\,\pi\,,\,\sigma[s\mapsto \texttt{I}\,(\texttt{ins}\,(d,i,\mathbf{d})\,,\texttt{ins}\,(p,i+1,\mathbf{p}))]\right\rangle \tag{1.7}$$

if $|\mathbf{p}| < \mathit{Max}_I$

where $\sigma(s) = \texttt{I}\,(\mathbf{d},\mathbf{p})$

$$\left\langle \texttt{D}\,(d,q)\,,\,r\,,\,s::\pi\,,\,\sigma\right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{D}\,(d',t)\,,\,s\,,\,\pi\,,\,\sigma[s\mapsto \texttt{I}\,(\mathbf{d}',\mathbf{p}')\,,t\mapsto \texttt{I}\,(\mathbf{d}'',\mathbf{p}'')]\right\rangle \tag{1.8}$$

where $\sigma(s) = \texttt{I}\,(\mathbf{d},\mathbf{p})$

where $i = \texttt{first}\,(\mathbf{d},P)$

and $P = \lambda x.x > d$

and $\langle \mathbf{d}',\mathbf{p}',d',\mathbf{d}'',\mathbf{p}''\rangle = \texttt{splitI}\,(i,d,q,\mathbf{d},\mathbf{p})$

and $t = \texttt{fresh}(\sigma)$

$$\left\langle \texttt{S}\,,\,r\,,\,[\,]\,,\,\sigma\right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{bt}\,(r,\sigma)\right\rangle \tag{1.9}$$

$$\left\langle \texttt{D}\,(d,s)\,,\,r\,,\,[\,]\,,\,\sigma\right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{bt}\,(t,\sigma[t\mapsto \texttt{I}\,([d],[r,s])])\right\rangle \tag{1.10}$$

where $t = \texttt{fresh}(\sigma)$

Figure 2.9: B+ tree insertion rules.

The fourth section describes the possible ripple of post operations up the tree. Again there are three cases, each popping the parent location off the stack:

- Transition 1.6: The result from the level below was a single node, so there is no change necessary in this level. We pass on a single node result to the level above.

- Transition 1.7: The result from below is a double node, so we must insert new entries in this level. The node can accommodate the new entries, so we return a single page result to the level above. Note here that the natural key ordering is used to calculate the position into which the posted discriminator value is to be inserted and to impose an ordering on the list of page identifiers, maintaining the association between discriminators and child nodes.

- Transition 1.8: The result from below is a double node, and there is insufficient space in this node to insert the new key value and page pointer. We must split this node and return a double node result to the level above.

The final section specifies the behaviour when the upward rippling finds the stack to be empty. At this point the system is trying to return a result from the root page level. Either the root has not been split, in which case the root of the new tree is the same as that of the old, or the root page has been split, in which case a new root page must be constructed and made to point to the two sub-trees.

2.5.6 Motivation

The purpose of the abstract machine approach to algorithm specification is to permit the construction of rules at a higher level of abstraction than would be the case with (pseudo)code representations. A rule is typically a small step in the execution of an operation, intuitively on the scale of “choose a subtree for insertion and descend one level in the tree”, but still contains sufficient formal detail to enable implementation. This level of abstraction is consistent with that used by designers of access methods when considering structures.

Elements of the approach, for example the pattern-matching required to test commands and arguments in input configurations, or the use of first-class functions, are features available in functional programming languages. However, our requirement for a store, to capture in-place destructive updates, and to allow referencing of nodes by page identifiers, makes a purely functional language inappropriate for rule implementations. We will find referencing by node address convenient in various places, for example section 3.2.1, and essential in section 5.4.

The combined functional and object-orientated capabilities of other languages like OCaml offer an alternative implementation language, indeed figure 2.10 gives an implementation of the B+ tree rules. Our abstract machine provides a very small subset of such a language’s functionality, constraining us to describe operations in a stepwise, recursion-free approach. This ‘syntactic salt’ deliberately restricts our specification of algorithms to the desired level of abstraction, ensuring that the state of the operation remains clear throughout; in chapter 5 we make use of this feature to provide examples of tree operations in progress. Conversely, the implementation of figure 2.10 makes a number of optimisations that, while appropriate for efficient code, are already deeply nested in a way that the abstract machine language explicitly prevents.

```
let exec_step (c:config) = match c with
  | (Insert(a), r, pi, sigma) ->
    (
      match getStore sigma r with
        | I(d, p) ->
          let i = first d (function x -> x > a.k) in
          let p_i = get p i in
            (Insert(a), p_i, (r,i) ::  pi, sigma)
        | L(e,f) ->
          let i = first e (function x -> x.k >= a.k) in
            if test i e (function x -> x.k = a.k) then
              (S, r, pi, saveStore (L(replace a i e, f)) r sigma)
            else if List.length e < max_L then
              (S, r, pi, saveStore (L(ins a i e, f)) r sigma)
            else
              let (ep, k, epp) = splitL i a e in
              let q = fresh sigma in
                (D(k,q), r, pi, saveStore (L(epp, f)) q (saveStore (L(ep, q)) r sigma))
    )
  | (S, r, (t, i)::pi, sigma) ->
    (S, t, pi, sigma)
  | (D(k,q), r, (t, i)::pi, sigma) ->
    (
      match getStore sigma t with
        | L(_,_) -> raise (Btree_store_corrupted r)
        | I(d, p) ->
          if List.length p < max_I then
            (S, t, pi, saveStore (I(ins k i d, ins q (i+1) p)) t sigma)
          else
            let qp = fresh sigma in
            let (dp, pp, kp, dpp, ppp) = splitI i k q d p in
              (D(kp,qp), t, pi, saveStore (I(dpp, ppp)) qp (saveStore (I(dp, pp)) t sigma))
    )
  | (S, r, [], sigma) ->
    (Ret, r, [], sigma)
  | (D(k,t), r, [], sigma) ->
    let q = fresh sigma in
      (Ret, q, [], saveStore (I([k], [r; t])) q sigma)
  | (Ret, q, [], sigma) ->
    print_btree r=q; s=sigma;
    raise Btree_insert_finished
  | (com, r, pi, sigma) ->
    (ERR(com), r, pi, sigma)
```

Figure 2.10: OCaml B+ tree insertion implementation.

2.6 Summary

In this chapter we began by setting out the scope of our discussion, describing a class of access methods with a common structural theme: external (and dynamic), hierarchical, multidimensional point access methods. We furthermore described their performance as being contingent on four additional factors or ‘desirable properties’; locality preservation, minimum fanout, IO-balance and the single path property.

In section 2.4 we introduced a predicate-based description of tree structures, noting that predicates may require further interpretation of node structures decoded from their on-disk representation. In section 2.5 we described our abstract machine approach to algorithm specification. We will use these tools in our review of previous work, presented in chapter 3, and in which we classify common access methods of the class defined here according to their provision of the four desirable properties.

Chapter 3

Analysis of previous work

In section 2.2, we described four properties that are desirable in a hierarchical multidimensional access method:

- IO-balance;
- the single path property (SPP);
- guaranteed minimum fanout ratio;
- locality preservation.

We remarked that the goal of designing a single structure possessing all of these properties has proved to be elusive, but a great many structures provide three of the four. This permits us to develop a classification of four groups of access methods, members of each group having in common the property that they lack. In this chapter, we review a number of access methods in the literature, of the class of structure in which we are interested (as characterised in section 2.1), placing each into one of these four groups.

3.1 Locality-neglectful structures

For efficient execution of queries with spatial extent, we require structures that reproduce the closeness of points to one another in space as close proximity within the structure. We refer to such structures as *locality preserving*; the structures presented later in this chapter are of this type, but locality preservation is normally only achieved at the expense of the single path property, IO-balance or of a guaranteed minimum fanout ratio. The B+ tree is a special case that achieves all of the latter, and preserves locality, in consequence of its indexing only one-dimensional spaces. In this section we describe an approach to multidimensional indexing by mapping multidimensional spaces into one dimension, relying on the mapping to project both the set of points and their spatial relationships into the target space. Other desirable index structure characteristics are then provided by the B+ tree, which is used to index the mapped space.

Recall that locality preservation is only useful in contexts where locality has meaning. In an environment requiring support solely for exact match queries, or in spaces of sufficiently high dimensionality to render measures of distance non-discriminatory (see section 2.3), methods such as those described here could be used to accelerate exact match queries while supporting queries with spatial extent via sequential scan of the B+ tree leaf level.

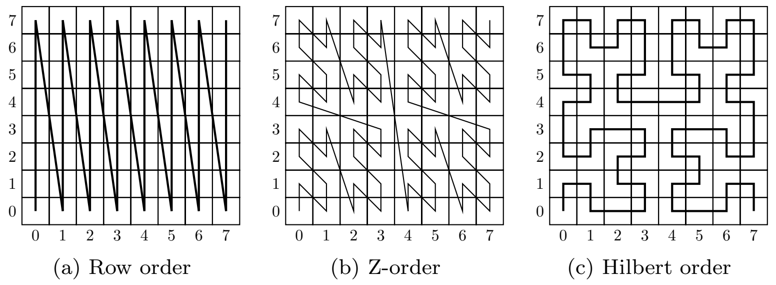


Figure 3.1: Space-filling curves.

3.1.1 Space-filling curves

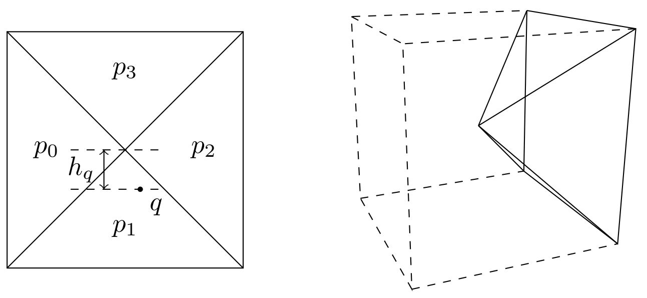
Consider a list of binary tuples, $\langle x, y \rangle$, sorted on attribute x then y, which can then be indexed very efficiently using a standard one-dimensional access method, for example a B+ tree. Figure 3.1a overlays this sort order on the (x, y) space, illustrating that although the sort order preserves very well the locality between x values, that between positions in the (x, y) space is not preserved well at all. For example, while cells (0,0) and (0,1) are close in space and on the curve, cells (0,0) and (1,0) are close in space but widely separated on the curve. Other mappings from n to one-dimensional space have been designed to produce better locality-preserving *space-filling curves* (a two-dimensional Z-ordering [44] and Hilbert [17] curve are shown in figures 3.1b and 3.1c), but even these tend to produce extended regions of preserved locality separated by ever-less local jumps; for example those between cells (3,0) and (4,0) in both figures 3.1b and 3.1c. Bayer's UB-tree [2] uses a B+ tree to index points and regions [43] in Z-ordered cells.

3.1.2 'Pyramid' mappings

Pyramid technique

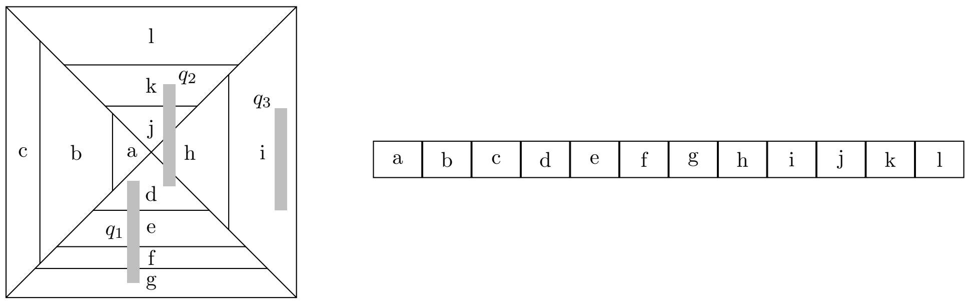
The pyramid technique [7] maps points in a d-dimensional unit hypercube into a one dimensional space by grouping points on the basis of the face of the hypercube to which they are closest. This divides the space into $2.d$ 'pyramids', the base of each of which is formed by one of the $(d-1)$-dimensional faces of the hypercube. Figure 3.2a shows the division of the two-dimensional space into four pyramids, while the extension of a single three-dimensional pyramid into the space is shown in figure Figure 3.2b. The pyramids are numbered 0 to $2d-1$, such that the dimension i coordinate of points on the base of pyramid p_i is 0 (where $0 \leqslant i < d$), while that of points on the base of the opposite pyramid, p_{i+d}, is 1. Points within a pyramid are ordered by their height, h, defined as the perpendicular distance from the apex of the pyramid to the point. The numbering scheme and height of a point q are shown in figure 3.2a. Since pyramid numbers are integers and $0 \leqslant h < 0.5$ (for pyramid numbers 0 to $d-1$; $0 \leqslant h \leqslant 0.5$ for numbers d to $2d-1$), a point's *pyramid value*, the sum of its pyramid number and height, maps the point into the interval $[0, 2d-0.5]$, with disjoint subregions $\{[0, 0.5), \ldots, [d-1, d-0.5), [d, d+0.5], \ldots, [2d-1, 2d-0.5]\}$ corresponding to each pyramid. Pyramid values are then indexed using a B+ tree. A pyramid value represents a horizontal slice of the pyramid concerned, and the authors motivate the structure by citing a query model that suggests fewer page accesses are required in this slicewise spatial decomposition.

Figure 3.3 shows an example transformation of a two-dimensional space into the leaf level of the B+ tree. The query rectangles q_1, q_2 and q_3 demonstrate some of the geometric effects of the pyramid mapping; while q_3 is answered by reading a single leaf, i, q_1 requires the read of the



(a) Four pyramids in $d = 2$ (b) One of the six pyramids in $d = 3$

Figure 3.2: Spatial decomposition using the pyramid technique.



(a) Spatial decomposition (b) B+ tree (leaves only)

Figure 3.3: Two- to one-dimensional mapping using the pyramid technique

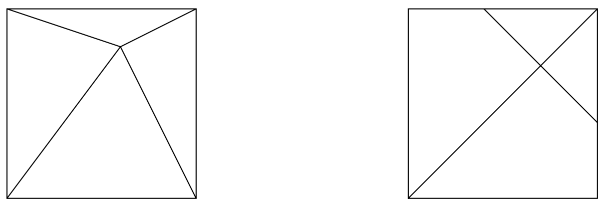
contiguous leaf range d–g. q_2 illustrates a deficiency in locality preservation, requiring the non-contiguous leaf ranges d, h and j–k to be read. The structure is also rather susceptible to problems caused by non-uniform data; if a large amount of data were now to be inserted into i, recurrent splitting would produce a series of thin, vertical slices, each of low height but containing widely separated points. The authors propose an extension (the *extended pyramid technique*) to handle non-uniform data that moves the common pyramidal apex to a region of high data density, but this requires prior knowledge of the data distribution (or offline reorganisation) and relies on there being only one such region.

iMinMax(θ)

The iMinMax(θ) [42] technique is a variation of the pyramid technique. It makes shifting of the pyramidal apex, as suggested in the extended pyramid technique, a tunable parameter of the structure and defines a slightly different one-dimensional mapping for a point $[x_1, \ldots, x_d]$:

$$y = \begin{cases} x_{min} + \theta < 1 - x_{max} & d_{min} + x_{min} \\ x_{min} + \theta \geqslant 1 - x_{max} & d_{max} + x_{max} \end{cases}$$

where $x_{min} = \min \{x_i.\ 1 \leqslant i \leqslant d\}$, d_{min} is the dimension number of x_{min}, and x_{max} and d_{max} are defined analogously. With $\theta = 0$ this provides the same pyramidal decomposition as in the pyramid technique, but alters the one-dimensional sort order such that pyramids p_i and p_{d+i} $(0 \leqslant i < d)$ are adjacent and the ordering within pyramid p_i is reversed. The pyramidal apex is moved around by varying the tunable parameter θ, although one could argue that this is less flexible than the extended pyramid technique, because the apex can only move along the space's diagonal.



(a) Extended pyramid technique; apex at (0.6,0.8)

(b) iMinMax(θ); $\theta = 0.2$

Figure 3.4: Shifting of the pyramidal apex in the extended pyramid technique and iMinMax(θ).

Finally, we observe that the regions specified by iMinMax(θ) are only truly pyramidal when $\theta = 0$. In the extended pyramid technique, the pyramidal apex is moved off-centre but the pyramid edges continue to extend from the apex to each corner of the space (figure 3.4a). Variation of θ in iMinMax(θ) has the effect of moving the apex along the diagonal of the space, while retaining orthogonal pyramid boundaries (figure 3.4b). We note this merely for completeness; Samet [52] remarks that any difference in query performance is likely to be due to differences in algorithm implementation, as the spatial decompositions are fundamentally the same.

3.1.3 iDistance

iDistance [30] is a one-dimensional mapping based, not on a direct ordering within the space, but on a secondary property, distance. It is not, strictly speaking, a dynamic structure, so we discuss it only briefly. Indexing a dataset takes place in three stages:

1. The data are split into m partitions. The choice of m depends on the nature of the partitioning; if the decomposition is designed closely to match an underlying clustering in the dataset, m might be the number of clusters therein.

2. From each partition, P_i ($1 \leqslant i \leqslant m$), a representative object, O_i is drawn.

3. For each partition, its representative object, O_i, is used to calculate an index key, y, for every object p_i in the partition:
$$y = i.c + d(p_i, O_i)$$
where d is a metric distance function. The offset $i.c$ separates the ranges of y values for different partitions, c being required to contract the actual partition radius to prevent overlap between y-ranges.

The iDistance approach can be applied to general metric spaces, and, in the case of queries close to partition representatives, preserves locality well. Towards the extremities of a partition, however, widely dispersed points collect together solely on the basis of similar $d(p_i, O_i)$ values.

3.1.4 GiMP

The similarities between members of the class of index structures based on one-dimensional mappings enables parameterisation of a number of their features in the GiMP (generalized multidimensional data mapping and query processing) [61]. The GiMP uses a B+ tree to index one-dimensional, mapped values, and requires the implementation of a number of standard functions to support mapping of multidimensional points into the space (for storage and exact match queries) and regions (for queries of non-zero extent in the source space).

The key value for a point p, $y(p)$, is calculated using three functions; **Base**, **Distance**, and **Reference**, such that:

$$y(p) = \mathbf{Base}(p) + \mathbf{Distance}(p, \mathbf{Reference}(p))$$

This has a clear correspondence to key calculation in the iDistance mapping, described in section 3.1.3; for that structure's definition of y we have:

- $\mathbf{Base}(p) = i.c$
- $\mathbf{Distance}(p, O_i) = d(p, O_i)$
- $\mathbf{Reference}(p) = O_i$ where $\mathbf{Distance}(p, O_i) = \min(\mathbf{Distance}(p, O_j).\ 1 \leqslant j \leqslant m)$

In a GiMP implementation of the pyramid approach, **Base** returns the pyramid number, **Reference** the pyramidal apex and **Distance** the perpendicular distance between the point and the apex. In the case of space-filling curves, **Reference** is the origin of the curve and **Distance** a point's separation from it. **Base** is zero — its function is to identify individual region partitions (like clusters or pyramids) in techniques employing mappings that proceed via such partitionings, and is otherwise not required.

Extent-based queries in the GiMP are answered by mapping regions in space to one or more intervals in the mapping range, requiring two further functions. **MapRange** returns a set of intervals in the one-dimensional space corresponding to a query region (specified as a query object and a range). Support for K nearest neighbour queries is delivered using an incrementally expanding range query; **MapAnnulus** returns the set of intervals required to search a range after a central region thereof has already been explored, allowing the K-NN algorithm to call for the next set of intervals as required until the K nearest neighbours are found.

The idea behind the GiMP is to identify and hard-code similarities across the class of one-dimensional mapping techniques, while parameterising their differences as functions. The idea of generalising classes of structures in this way was first presented for post-and-grow trees in the GiST, later for space-partitioning trees of the quadtree type in the SP-GiST [1] and for one-dimensional mappings in the GiMP in 2005. We discuss the merits of such generalisations in section 3.2.1.

3.1.5 Mappings into more than one dimension

Given that the performance of multidimensional access methods tends to deteriorate with dimensionality (see section 2.3), an approach to performance enhancement is to map a dataset of dimensionality d into another of dimensionality k, where $1 < k < d$, using a technique like principal component analysis (PCA), then indexing the reduced-dimensionality dataset. We do not discuss these techniques further here, because they rely on discrimination between feature vectors being such that some features are more important than others, allowing unimportant features to be neglected. By definition, such transforms from d dimensions to k are only of value if they preserve locality. Note also that organising a transformed dataset remains, even after reduction of d, a multidimensional indexing problem.

3.2 Non-SPP structures

Non-SPP structures are access methods that permit overlap between the predicates in a node, and, as such, are not guaranteed to provide a unique, deterministic query path from root to leaf for every point in the tree. As a result, they are characterised by search algorithms that feature *backtracking*, informally the potential requirement to look in more than one place for data satisfying a query predicate.

The Hybrid-tree [11] is an example of the class of non-SPP structures discussed in this section, but that structure is better described in the context of other trees using kd-tree intranode organisations. We examine some of these in section 3.3, and for now merely note that the Hybrid-tree is omitted here, returning to it in section 3.3.4.

3.2.1 GiST

As the GiMP does for one-dimensional mappings, the GiST [26] provides a generalisation of the class of non-SPP structures. The GiST supports implementation of the B+ tree, but, in the main, the GiST's standard post-and-grow behaviour produces structures with predicates that inevitably overlap, because poorly-balanced node splits are not tolerated. The GiST identifies, and provides an implementation of, the common 'core' of a number of tree operations. Features that fall outside that core are parameters for the GiST, and providing implementations of those features is the means of implementing specific non-SPP structures. We begin this section with the GiST in order to obviate the need to describe operations' core behaviour more than once. The GiST uses an internal node structure that corresponds to an explicit representation of the interpreted $[\langle \mathit{LocalPredicate}, \mathit{PageId}\rangle]$ list of pairs described in section 2.4.3, which we specify using the term grammar introduced in chapter 2:

$$\begin{aligned} \mathit{Node} &::= \mathit{INode} \mid \mathit{LNode} \\ \mathit{INode} &::= \texttt{I}\,([\langle \mathit{EntryPredicate}, \mathit{PageId}\rangle]) \\ \mathit{LNode} &::= \texttt{L}\,([\langle \mathit{Key}, \mathit{Value}\rangle]) \end{aligned}$$

where an *INode*'s *EntryPredicate* is an explicit local predicate for the node at the associated address *PageId*. Hellerstein *et al.*'s leaf entries are 'index tuples' consisting of a (perhaps zero-extent) predicate and a pointer to a secondary data record, but for consistency we retain the leaf entry description introduced in chapter 2.

We illustrate the GiST's use of calls to parameterised functions with a presentation of the `Insert` algorithm in figure 3.5, in which those calls may be found emboldened in the 'where' clause of the relevant rules. We assume a constant maximum number of entries per node, M, common to internal and leaf nodes. The configuration structure for the operation is as follows:

$$\langle \texttt{C}\,,\, r\,,\, \pi\,,\, \sigma \rangle$$

where, as in the B+ tree example of section 2.5.5, `C` is a command, r is a page identifier, π is a stack of page identifiers and σ is a page store. The grammar for the GiST insertion command is

$$\mathit{gistInsertCommand} ::= \texttt{ins}\,(\mathit{Entry}) \mid \texttt{S} \mid \texttt{R}\,(\mathit{EntryPredicate}) \mid \texttt{D}\,([\langle \mathit{EntryPredicate,PageID}\rangle])$$

Where `ins`, `S` and `D` are used in the same way as in the B+ tree case; we describe the use of the `R` command below. Note that the `D` command now contains a list of ⟨*EntryPredicate,PageID*⟩ pairs; the explicit pairwise representation requires two predicates to be posted after a node split, and for convenience we post the full details of both node entries required to replace the parent entry of the splitting node, encapsulated in a list.

Many GiST-type structures, for example the R-tree, must sometimes expand one or more subtree predicates to accommodate an incoming entry. To avoid the possibility of writing such an update to disk on descent and having to write the disk page for a second time if the node subsequently overflows (on returning from the insertion), the GiST postpones predicate adjustments until returning to the root from the leaf level. This is the purpose of the `R` command: it carries any necessary predicate update into a parent node.

Insertion is bootstrapped from an external `gistInsert` command configuration containing the ⟨*Key*, *Value*⟩ pair for insertion, the root address and the store, and terminates by making a transition into `gist` configuration containing the updated store and the (possibly new) root page ID. We described parameterised split policies in section 2.2.5; **PickSplit** is that of the GiST, and **Penalty** is some structure-specific measure of the cost (usually in terms of deterioration in structure quality) of inserting a point into a subtree. **Union** is used to create parent predicates for collections of entries (or to expand existing predicates on the insertion of new entries).

The GiST insert algorithm is given in figure 3.5. It is structurally very similar to the B+ tree algorithm; transitions 2.1 – 2.5 are exactly analogous to the first five B+ tree transitions, handling bootstrapping, tree descent and the three leaf insertion cases (replacement of an entry, appending an entry to the leaf without overflow and appending an entry to the leaf causing overflow). The use of `first` in transition 2.2 effects choice of an arbitrary subtree meeting the condition and is consistent with the GiST implementation [33].

Upward predicate adjustment requires three possible transitions to handle return from a node without overflow:

- In transition 2.6, a predicate p is posted into the node, replacing an existing predicate. This requires recalculation of the node's own predicate, yielding p' to be posted into the level above.

- In transition 2.7, a predicate p is also posted into the node, but is found to be equal to that which it is to replace. We therefore make no replacement, so no amendment to the node's own predicate is required; the output configuration command, `S`, has no parameters.

- Transition 2.8 demonstrates handling of `S`, and corresponds to a single step in returning to the root of the tree.

Transitions 2.9 and 2.10 handle the two cases of entries being posted into a node and either that node overflowing (2.10) or not (2.9). In both cases, the posted sequences are of length 2, but we do not specify this in the input configuration; it is instead specified in the two places in which the lists are constructed prior to posting (in transitions 2.5 and 2.10).

Transitions 2.11 – 2.13 handle termination of insertion, in the various cases where either the root node splits or does not. Note that change to an entry stored in the root node, but without root overflow, will result in a new predicate being posted from the root in an `R` command; storage of this predicate is not required because it represents the entire indexed space.

$$\left\langle \texttt{gistInsert}\,(\langle k,v\rangle, r, \sigma) \right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{ins}\,(\langle k,v\rangle)\,,\, r\,,\, [\,]\,,\, \sigma \right\rangle \tag{2.1}$$

$$\left\langle \texttt{ins}\,(\langle k,v\rangle)\,,\, r\,,\, \pi\,,\, \sigma \right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{ins}\,(\langle k,v\rangle)\,,\, s\,,\, r :: \pi\,,\, \sigma \right\rangle \tag{2.2}$$

if $\sigma(r) = \texttt{I}\,(\mathbf{E})$

where $i = \texttt{first}\,(\mathbf{E}, P)$ and $P = \lambda x.\mathbf{Penalty}(x, \langle k,v\rangle) = \min_{e\in\mathbf{E}} \mathbf{Penalty}(e, \langle k,v\rangle)$

and $\langle \_, s\rangle = \mathbf{E}_i$

$$\left\langle \texttt{ins}\,(\langle k,v\rangle)\,,\, r\,,\, \pi\,,\, \sigma \right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{S}\,,\, r\,,\, \pi\,,\, \sigma[r \mapsto \texttt{L}\,(\mathbf{E}')] \right\rangle \tag{2.3}$$

if $i \leqslant |\mathbf{E}|$

where $\sigma(r) = \texttt{L}\,(\mathbf{E})$

and $i = \texttt{first}\,(\mathbf{E}, P)$ and $P = \lambda x.\text{match } x \text{ as } \langle j, w\rangle \text{ in } j = k$

and $\mathbf{E}' = \texttt{repl}\,(\langle k,v\rangle, i, \mathbf{E})$

$$\left\langle \texttt{ins}\,(\langle k,v\rangle)\,,\, r\,,\, \pi\,,\, \sigma \right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{R}\,(p)\,,\, r\,,\, \pi\,,\, \sigma[r \mapsto \texttt{L}\,(\mathbf{E}')] \right\rangle \tag{2.4}$$

if $|\mathbf{E}| < \mathit{Max}_L$

where $\sigma(r) = \texttt{L}\,(\mathbf{E})$

and $p = \mathbf{Union}(\mathbf{E}')$

and $\mathbf{E}' = \texttt{append}\,(\langle k,v\rangle, \mathbf{E})$

$$\left\langle \texttt{ins}\,(\langle k,v\rangle)\,,\, r_L\,,\, \pi\,,\, \sigma \right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{D}\,([\langle p_L, r_L\rangle, \langle p_R, r_R\rangle])\,,\, r_L\,,\, \pi\,,\, \sigma[r_L \mapsto \texttt{L}\,(\mathbf{E_L})\,, r_R \mapsto \texttt{L}\,(\mathbf{E_R})] \right\rangle \tag{2.5}$$

where $\sigma(r) = \texttt{L}\,(\mathbf{E})$

and $\langle \mathbf{E_L}, \mathbf{E_R}\rangle = \mathbf{PickSplit}(\texttt{append}\,(\mathbf{E}, \langle k,v\rangle))$

and $p_L = \mathbf{Union}(\mathbf{E_L})$ and $p_R = \mathbf{Union}(\mathbf{E_R})$

and $r_R = \texttt{fresh}(\sigma)$

$$\left\langle \texttt{R}\,(p)\,,\, r\,,\, s :: \pi\,,\, \sigma \right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{R}\,(p')\,,\, s\,,\, \pi\,,\, \sigma[s \mapsto \texttt{I}\,(\mathbf{E}')] \right\rangle \tag{2.6}$$

if $p \neq q$

where $\sigma(s) = \texttt{I}\,(\mathbf{E})$

and $i = \texttt{first}\,(\mathbf{E}, P)$ and $P = \lambda x.\text{match } x \text{ as } \langle v, w\rangle \text{ in } w = r$

and $\langle q, t\rangle = \mathbf{E}_i$

and $\mathbf{E}' = \texttt{repl}\,(\langle p, t\rangle, i, \mathbf{E})$

and $p' = \mathbf{Union}(\mathbf{E}')$

$$\left\langle \texttt{R}\,(p)\,,\, r\,,\, s :: \pi\,,\, \sigma \right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{S}\,,\, s\,,\, \pi\,,\, \sigma \right\rangle \tag{2.7}$$

$$\left\langle \texttt{S}\,,\, r\,,\, s :: \pi\,,\, \sigma \right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{S}\,,\, s\,,\, \pi\,,\, \sigma \right\rangle \tag{2.8}$$

$$\left\langle \texttt{D}\,(\mathbf{P})\,,\, r\,,\, s :: \pi\,,\, \sigma \right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{R}\,(p)\,,\, s\,,\, \pi\,,\, \sigma[s \mapsto \texttt{I}\,(\mathbf{E}')] \right\rangle \tag{2.9}$$

if $|\mathbf{E}| < \mathit{Max}_I$

where $\sigma(s) = \texttt{I}\,(\mathbf{E})$

and $i = \texttt{first}\,(\mathbf{E}, P)$ and $P = \lambda x.\text{match } x \text{ as } \langle v, w\rangle \text{ in } w = r$

and $\mathbf{E}' = \texttt{del}\,(i, \mathbf{E}) \oplus \mathbf{P}$

and $p = \mathbf{Union}(\mathbf{E}')$

$$\left\langle \texttt{D}\,(\mathbf{P})\,,\, r\,,\, s_L :: \pi\,,\, \sigma \right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{D}\,([\langle q_L, s_L\rangle, \langle q_R, s_R\rangle])\,,\, s_L\,,\, \pi\,,\, \sigma \right\rangle \tag{2.10}$$

where $\sigma(s_L) = \texttt{I}\,(\mathbf{E})$

and $i = \texttt{first}\,(\mathbf{E}, P)$ and $P = \lambda x.\text{match } x \text{ as } \langle v, w\rangle \text{ in } w = r$

and $\langle \mathbf{E_L}, \mathbf{E_R}\rangle = \mathbf{PickSplit}(\texttt{del}\,(i, \mathbf{E}) \oplus \mathbf{P})$

and $q_L = \mathbf{Union}(\mathbf{E_L})$ and $q_R = \mathbf{Union}(\mathbf{E_R})$

and $s_R = \texttt{fresh}(\sigma)$

$$\left\langle \texttt{S}\,,\, r\,,\, [\,]\,,\, \sigma \right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{gist}\,(r, \sigma) \right\rangle \tag{2.11}$$

$$\left\langle \texttt{R}\,(p)\,,\, r\,,\, [\,]\,,\, \sigma \right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{gist}\,(r, \sigma) \right\rangle \tag{2.12}$$

$$\left\langle \texttt{D}\,(\mathbf{P})\,,\, r\,,\, [\,]\,,\, \sigma \right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{gist}\,(r, \sigma[s \mapsto \texttt{I}\,(\mathbf{P})]) \right\rangle \tag{2.13}$$

where $s = \texttt{fresh}(\sigma)$

Figure 3.5: GiST insertion rules.

Predicates are stored in GiST entries in compressed format; users of the GiST are required to implement two functions to support key compression: **Compress** and **Decompress** (although at their simplest these could consist of identity operations). These operators are not shown in figure 3.5, but could be added easily, by changing every store update to an update with compression; *e.g.* by substituting $\sigma[s \mapsto \mathtt{I}\,(\mathbf{Compress}(\mathrm{X}))]$ for $\sigma[s \mapsto \mathtt{I}\,(\mathrm{X})]$. The GiST's authors describe the use of compression for storage efficiency, for example in the prefix B+ tree [15], although in the GiST implementation of the (standard) B+ tree, the compression routines are used to effect a mapping into the GiST node structure:

$$[\langle i_0 : \mathit{Interval},\ a_0 : \mathit{PageId}\rangle, \ldots, \langle i_n : \mathit{Interval},\ a_n : \mathit{PageId}\rangle]$$

from a stored representation:

$$[\langle k_0 : \mathit{Key},\ a_0 : \mathit{PageId}\rangle, \ldots, \langle k_n : \mathit{Key},\ a_n : \mathit{PageId}\rangle]$$

which, with a stored length of 0 bytes for k_0, corresponds closely to the interleaved representation:

$$[a_0 : \mathit{PageId},\ k_1 : \mathit{Key},\ a_1 : \mathit{PageId}, \ldots,\ k_n : \mathit{Key},\ a_n : \mathit{PageId}]$$

This is effectively the interpretation of B+ tree local predicates from information stored in the node, as described in section 2.4.3, and, although intended to permit compression for the sake of storage considerations, appropriate implementations of **Compress** and **Decompress** might permit predicate representations like that of the BANG file (see section 3.3.5) to be supported in the GiST.

For query algorithms, subtree exploration is predicated on the value returned by the function **Consistent**. A value of `True` indicates that points satisfying the query predicate might be found in the associated subtree (as described in section 2.4.1, the GiST interprets `True` in this situation to mean '`Maybe`'), while a value of `False` indicates that they are certain not to be, permitting that subtree to be pruned from the search.

The power of the GiST (as for the GiMP) lies in its identification of core algorithms common to a number of structures. For the GiST, this is the set of post-and-grow trees whose insertion algorithms have a direct 'descend to leaf, post to parent' structure, but differ in their representation and manipulation of node predicates. This is useful both practically, by reducing the burden of implementation, and conceptually, by highlighting the features in which such structures differ. The practical applicability of the GiST is reduced, however, by the fact that access methods using insertion algorithms of this structure are few in number. For example, structures employing forced splitting, such as the K-D-B tree [49] (see section 3.3.1) or BANG file [21] (section 3.3.5) are not supported; similarly, support for structures using forced reinsertion, like the R*-tree [3] (section 3.2.3), requires a forced-reinsert flag to be set explicitly [12], undermining the GiST's claim to generality.

Even within the GiST class of access methods, many structures improve their performance by making small modifications away from the GiST's approach to predicate manipulation. The R-tree's **Penalty**, for example, is the area of expansion required of a rectangle to accommodate an incoming entry. In Guttman's implementation, when entries of equal expansion are encountered, their current rectangle areas are compared, but the GiST is not equipped to handle 'secondary' penalties of this kind (see [33, 48]). Similarly, as described in explaining the `R` command, the GiST adjusts node entries upwards after insertion to ensure that nodes' local predicates are contained in their global predicates (this is described using the *AdjustKeys* routine in [26] and captured by transition 2.6 in figure 3.5). The M-tree [14], however, relies on an optimisation that explicitly avoids this approach (see section 3.2.6), requiring alterations to be made to the core GiST code to enable the M-tree's implementation in the framework [46].

3.2.2 R-tree

The R-tree family of structures index vector spaces using predicates based on rectangles; a point satisfies the predicate if it is contained in the rectangle. The local predicate of a leaf node is $\lambda x.(x \in R)$, where R is the minimum bounding rectangle (MBR) of the points in the leaf. In the case of internal nodes, R is the MBR of the rectangle components of the node's entries. In consequence, every node N's local predicate is also its global predicate; $P_{\mathrm{N}} = \mathscr{P}_{\mathrm{N}}$. Rectangles are stored explicitly with the associated subtree page address, giving R-tree nodes a GiST-like structure:

$$[\langle R_0 : \mathit{Rectangle},\ a_0 : \mathit{PageId}\rangle, \ldots, \langle R_n : \mathit{Rectangle},\ a_n : \mathit{PageId}\rangle]$$

Published in 1984, the R-tree [25] could be considered to be the 'classic' non-SPP external tree structure. Its internal node structure corresponds closely with our notion of an interpreted node structure, and the provision of three split policies in the original paper illustrated, from the outset, their parametric nature. Of these split policies, one is of linear cost, one is of quadratic cost, and one is an expensive exhaustive algorithm that examines every split possibility before choosing that with least overlap. As might be expected, the quality of the partitioning increases with the cost of finding it, and, reflecting the trade-off between cost and quality, the quadratic-cost split policy became that of choice. The fact that split policies rarely eliminate overlap altogether means that backtracking is a regular feature of search in R-trees, even in the case of exact-match queries. The reason for this is simple: overlap of node predicates means that some regions of space are indexed by more than one node. If a point in such a region of space is sought, then every node indexing that region must be searched.

On insertion of a point, the subtree requiring least rectangle area expansion to accommodate it is selected, choosing that with smallest area where more than one such subtree exists. This continues recursively, until the point undergoing insertion reaches the leaf level, at which time, if the selected leaf node is already full, it is required to split. The overflowing leaf splits into two and posts a pair of entries into the level above, one of which replaces the parent of the splitting node. If there is sufficient space in the node for the second entry, it is added to the node, otherwise the internal node splits. We describe below the action of the quadratic split policy for such an internal node split, observing that leaf splits are handled the same way by treating points as entry predicates with zero extent. $\mathbf{E}$ is a set of entries from an overflowing node, and includes the new entry that provoked the overflow. We assume that a node may contain no fewer than m entries. For brevity, we refer to the MBR of a set of rectangles $\{R \mid \langle R, a\rangle \in \mathbf{E}\}$ as the MBR of $\mathbf{E}$.

QuadraticSplit:

1. **PickSeeds**. For every possible pair of entries $\langle R_i, a_i\rangle$, $\langle R_j, a_j\rangle$ drawn from $\mathbf{E}$, construct J, the MBR of R_i and R_j and calculate $d = \text{area}(J) - \text{area}(R_i) - \text{area}(R_j)$. Let the pair of entries with largest d be the first elements (or *seeds*) of two sets, $\mathbf{E_L} = \{\langle R_i, a_i\rangle\}$ and

$\mathbf{E_R} = \{\langle R_j, a_j\rangle\}$, and remove them from **E**.

2. **PickNext**. Let J_L be the MBR of $\mathbf{E_L}$ and J_R that of $\mathbf{E_R}$. For each entry e in **E**, construct J'_L, the MBR of $\{e\} \cup \mathbf{E_L}$ and J'_R, the MBR of $\{e\} \cup \mathbf{E_R}$. Calculate $d_L = \text{area}(J'_L) - \text{area}(J_L)$ and $d_R = \text{area}(J'_R) - \text{area}(J_R)$. Choose the entry e_{best} for which $\min(d_L, d_R)$ is smallest, and remove it from **E**.

3. Consider the values of d_L and d_R for e_{best}. If $d_L < d_R$, add e_{best} to $\mathbf{E_L}$, if $d_L > d_R$, add e_{best} to $\mathbf{E_R}$, otherwise add it to the set with smallest MBR.

4. If $|\mathbf{E_L}| + |\mathbf{E}| = m$, move every entry of **E** to $\mathbf{E_L}$, otherwise if $|\mathbf{E_R}| + |\mathbf{E}| = m$, move every entry of **E** to $\mathbf{E_R}$.

5. Repeat from step 2 until **E** is empty.

An exhaustive split policy will, by definition, find the ‘best’ partitioning, usually determined by a measure of overlap, because it considers all possible partitionings. A good split policy considers fewer partitionings, doing so in such a way that they are likely to be amongst the better possible partitionings, and then selects a partitioning from amongst that limited set. In the quadratic algorithm, the approach taken is to calculate a single partitioning by seeding the output sets with the entries it would be most expensive to group together (in terms of overlap), then adding entries to either set according to which addition would be cheapest.

The R-tree may be regarded as being at the head of a (now large) family of R-tree variants. We discuss the R*-tree in section 3.2.3; a wider survey is available in [40].

3.2.3 R*-tree

The R*-tree [3] is identical in structure to the R-tree, differing only in its approach to reducing overlap by using an insertion algorithm modified in three ways.

First, the authors tested several ways in which to select a subtree for insertion, and determined that selection of a rectangle on the basis of lowest increase in intranode overlap was sometimes more effective than lowest increase in area. Specifically, improvement was noted in the level of the tree immediately above the leaf, and so the mode of subtree selection was amended in this level only.

The second feature by which the R*-tree distinguishes itself from the R-tree is the use of *forced reinsertion*. The authors observe that the quality of many dynamic structures is dependent on the order in which data points are inserted, and that the behaviour of such trees can be improved by deleting a proportion of their entries and reinserting them. The R*-tree implements this by, at node overflow, first removing a proportion of the overflowing node’s points or entries (the authors suggest 30%) and reinserting them. Because this could cause an unbounded chain of further overflow and reinsertion, this approach is taken no more than once in each level in the tree for a single insertion operation, and is otherwise dealt with using a conventional node split.

The manner of such a node split is the subject of the R*-tree’s final modification — a revised split policy. Given a minimum node occupancy of m entries, a maximum capacity of M entries and a d-dimensional data space, the R*-tree’s split policy generates and evaluates $d.2(M - 2m + 2)$ distributions of a set of entries as follows:

1. For each axis (dimension) of the space:

   (a) Sort the entries into order of their rectangles' lower bounds and create $M - 2m + 2$ distributions by splitting the list into two sublists at every point that yields a split with permitted balance.
   (b) Sort the entries into order of their rectangles' upper bounds and create $M - 2m + 2$ distributions by splitting the list into two sublists at every point that yields a split with permitted balance.
   (c) Calculate the 'margin value' (MBR perimeter/2) for each of the $2(M - 2m + 2)$ distributions for this dimension, and sum them.

2. Choose the dimension with lowest margin value sum and generate distributions in that dimension again, as above.

3. Choose the distribution with the lowest area of intersection between the MBRs of its two sublists.

This approach is far more successful than the R-tree quadratic algorithm at reducing overlap, often eliminating it altogether from certain node splits in spaces of low dimensionality.

The R*-tree's authors report the surprising result that, despite the cost of reinsertion, build IO cost is actually lower in the R*-tree than in the R-tree. This is hard to explain, and indeed our results (see section 7.2.1) would suggest a build IO cost that exceeds that of the R-tree by some 30%. The benefits offered by the R*-tree, however, are clear in the area of query performance, where the IO cost of query execution can be less than half that of the R-tree.

3.2.4 SS-tree

Similarity queries are typically implemented using the assumption that similarity is quantifiable as a distance between objects in space, and that objects can satisfactorily be represented using feature vectors such that the distance between those vectors captures the intuitive notion of similarity. This notion of quantifiable similarity leads to the specification of *range* queries as a query object and radius, implying (for Euclidean distances) a hyperspherical query region. White *et al.* suggested that this sort of query geometry might be better supported by a structure with the same geometry of spatial decomposition, and proposed the SS-tree [60] as such a structure. (This notion was used subsequently to motivate SPY-TEC [35], a spherical decomposition based pyramid technique analogue).

The SS-tree entry structure uses hyperspherical local predicates, representing them explicitly as $\langle \mathit{Centroid}, \mathit{Radius} \rangle$ pairs, and including additional information such as the number of children in the entry's child node, the total number of children in its subtree and an 'update count'. Radius values are updated periodically (after a specified number of updates to the node) using an entry's children; each entry includes a radius variance to prevent over-assertive pruning during query execution while between radius updates. This ensures that, as in the case of the R- and R*- trees, every node's local predicate is contained by its global predicate.

Like the R*-tree, the SS-tree relies on forced reinsertion to allow periodic reorganisation, but pursues the policy rather more aggressively. When splitting a node, one of the entries resulting from the split replaces the original parent of the splitting node in the level above, while the other is always reinserted. When reinserting an entry, on reaching that entry's (new) parent, the parent's child count is checked. If the parent's child is not already full, reinsertion of the entry continues

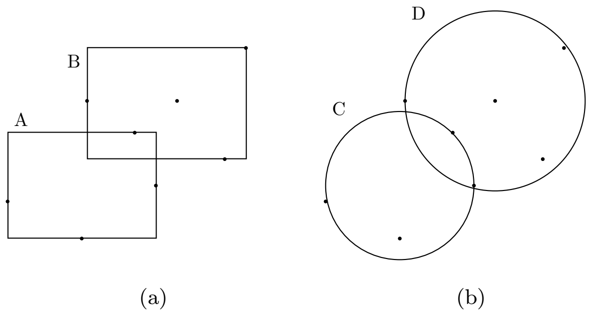


Figure 3.6: Spherical index regions may have higher volume than rectangular regions and consequently tend to suffer from a higher degree of overlap. In this example, in indexing the same set of 8 points, entries C & D cover more area than A & B.

into its new location (incrementing the parent entry's child count). If, however, the desired location is already full, a proportion (a suggested 30%, as in the R*-tree) are removed for reinsertion unless entries from the overflowing node have already been reinserted during this operation. Therefore, while the R*-tree permits only one 'batch' of reinsertions from each level in the tree, the SS-tree permits a batch from every node in the tree, potentially a significantly larger reorganisation.

The authors of the SS-tree claim a query performance gain over the R*-tree for similarity indexing on the basis of query region geometry. It seems likely, however, that the performance gain experienced by the SS-tree is due, at least in part, to its more extensive reorganisation behaviour, and indeed its construction IO cost is concomitantly higher than the R*-tree [31].

3.2.5 SR-tree

The authors of the SR-tree [31] present results suggesting that, particularly in spaces of higher dimensionality, a hypersphere bounding a given set of points encloses more space than a hyperrectangle (an example comparison is given in figure 3.6). Appealing, however, to the advantageous effects, cited by the SS-tree's authors, of the geometry of a hyperspherical decomposition, the authors use an explicit local predicate representation consisting of both representations: an overlaid sphere and rectangle. The effect of this is for the sphere to clip off unoccupied regions of space from the rectangle (see figure 3.7), producing a composite region description that corresponds to the intersection of the two.

The SR-tree's insertion algorithm is the same as that of the SS-tree; subtree selection is made on the basis of proximity to an entry's centroid, one of each pair of posted entries is always reinserted, and entries from an overflowing node are reinserted unless entries from that node have already undergone reinsertion during the primary insert operation.

The two competing features affecting the performance of the SR-tree with respect to the SS-tree are its capability for more precise region description (as illustrated in figure 3.7), improving pruning during search, and its larger node entry sizes, reducing overall fanout. As might be expected, the index construction IO cost is higher than for the SS-tree, but perhaps for no other reason than that the SR-tree requires more disk pages to index a given data set than does the SS-tree. Despite larger index size, the SR-tree still manages to report improved query performance because, in common with many structures, the vast majority of subtree pruning occurs at the level above the leaf. The

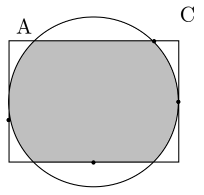


Figure 3.7: Overlaid spherical and rectangular regions provide a more precise predicate description in the SR-tree.

authors present a breakdown of disk accesses between internal node and leaf pages in both the SR- and SS-trees, indicating that, given a query answered by both structures indexing a common dataset, the SR-tree will read more internal nodes than the SS-tree, but fewer leaf pages.

3.2.6 M-tree family

The access methods we have discussed so far permit the indexing of vector spaces, in which an object's position can be described absolutely. The M-tree [14] permits the indexing of general metric spaces, that is to say spaces in which a distance function, with certain properties, is defined, but the notion of position need not be. Using $d(X, Y)$ to mean 'the distance from object X to object Y', these properties are:

- **Non-negativity**: $d(X, Y) \geqslant 0$; $d(X, Y) = 0 \leftrightarrow X = Y$
- **Symmetry**: $d(X, Y) = d(Y, X)$
- **Triangle inequality**: $d(X, Z) \leqslant d(X, Y) + d(Y, Z)$

Features of M-tree node entries are similar to those of SS-tree entries, with the exception that, because of the absence of the notion of position, the SS-tree entry predicate's centroid is replaced with a copy of an object drawn from the data space, referred to as a *routing object.*

The M-tree's region representation is of the form $\langle$*Routing Object, Radius*$\rangle$, and given a region $\langle O, r\rangle$, the predicate interpreted is $\lambda x.d(x, O) \leqslant r$. Entry radii, referred to as *covering radii*, are updated as objects being inserted pass into the entry's subtree, and, unlike in the SS-tree, are not updated based on their children save in the case of node splits. (It is this optimisation that required alteration of GiST code for the M-tree's implementation). The term 'radius' means 'the maximum distance from the routing object at which an object in the entry's subtree can be found', and does not necessarily indicate a hyperspherical predicate geometry; rather the geometry of the predicate is dependent on the nature of the metric employed. In three dimensions, while the Euclidean d_2 distance does indeed produce spherical predicates, the d_1 ('Manhattan') distance produces diamond-shaped regions and the d_∞ distance produces cubes.

Consider the example in figure 3.8, in which we identify entries in tree diagrams by their routing object labels in the associated space diagrams. We use r_{X} to denote the entry covering radius associated with routing object X. Object O_i is undergoing insertion into the subtree rooted on X, but since $d(\mathrm{X}, O_i) > r_{\mathrm{X}}$ (figure 3.8b), the value of r_{X} is updated to $d(\mathrm{X}, O_i)$ (figure 3.8c). O_i is subsequently inserted into the subtree rooted on B, causing a similar update of r_{B} to $d(\mathrm{B}, O_i)$ (figure 3.8d). Notice that the boundary formed by r_{B} around B in figure 3.8d now exceeds that of r_{X} around X. This is not incorrect, since no actual object in the subtree rooted on B falls outside

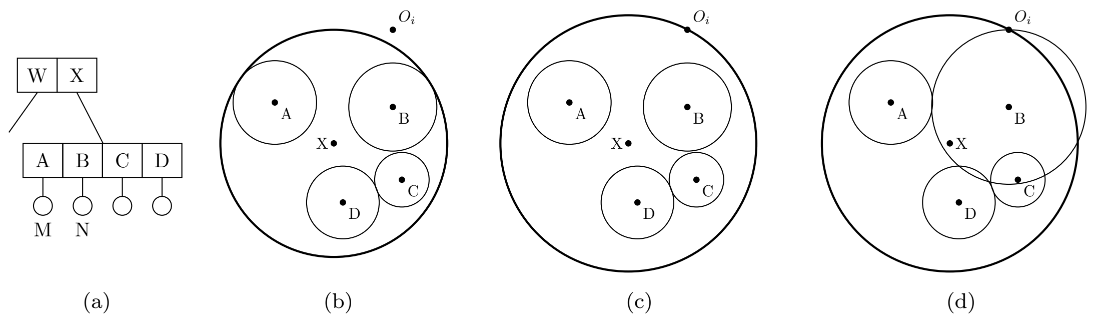


Figure 3.8: Update of M-tree covering radii only as far as is locally necessary produces children whose local predicates are not contained by their global predicates.

r_{X} of X, but node M's global predicate, $\lambda x.(d(x, \mathrm{B}) \leqslant r_{\mathrm{B}} \wedge d(x, \mathrm{X}) \leqslant r_{\mathrm{X}})$, now no longer contains its local predicate $\lambda x.d(x, \mathrm{B}) \leqslant r_{\mathrm{B}}$. Similarly, if a node, for example M, were to overflow, its parent entry would be replaced by two entries with region descriptors $\langle \mathrm{A}', r_{\mathrm{A}'} \rangle$ and $\langle \mathrm{A}'', r_{\mathrm{A}''} \rangle$, the predicates interpreted from either of which might no longer be contained in $\lambda x.d(x, \mathrm{X}) \leqslant r_{\mathrm{X}}$. In general, an M-tree local predicate is not guaranteed to be contained by the global predicate of the subtree at whose root it appears.

A practical effect of this in the M-tree is to remove the direct relationship between the local predicate represented in a node's entries and that represented in its parent entry: a parent entry's radius is dependent not directly on its child, but rather on all points that have been inserted into its child. The reasons for not expanding a parent's radius to encompass all those in its child are clear — reduced expansion produces reduced overlap — but this means that radii cannot easily be contracted on delete, and may actually expand when merging underflowing nodes. Furthermore, a bottom-up bulk-loaded tree does not approach the query performance of an insertion-built M-tree unless the data are carefully pre-clustered [54]; the M-tree authors' approach to bulk-loading the tree [13] is to construct an unbalanced tree, top-down, and rebalance it in a post-processing step. Similarly, a forced-reinsertion approach, analogous to that of the R*-tree, could introduce significant overlap at higher levels, as reinsertion of entries introduced large radius expansions. A forced reinsertion M-tree is described in [38], and permits reinsertion of point entries from overflowing leaf nodes only.

The original M-tree publication provided a suite of split policies, all of which, however, consist of two steps (analogous to the R-tree's **PickSeeds** and **PickNext**):

- **Promote** selects the routing objects for promotion in new parent entries.
- **Partition** distributes the splitting node's entries around the selected routing objects.

The Slim-tree [58] variant takes a more direct approach by constructing a minimal spanning tree (MST) of the splitting nodes entries and cutting the longest edge that provides acceptable split balance. The contents of each of the two nodes resulting from the split are formed from one portion of the split MST.

3.3 Structures lacking minimum fanout guarantees

This class of structures provides global predicate disjointness while attempting to preserve the notion of spatial locality lost in mapping techniques. This gives index structures of this type the single path property, but often at the expense of node occupancy guarantees. Node underoccupancy leads to taller trees as a result of low fanout, and generally higher disk space requirements. Node overoccupancy (effected by permitting nodes to occupy more than one disk page) is easier to justify if it reflects inevitable structural degeneration due to geometric effects (as in the case of the X-tree [8]) or if it has a well-defined upper bound (as in the BV-tree [19]); we examine these structures in section 3.4.

In section 3.3.4 we describe the Hybrid-tree, a non-SPP structure with strict occupancy guarantees. It is a member of the class of structures discussed in section 3.2, but is introduced here as a development of the K-D-B tree (section 3.3.1), requiring prior discussion of the latter.

3.3.1 K-D-B tree

Robinson's K-D-B tree [49] uses kd-tree based spatial decomposition, but unlike the original kd-tree implementation does not insist that the splitting dimension is chosen in a strictly cyclic order. kd-tree based decomposition produces disjoint local predicates by definition, and disjointness is enforced at K-D-B node splitting to ensure that the single path property is maintained. As in the R-tree family, local predicates are interpreted directly as rectangles (rather than from an intranode kd-tree), and for a node N we have $P_{\mathrm{N}} = \mathscr{P}_{\mathrm{N}}$; interpreted entries are therefore effectively of the form $\langle \mathit{GlobalPredicate}, \mathit{PageId} \rangle$. When splitting a set of such entries, $\mathbf{E}$, from an overflowing node N, a split boundary is chosen and the entries that lie on either side of that boundary are distributed into two corresponding nodes. There may, however, exist entries that lie *across* the split boundary; each of these must be split forcibly into two entries that no longer cross the boundary, permitting them to be distributed between the two nodes described above. Each forced split of an entry requires the recursive forced split of its child node, potentially with poor split balance and consequential underoccupancy.

We describe this in the language of predicates as follows. Choice of a split boundary amounts to the choice of two disjoint global predicates, $\mathscr{P}_L$ and $\mathscr{P}_R$, for the two nodes resulting from the split. We therefore have $\mathscr{P}_L \cup \mathscr{P}_R = \mathscr{P}_N$, and for every entry $\langle \mathscr{P}, a \rangle \in \mathbf{E}$, exactly one of the following holds:

- $\mathscr{P} \subseteq \mathscr{P}_L$
- $\mathscr{P} \subseteq \mathscr{P}_R$
- $\mathscr{P} \nparallel \mathscr{P}_L \wedge \mathscr{P} \nparallel \mathscr{P}_R$

This allows $\mathbf{E}$ to be separated into three disjoint subsets:

- $\mathbf{E_L} = \{\langle \mathscr{P}, a \rangle \mid \langle \mathscr{P}, a \rangle \in \mathbf{E} \ \wedge \mathscr{P} \subseteq \mathscr{P}_L\}$
- $\mathbf{E_R} = \{\langle \mathscr{P}, a \rangle \mid \langle \mathscr{P}, a \rangle \in \mathbf{E} \ \wedge \mathscr{P} \subseteq \mathscr{P}_R\}$
- $\mathbf{E_{fs}} = \{\langle \mathscr{P}, a \rangle \mid \langle \mathscr{P}, a \rangle \in \mathbf{E} \ \wedge \mathscr{P} \nparallel \mathscr{P}_L \wedge \mathscr{P} \nparallel \mathscr{P}_R\}$

such that $\mathbf{E_L} \cup \mathbf{E_R} \cup \mathbf{E_{fs}} = \mathbf{E}$. Each entry in $\mathbf{E_{fs}}$ then undergoes a *forced split* to yield two (interpreted) entries $\langle \mathscr{P}_l, a_l \rangle$ and $\langle \mathscr{P}_r, a_r \rangle$, where $\mathscr{P}_l \subseteq \mathscr{P}_L$ and $\mathscr{P}_r \subseteq \mathscr{P}_R$, and which are then added to $\mathbf{E_L}$ and $\mathbf{E_R}$ respectively. a_l and a_r are the page addresses of nodes resulting from applying the forced split recursively to the child of a; we refer to this as *downward forced splitting* and describe the condition $\mathscr{P} \nparallel \mathscr{P}_L \wedge \mathscr{P} \nparallel \mathscr{P}_R$ as the *forced split criterion.*

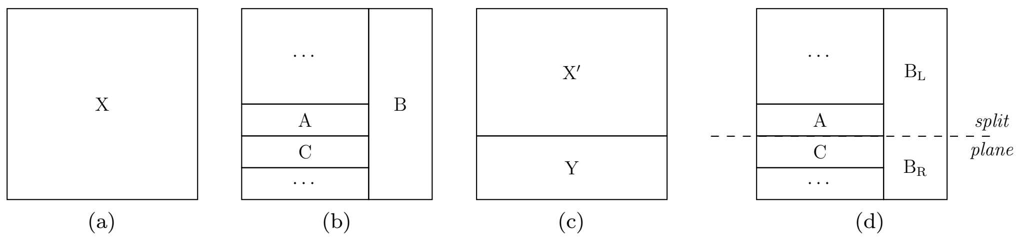


Figure 3.9: K-D-B tree decomposition of a two-dimensional data space, requiring forced split.

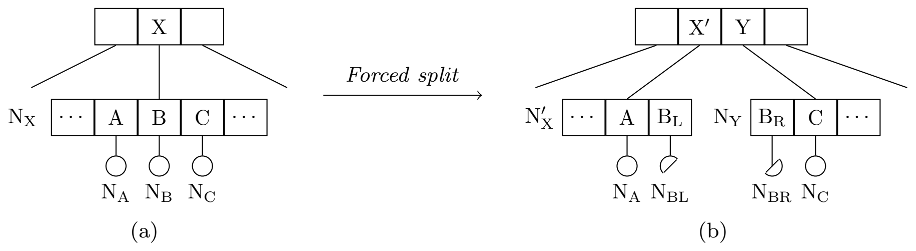


Figure 3.10: K-D-B tree fragment representing spatial decomposition shown in figure 3.9 undergoing forced split.

Downward forced splitting is the reason that we require $\mathscr{P}_L \cup \mathscr{P}_R = \mathscr{P}_N$. If, in the forced splitting case, $\mathscr{P}_L \cup \mathscr{P}_R$ were a proper subset of $\mathscr{P}_N$, (*i.e.* $\mathscr{P}_L \cup \mathscr{P}_R \neq \mathscr{P}_N$), applying the forced split to an entry $\langle \mathscr{P}, a \rangle$ in $\mathbf{E}_{fs}$ yields $\langle \mathscr{P}_l, a_l \rangle$ and $\langle \mathscr{P}_r, a_r \rangle$, where $\mathscr{P}_l \cup \mathscr{P}_r \subset \mathscr{P}$; points in the leaf level satisfying $\mathscr{P} \wedge \neg\mathscr{P}_l \wedge \neg\mathscr{P}_r$ have now been lost.

Figure 3.9a shows a two-dimensional region predicate, X, partitioned into subregions in figure 3.9b. Three of those regions are labelled explicitly A, B and C; partitioning of the remainder of the space (in the areas marked $\cdots$) is not shown. We assume that the number of subregions in the figure is such that the node $\mathrm{N_X}$ indexing those subregions (see figure 3.10a) must be split into two. This is effected by dividing region X into disjoint regions X′ and Y (figure 3.9c) and attributing the subregions of erstwhile region X to either X′ or to Y; with the exception of B, each subregion is contained in either X′ or Y.

Because region B is contained in neither X′ nor Y, it meets the forced split criterion and must be split forcibly, as described above, into the portion that lies above the split plane, $\mathrm{B_L}$, and that which lies below $\mathrm{B_R}$ (see figure 3.9d), generating two new node entries with global predicates $\lambda x.(x \in \mathrm{B_L})$ and $\lambda x.(x \in \mathrm{B_R})$, and splitting the contents of node $\mathrm{N_B}$ recursively, in a downward forced split. Note, however, that because the position of the $\mathrm{B_L}/\mathrm{B_R}$ split was determined solely by the geometry required to split the contents of node $\mathrm{N_X}$ there is no guarantee that the downward forced split of $\mathrm{N_B}$ has left a balanced split between $\mathrm{N_{BL}}$ and $\mathrm{N_{BR}}$. Furthermore, had $\mathrm{N_B}$ also been an internal node, predicates and subtrees therein may have required further downward forced splitting. In the worst case, it might be necessary to split forcibly every entry in a node into which a forced split is propagated.

Because K-D-B nodes' global predicates are disjoint, search for a point is a simple matter of identifying the single global predicate from amongst a node's entries that is satisfied by that point, then retrieving the child node associated with that predicate. The process is repeated recursively until the leaf level of the tree is reached and the search terminates.

The advantage of the K-D-B tree is that it offers a guaranteed exact-match query performance by maintaining the single path property. Its major disadvantage is that, in using downward forced splitting to enforce the SPP, it can leave nodes heavily underoccupied. This reduces tree fanout, causing search efficiency in the worst case to degrade from logarithmic to linear.

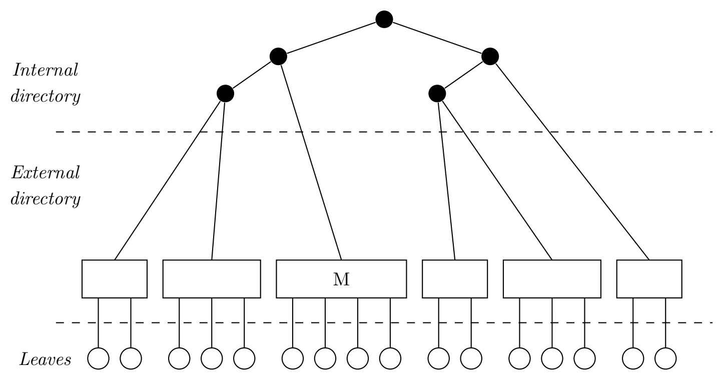
3.3.2 LSD-tree

The LSD-tree [27], like the K-D-B tree, uses non-cyclic kd-tree-based spatial decomposition, in this case using intranode kd-trees to describe the decomposition directly; nodes' local predicates are interpreted from the parent node's intranode kd-tree and are not contained in the associated global predicate. This is a significantly cheaper representation than the rectangle approach in the K-D-B tree, requiring $n-1$ kd-nodes in an LSD-node with n children. While a rectangle predicate consists of $2d$ coordinates (where d is the dimensionality of the space), a kd-node predicate consists of only a split location and dimension. This is independent of the dimensionality of the space, and, if the choice of splitting dimension is strictly cyclic, can be reduced further to a split location only. The structure is essentially a large kd-tree broken into fragments across disk pages for externalisation; we make the distinction between those fragments and the full tree by referring to *intranode kd-trees* and the (single) *extended kd-tree*.

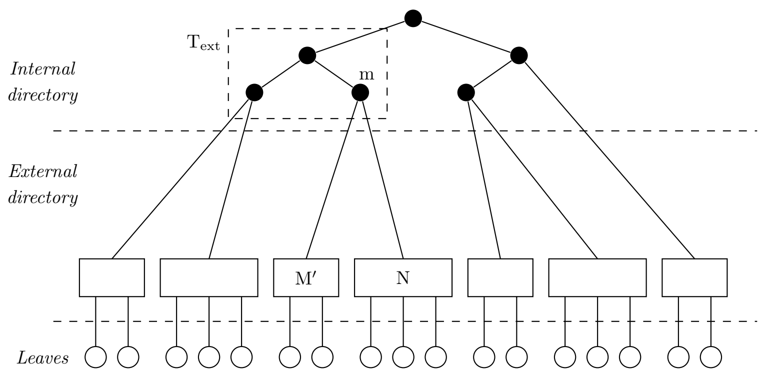
The LSD-tree retains the root and upper levels of the extended kd-tree in memory at all times as an *internal directory*; the *external directory* consists of the extended kd-tree's lower levels and is resident on LSD-node disk pages. Separation of the root fragment of the extended kd-tree from the rest of the structure (and into the internal directory) removes the constraint on that fragment of remaining within the capacity of a single disk page; instead, the internal directory is allowed a larger maximum size, specified in terms of number of kd-nodes. This is intended to provide greater flexibility in tree growth, to allow minimum node occupancy guarantees and tree height balance to be maintained. When a kd-node resulting from an external LSD-node split is posted into the internal directory and causes the internal directory to exceed its permitted maximum size, a paging algorithm extracts a subtree from the directory and flushes it to disk (see figure 3.11). The external directory can therefore grow in two directions: upwards, from a standard LSD-node split and kd-node post, and downwards as a result of paging an internal directory fragment.

The LSD-tree forbids forced splitting to avoid the associated occupancy problems, allowing a node to split only at the root of its intranode kd-tree, to ensure that none of the node's predicates are bisected by the split plane. This permits rather unbalanced splits in the case of poorly balanced kd-trees, in the worst case in the ratio 1:n, where n is the maximum number of child pointers that an LSD-node can accommodate (see figure 3.12). To limit split unbalancedness, a node is split not only on overflow but also if its intranode kd-tree exceeds a certain height, h_{max}, but this still permits a node split to result in underoccupancy and merely restricts the worst case balance to 1:h_{max}.

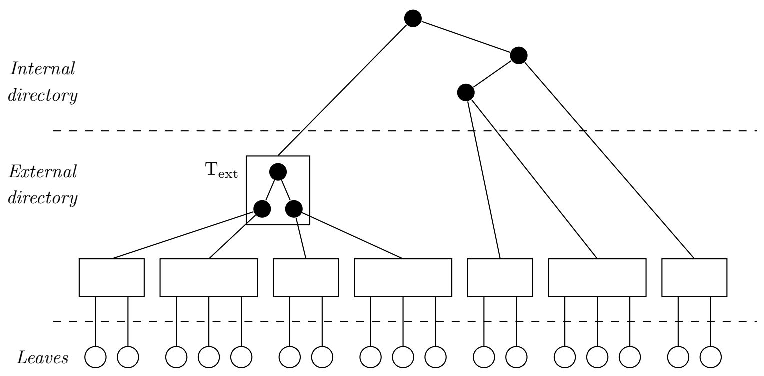
Downward growth is typically associated with unbalanced trees, and as figure 3.11 shows, paging does indeed introduce imbalance in the height of the LSD-tree. The authors claim that this is limited to a maximum of one external level by restricting eligibility for paging to subtrees in which all paths from kd-root to LSD-leaf visit the minimal number of external pages, H. Extraction of a subtree such that the external path length from its kd-root becomes $H+1$ thereby excludes ancestors of that subtree from further extraction until the entire LSD-tree is once again balanced and of height $H+1$. The problem with this claim is that there is at least one pathological case in which no subtree meets that eligibility criterion. Consider the example of figure 3.12. Supposing



(a) Node M is full, as is the internal directory.



(b) Further insertion into node M causes it to split, posting kd-node m into the internal directory. This in turn causes the internal directory to overflow, requiring paging of T_{ext}.



(c) T_{ext} is paged into the external directory, leaving the LSD-tree imperfectly balanced.

Figure 3.11: Paging of an internal directory fragment in the LSD-tree.

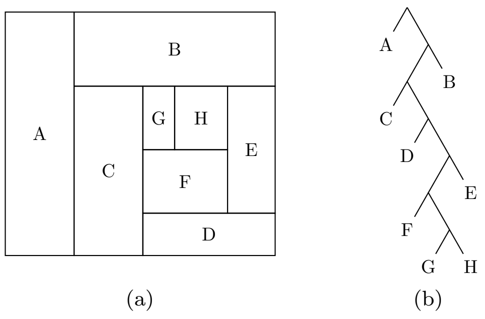


Figure 3.12: LSD-tree worst-case space decomposition and the associated kd-tree. The tree is of height 7 (not including leaves), and a root split will separate leaf entries in the ratio 1:7 ($\{A\} : \{B, C, D, E, F, G, H\}$).

the kd-tree to form an overflowing internal directory, paging might extract the lower three or four kd-nodes. If the paged kd-fragment continues to grow linearly, postings from a succession of (poorly balanced) root splits of the paged fragment would cause continued linear growth in the internal directory. Under these conditions, when the internal directory overflows, no subtree that meets the paging criterion will be found to exist. While a large internal directory may protect against this (a paging candidate being more likely to be found due to sheer weight of numbers), the structure remains at risk from very spatially-skewed data insertion orders. Furthermore, as Gaede and Günther [24] remark, requiring a special internal directory at all is an obstacle to incorporation of the LSD-tree into a standard DBMS architecture.

The authors of the LSD-tree acknowledge that, while efficient to store, all spatial decompositions represented directly using kd-trees automatically index the entire data space, occupied or not. They argue that 'dead space' indexing is of itself inefficient, because all tree searches must descend all the way to the leaf level, rather than terminating early, as might be possible if regions of dead space were not represented in the structure at all. Data-driven decompositions in the style of the R-tree need not necessarily index the entire space explicitly, and indeed we could choose in the K-D-B tree, as in the buddy-tree (see section 3.3.3), to avoid representing the whole space, by restricting rectangles' extents to cover only occupied regions. A solution for this problem was postulated in the LSD$^{\text{h}}$-tree [28], in which a distinction is drawn between the *potential data region* suggested by the kd-tree representation and the *actual data region* consisting of occupied space. Child node pointers in LSD$^{\text{h}}$-nodes are accompanied by a description of the actual data region to rule out areas of dead space; instead, however, of being described directly by an explicit rectangle, the actual data region is encoded in a bitstring corresponding to a rectangular range of cells from a 2^{zd} binary grid overlaying the space (a notion suggested to increase fanout in the Buddy-tree [53]; see section 3.3.3), where d is its dimensionality and z is a user-supplied integer parameter specifying the balance to be struck between precision and cost of representation.

3.3.3 Buddy-tree

Like the K-D-B tree, the buddy-tree [53] uses a non-cyclic kd-tree based decomposition, is fully paged, and represents node local predicates explicitly as rectangles. Like the LSD-tree, node splits are permitted solely at the root of intranode kd-trees (which, not being physically represented, must be constructed when splitting a node, allowing a node's composition to be described by possibly several kd-trees). Unlike both of these structures, however, spatial decomposition is *regular*: split of a region is permitted only at the midpoint of one of its faces. Regular partitioning is usually associated with structures of unbalanced height, unsuitable for external access methods, but its use here increases the likelihood of a spatial decomposition being describable by multiple kd-trees, in turn providing more opportunities for finding a well-balanced split. A possible secondary advantage is that, given an underoccupied partition, regular decomposition is more likely to provide *buddy*[1] partitions with which a merge would be permitted (while preserving an intranode kd-tree); the corollary to this, however, is that underoccupied partitions are more likely to exist simply because the spatial partitioning is regular. This of itself presents an additional constraint on partition selection at node split.

A feature of the buddy-tree's rectangle representation is, as noted in section 3.3.2, that rectangles can be contracted to cover only occupied space; *i.e.* the buddy-tree's local predicates are interpreted from minimum bounding rectangles (MBRs). In this representation, the kd-tree can be considered to decompose the space into disjoint subregions, each of which contains a single data partition whose spatial extent may be less than that of the subregion. As in the R-tree, insertion of new data may be accompanied by MBR expansion, but only up to the boundary of the containing kd-partition. Note, therefore, that insertion can occur into as yet unoccupied regions into which MBR expansion is not permissible. In such cases the buddy-tree permits the introduction of leaf pages containing a single point, but, although the tree may already have height $h > 1$, does not insist on the construction of a chain of pages of length h, permitting instead a degree of height imbalance. Subsequent overflow of such nodes is handled by insertion of an internal node at the node's present position, pushing it and its buddy into the level below.

Some example partitionings are given in figure 3.13, in which kd-tree split planes are marked, and MBRs displayed as filled rectangles.

- Figure 3.13a shows the decomposition of a space into regular kd-regions and the reduction in dead space indexing as a result of MBR representation.
- kd-tree decomposition of the partitions in figure 3.13b is possible in either of two ways. A root split of the kd-tree shown in figure 3.14a gives a buddy-tree node split balance of 4:2, but that of the tree in figure 3.14b a better ratio of 3:3.
- Figure 3.13c illustrates that expansion of the MBR into an adjacent grid cell (to accommodate the point p) is possible, as long as the kd-tree decomposition is not compromised.
- In figure 3.13d, expansion of the MBR in region Q to accommodate point p is not possible, because region $\mathrm{P} \cup \mathrm{Q}$ is not regular.

Description of MBRs in high-dimensional space becomes increasingly expensive, and the grid overlay provided by the regular kd-tree decomposition suggests the optimisation to which we alluded in section 3.3.2. A bit-encoding for a range of grid cells requires significantly less space to represent than an n-dimensional MBR, albeit with some loss of precision.

The use of (exact or grid-approximated) MBRs in the buddy-tree is its key advantage over other kd-tree based structures, allowing it to avoid indexing dead space and permitting early termination of some queries; furthermore its use of regular spatial partitioning makes it less sensitive to insertion order. Like all kd-tree based decompositions, however, the buddy-tree remains vulnerable to skewed data distributions, all the more so for its regular mode of spatial decomposition.

[1]The use of a regular decomposition implicitly defines a grid over the space, similar to that found in the grid file [41]. The term *buddies* refers, in the grid file, to a pair of adjacent cells that can be merged on underflow into the original cell from which they were produced.

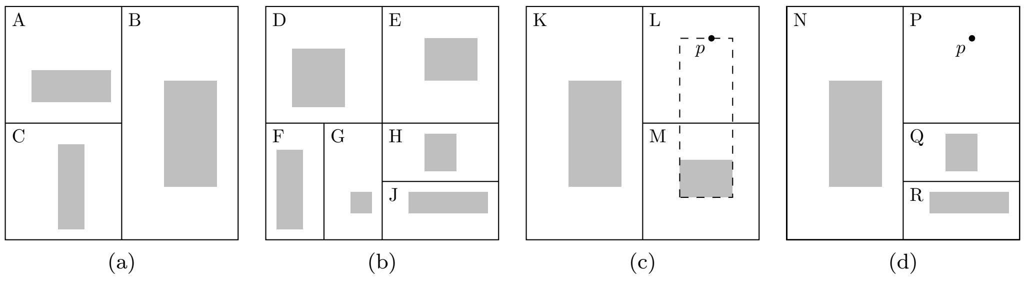


Figure 3.13: Examples of spatial decomposition in the Buddy-tree.

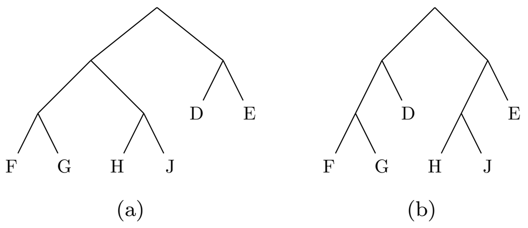


Figure 3.14: kd-tree decompositions of the space shown in figure 3.13b.

3.3.4 Hybrid tree

The Hybrid tree [11] is a hybrid of the K-D-B tree and other non-SPP structures, forbidding overlap unless it is necessary to avoid a forced split; forced splits are not permitted. This puts it into the class of non-SPP structures, rather than that of structures lacking occupancy guarantees, but its discussion has been postponed until now to allow our presentation of the K-D-B and LSD-trees to provide the necessary context.

The Hybrid tree uses an adapted kd-tree representation that permits overlap by using kd-nodes that, in addition to a splitting dimension, s, contain *two* location values, l and h, with $l < h$, that define two boundary positions in s. Given a d-dimensional space and points of the form $x = \langle x_1, \ldots, x_d \rangle$:

- h, the higher value in the splitting dimension, defines a region *low*, with predicate

$$\lambda x.\text{match } x \text{ as } \langle x_1, \ldots, x_d \rangle \text{ in } x_s \leqslant h$$

- l, the lower value in the splitting dimension, defines a region *high*, with predicate

$$\lambda x.\text{match } x \text{ as } \langle x_1, \ldots, x_d \rangle \text{ in } x_s > l$$

Because $l < h$, overlap is introduced between the regions *low* and *high*.

Consider figure 3.15, showing a cyclic kd-tree decomposition of a space yielding ten partitions. The LSD-tree approach to splitting this collection of entries would be to do so at the root, $y = 30$, producing the unbalanced split [A,B,C]:[D,E,F,G,H,I,J] in the ratio 3:7. The K-D-B tree's *local* balancing of the split is much better at 5:6, with a split at $y = 20$ yielding [A,B,C,D_R,E]:[D_L,F,G,H,I,J], where D_L and D_R are the results of splitting D forcibly, but the balance between the results of forcing that split into D's subtree may be very poor. The Hybrid-tree, in this situation, produces a perfectly balanced 5:5 split, [A,B,D,G,H]:[C,E,F,I,J], as shown in figure 3.15c, but at a cost of losing the single path property.

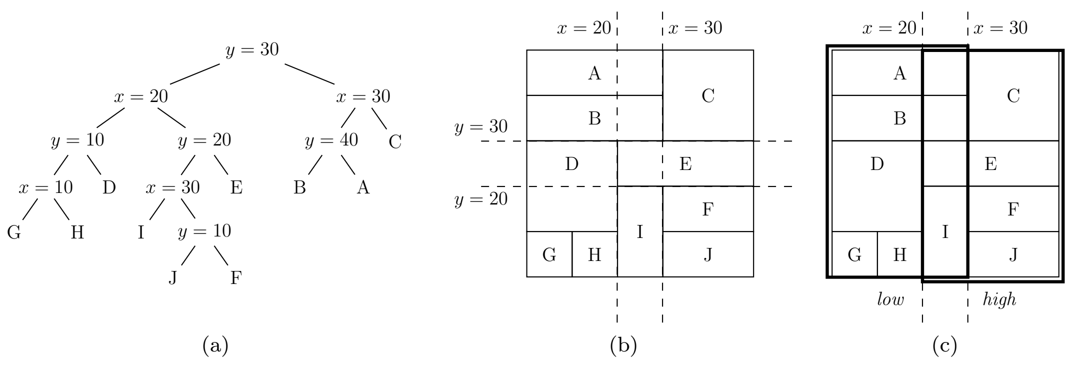


Figure 3.15: kd-tree and associated cyclic spatial decomposition. The LSD-tree would split at $y = 30$, and the K-D-B tree at $y = 20$, splitting D forcibly. The Hybrid-tree splits with *low*:$x < 30$, *high*:$x > 20$.

As we described in section 3.3.2, the kd-tree decomposition requires much less space to represent on disk (even with the addition of a second split position value in the Hybrid-tree) and is independent of dimensionality, but is prone to index dead space. If indexing dead space is undesirable, then doing so more than once due to overlap is even less so, and so the Hybrid tree employs a binary grid overlay approach like the LSD^h and buddy trees, referring to it as an *encoded live space optimisation.*

The main difference between the Hybrid-tree and the R-tree family is therefore one of representation; both structures permit overlap to enable split balance preservation, and entry size in both grows with increasing dimensionality. The Hybrid tree chooses splits on the basis of least increase in expected number of disk accesses, which coincides with the motivation behind R-tree split selection on the basis of minimising overlap. The authors report a significant performance gain (reduced IO cost) in benchmarking the structure against the SR- and hB-trees [39]; this seems likely to be the result of higher fanout due to lower entry size than in the former, and reduced dead space indexing compared to the latter.

3.3.5 BANG file

We described the BANG (balanced and nested grid) file [22] partitioning scheme in section 2.4.3. Recalling that BANG file node entries have the structure $\langle \mathit{OuterRegionBoundary}, \mathit{PageId} \rangle$, we will not discuss the partitioning scheme here further other than to add a note on representation of outer region boundaries.

Because the region decomposition is binary and regular, and because entry predicates are represented using only regions' outer boundaries, regions can be described efficiently as bitstrings. Consider the two-dimensional example in figure 3.16. The outer boundary of A is the whole space, so is represented by the empty string, []. The single-bit string has either the value [0] or [1], corresponding to the two halves of the first split of the space. From the figure it can be seen that the cyclic splitting order starts with a split in the vertical, so the 'upper' half of that split, B, is described by [1]. The 'lower' half is split again, with its lower half becoming C, thus described by [00]. Similarly D and E are addressed by [0111] and [0111001] respectively, or if we use the notation b^n as shorthand for a string of repeated bits of length n, $[01^3]$ and $[01^30^21]$.

In section 2.4.3 we motivated our description of predicate interpretation by alluding to situations in which failing to do so can cause error. Figure 3.17 shows an overflowing BANG file root node

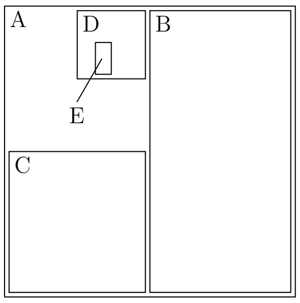


Figure 3.16: BANG file space decomposition. Regions A–E are represented by bitstrings [], [1], $[0^2]$, $[01^3]$ and $[01^30^21]$.

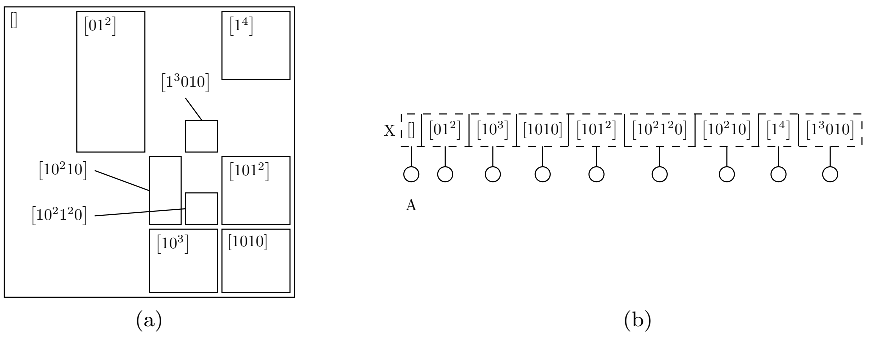


Figure 3.17: BANG file space decomposition and associated (overflowing) root node.

(represented as decoded from disk rather than using its interpreted structure) that must be split to create a new root; notice that the outer boundary for leaf node A, [], is larger than the region representing A's local (and in this case, global) predicate, because the other entries in node X describe 'holes' within A's outer boundary:

$$P_{\mathrm{A}} = \mathscr{P}_{\mathrm{A}} = \lambda x.x \in \left([] \setminus \left([01^2] \cup [10^3] \cup [1010] \cup [101^2] \cup [10^21^20] \cup [10^210] \cup [1^4] \cup [1^3010]\right)\right)$$

The split algorithm for regions, as described for points in section 2.4.3, consists of generating a hole, through a succession of binary splits, such that the balance between entries inside and outside of the hole is acceptable. In practice this is managed by repeatedly splitting the inner partition into two, in each case merging the less heavily occupied half with the outer partition, and continuing until the split balance between the inner and outer partitions is reversed. The series of partitionings for figure 3.17a is as follows:

Split	Outer partition		Inner partition		Balance
	Boundary	Elements	Boundary	Elements	
1	[]	[], $[01^2]$	[1]	$[10^3]$, [1010], $[101^2]$, $[1^4]$, $[10^21^20]$, $[10^210]$, $[1^3010]$	2:7
2	[]	[], $[01^2]$, $[1]^4$, $[1^3010]$	[10]	$[10^3]$, [1010], $[101^2]$, $[10^21^20]$, $[10^210]$	4:5
3	[]	[], $[01^2]$, $[1^4]$, $[1^3010]$, $[10^3]$, $[10^21^20]$, $[10^210]$	[101]	[1010], $[101^2]$	7:2

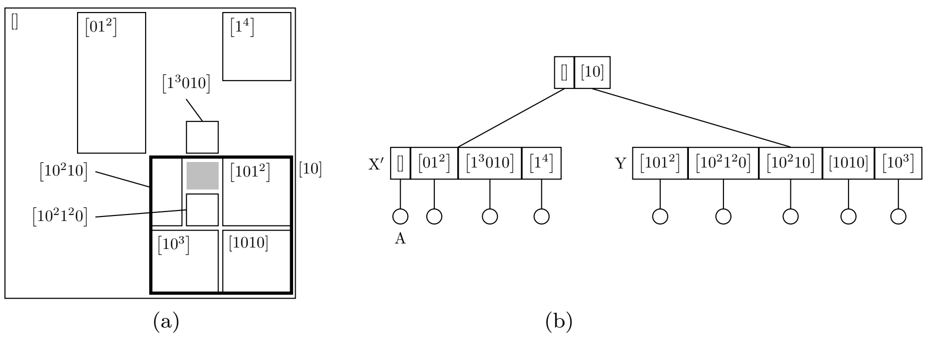


Figure 3.18: Execution of BANG file root split and tree growth in response to overflow shown in figure 3.17. The shaded area is ‘lost’.

resulting in the selection of the second partitioning, with split balance 4:5 and illustrated in figure 3.18 (again, node entries are shown as decoded, not interpreted). The outer boundaries associated with the page pointers to nodes X′ and Y are [] and [10] respectively; so we have:

$$P'_X = \lambda x.x \in ([] \setminus [10])$$

revising local and global predicates for leaf node A:

$$\begin{aligned}
P'_A &= \lambda x.x \in \left([] \setminus \left(\left[01^2\right] \cup \left[1^4\right] \cup \left[1^3010\right]\right)\right) \\
\mathscr{P}'_A &= \lambda x.\left(x \in ([] \setminus [10]) \wedge x \in \left([] \setminus \left(\left[01^2\right] \cup \left[1^4\right] \cup \left[1^3010\right]\right)\right)\right) \\
&= \lambda x.x \in \left([] \setminus \left([10] \cup \left[01^2\right] \cup \left[1^4\right] \cup \left[1^3010\right]\right)\right)
\end{aligned}$$

Clearly, $\mathscr{P}'_A \neq \mathscr{P}_A$, a problem that manifests itself in the ‘loss’ of region $\left[10^21^3\right]$, shaded in figure 3.18: that region is actually represented in a subtree of node X′, but searches for points therein will be directed into node Y.

The real problem here is that the information posted into the root to allow interpretation of local predicates for nodes X′ and Y is insufficient. In fact, the situation is that

$$\mathscr{P}_A \nparallel \mathscr{P}'_X \wedge \mathscr{P}_A \nparallel \mathscr{P}_Y$$

the exact description of the forced split criterion introduced in section 3.3.1. Two solutions to this problem were posited in [21], one of which was to execute the forced split as described here. Unlike the K-D-B tree, however, the extent of the requirement for forced splits in the BANG file is strictly limited.

The BANG file’s spatial decomposition scheme of nested partitions is such that, given a pair of outer region boundaries R and S, either $R \parallel S$, $R \subseteq S$ or $S \subseteq R$. Assume that S is the boundary for a node split, and that R_1 and R_2 are the boundaries associated with two entries to be partitioned. If S contains R_1, no part of R_1 belongs outside S, and no forced split of R_1 is required. Similarly, if S and R_1 are disjoint, no part of R_1 belongs inside S, and no forced split of R_1 is required. The only case in which a forced split of R_1 can be required, then, is that in which R_1 contains S. The same holds for R_2 and S; if R_2 is to require a forced split, it must be true that R_2 contains S.

Assuming then that both R_1 and R_2 contain S, now consider the relationship between the two;

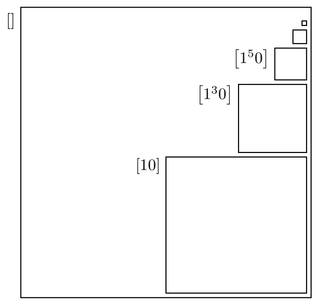


Figure 3.19: A pathological case for the BANG file. No spanning split can be found for the collection of entries $\left[1^{2i+1}0\right]$ where $0 \leqslant i \leqslant n$. Entries are labelled in the figure up to $i = 2$, and shown up to $i = 4$.

once again either $R_1 \parallel R_2$, $R_1 \subseteq R_2$ or $R_2 \subseteq R_1$. If $R_1 \parallel R_2$, at most one of the two can contain S, so only one forced split could be required. If $R_1 \subseteq R_2$, then R_1 is actually a hole in R_2, so although both boundaries contain S, the interpreted region $R_2 \setminus R_1$ does not intersect the region within S at all, and only R_1 requires a forced split. The converse is true if $R_2 \subseteq R_1$. In general, therefore, in any splitting node at most one entry will require a forced split. This means also that downward forced splits do not 'ripple out' further as in the K-D-B tree, but are also limited to, at most, one forced split per level below the original splitting node.

The second solution postulated for the 'region loss' problem was to allow only partitionings between two new parent predicates, $\mathscr{P}_L$ and $\mathscr{P}_R$, where no entry exists with predicate $\mathscr{P}$ for which it is true that $\mathscr{P} \nparallel \mathscr{P}_L \wedge \mathscr{P} \nparallel \mathscr{P}_R$. Freeston [21] describes these as *spanning splits* because of the spatial sense in which, in this situation, the union of a node's entry predicates is exactly equal to the entry predicate posted to the node's parent entry. The third split found when partitioning the example of figure 3.17 (for which the outer boundary associated with X′ is described by the bitstring [101]) is an example of a spanning split. Occasionally it will be found that a top-level split is naturally spanning; in such cases a hole need not be taken, but the node can be split with a conventional, disjoint 'buddy' split.

These approaches are exactly analogous to those taken by the K-D-B and LSD trees respectively in handling the problem of splitting an extended kd-tree representation, and suffer from the same deficiencies. Forced splits, even if limited to one per splitting node, are to be avoided for the previously described reason of underoccupancy, and spanning splits because there exist pathological cases in which every spanning partition consists of only one entry (figure 3.19 provides an example).

3.3.6 hB-tree

The hB-tree [39] has in its origins the recognition that a planar split of an intranode kd-tree may cause underoccupancy. We have seen that free choice of split plane (as in the K-D-B tree) leads to the requirement for forced splits, which, although locally well-balanced, may be less so when propagated to lower levels of the tree. The only split plane choice in an intranode kd-tree guaranteed not to require a forced split is at the kd-root, although the local balance of such a split may be very poor; consider, for example, the kd-tree of figure 3.12b. A result presented in [39], however, indicates that a worst-case occupancy lower bound of one-third can be guaranteed if an intranode kd-tree is split in more than one dimension simultaneously.

Like the LSD tree, the hB-tree uses an explicit kd-tree intranode representation; we use the

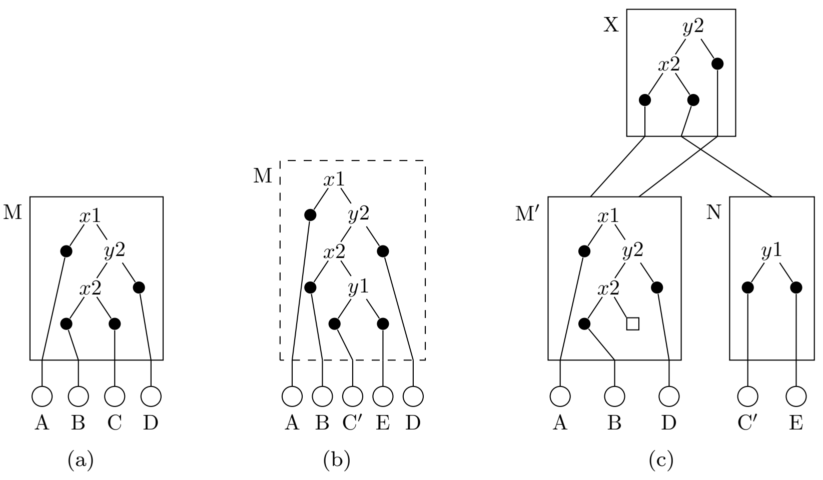


Figure 3.20: hB-tree node splitting.

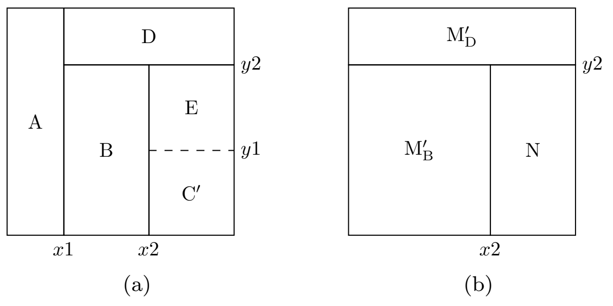


Figure 3.21: Spatial decomposition described by the hB-tree in figure 3.20.

same *intranode* or *extended* kd-tree terminology to distinguish kd-trees confined to a single hB-node from that kd-tree spread throughout all hB-nodes. Figure 3.20a shows an hB-node with four children. In figure 3.20b, split of node C into nodes C′ and E has caused posting of kd-node $y1$ into node M, in turn causing M to overflow, indicated by the dashed node boundary. The spatial decomposition described by the overflowing M's kd-tree is shown in figure 3.21a, with the new split at $y1$ shown as a dashed line.

In general, let the number of kd-nodes in an intranode kd-tree be n. A split point in that tree that respects the occupancy guarantee described above is found by descending the more heavily occupied subtree of successive kd-nodes, beginning with the intranode kd-root, until a subtree is found containing between $n/3$ and $2n/3$ nodes. In the case of M, this occurs at node $x2$, so the subtree rooted on $y1$ is *extracted*, as a corner of the space, to form a new node. The smallest kd-fragment necessary to represent the corner (referred to as the *condensed path*) is posted into the level above, yielding the hB-tree shown in figure 3.20c; the square kd-node beneath $x2$ in M′ indicates that a subtree has been extracted at this point; M′ is now a 'holey brick'. Posting the condensed path as an index term places an upper bound on the size of posted terms; in a k-dimensional space, a corner can be represented by k bounds, and any hole by at most $2k$. It is often considerably smaller, since any kd-node previously posted need not be posted a second time.

This decomposition has two notable features. Firstly, the kd-tree in node X contains two nodes that refer to hB-node M′, so the extended kd-'tree' is really a directed acyclic graph (DAG). Secondly, the condensed path posted to X is such that neither of the two subregions of M′ represented

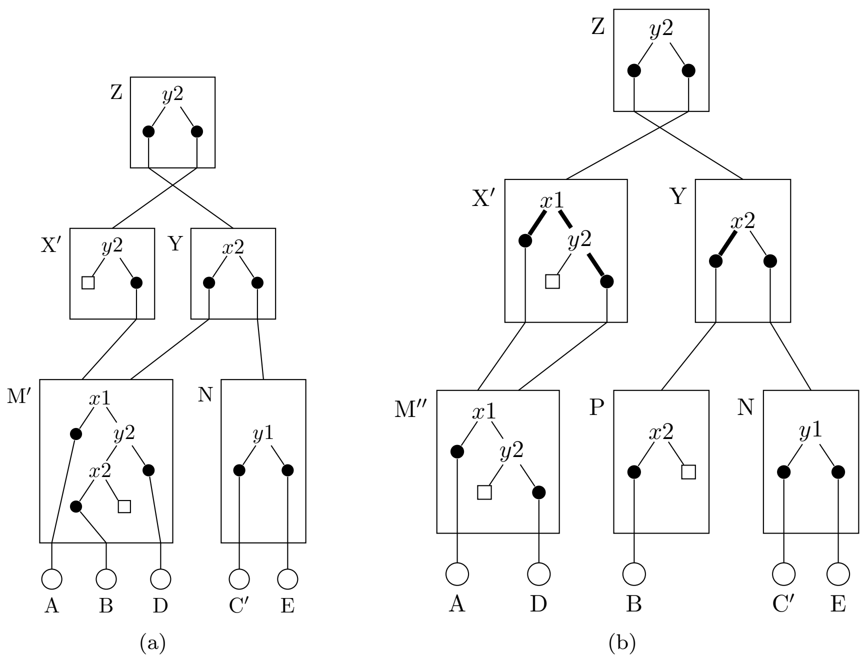


Figure 3.22: Split of hB-node M′ into M″ and P. The resulting index term is integrated into X′ and Y in such a way that region $(x < x1, y < y1)$, a subregion of node A, is lost.

in X (shown in figure 3.21b as M'_B and M'_D) is the union of a set of subregions represented within M′; both contain a portion of region A. A subsequent split of node X at $y2$ would separate the kd-nodes referring to M'_B and M'_D between two hB-nodes, giving node A two hB-grandparents and making the hB-tree itself also a DAG. This requires careful handling at a later split of M′ to ensure that split information is posted to its (potentially many) hB-parents as required. In fact, in the original implementation of the hB-tree, integration of the condensed path into the extended kd-tree above a splitting node was found to permit 'loss' of portions of such multi-parent nodes.

We provide an example of this by executing further splits in the hB-tree of figure 3.20c. Figure 3.22a shows the result of a split of node X into nodes X′ and Y. In practice, this split would not occur until split of several of node X's children had caused it to overflow, however we proceed directly to the split of X to avoid cluttering the discussion. In the figure, split of node X has caused extraction of the kd-fragment rooted on $x2$, separating the two parents of node M′.

Suppose that node M′ now undergoes a split by extraction of the subtree rooted on $x2$. Figure 3.23a shows the intranode kd-tree of M′ and figures 3.23b and 3.23c the two fragments resulting from its split. In figure 3.23d, each child node pointer from figure 3.23a has been replaced with the label of the node which will contain that pointer after the split; either M″ or P. Index term posting consists of integrating this information into the extended kd-tree above nodes M″ and P. Figure 3.22b shows the result of integrating the fragment into nodes X′ and Y, with the heavier lines in the kd-tree corresponding to those marked in figure 3.23d. Note that the condensed path is just $x1$, since of the bounds describing the split, $x1$ and $y2$, only $x1$ has not previously been posted. As a result of integrating the fragment, however, M″ is no longer a child of Y. Searches for a point (x, y) where $x < x1 \wedge y < y2$ will now be directed through node Y into node P, although any point stored in the tree answering such a query is actually located in node A, a child of M″.

A number of solutions to the problem were reported in [16]. These included posting full (instead of condensed) paths; had this been the case in figure 3.20, kd-node $x1$ would have been posted into

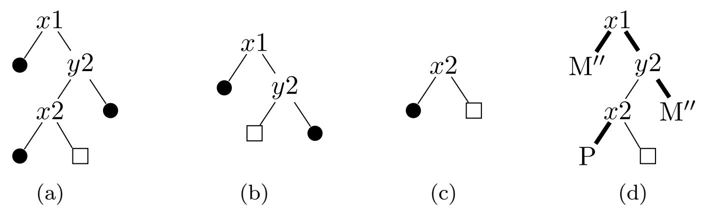


Figure 3.23: kd-tree fragments related to split of node M′ in figure 3.22.

hB-node X and subsequently into nodes Y and Z, correctly maintaining the graph. An alternative is to forbid splits that separate kd-nodes with a common child into different hB-nodes, constraining the external structure to remain a true tree; this would have forbidden the split point selected to form nodes X′ and Y in our example. This is referred to in [16] as *splitting at decorations* and is equivalent to Freeston's spanning split constraint in the BANG file. Finally, observing that the problem in our example occurred when the split between X′ and Y resulted in node Y containing an incomplete description of the region it represents, a split that preserves the so-called *complete boundary* of Y would be correct.

The problem with each of these three approaches is one of node occupancy: posting full paths increases the size of index terms, while constraining permitted split positions in an intranode kd-tree to either decorations or complete boundaries may require the structure to tolerate poorly balanced splits. Unbalanced splits cause underoccupancy simply by providing a node with too few entries, but enlarged index terms do so by physically reducing a node's capacity for children, because the average amount of space required to represent a single child increases. In either case the effect is to reduce fanout and cause the tree's height to increase.

3.4 Structures requiring variable node sizes

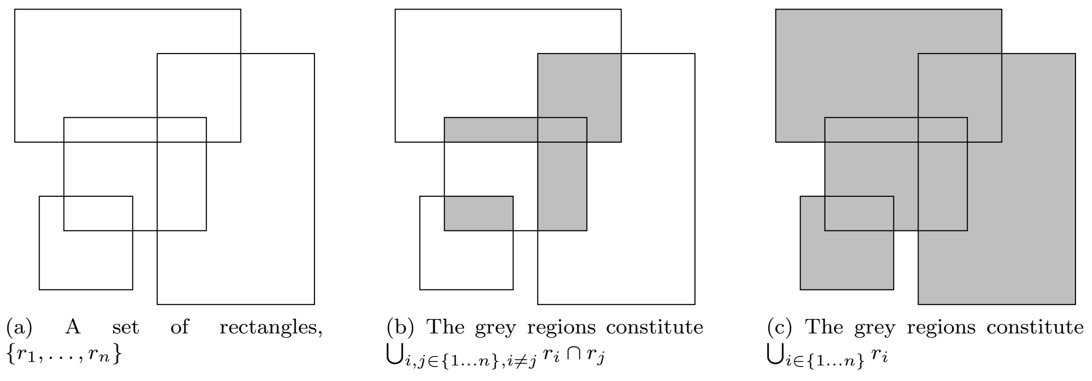
There are comparatively few structures that fall into this class, and in this section we present only two. The assignment of the BV-tree to this category may seem controversial at first, but we present a variation on an argument of Samet's [51] that justifies this. We note that variable node size does not necessarily indicate IO-imbalance, however, for a structure with variable-sized nodes to be IO-balanced, it must exercise some degree of control over node size variability.

3.4.1 X-tree

Berchtold *et al.* [8] define overlap between members of the set of local predicate rectangles represented in a node, $\{R_1, \ldots, R_n\}$, in terms of the ratio of space occupied by more than one rectangle (the union of all pairwise intersections of R_i and R_j) to that occupied by any rectangle, $\cup_{i \in \{1 \ldots n\}} R$:

$$\textit{Overlap} = \frac{\textit{Space occupied by } \bigcup_{i,j \in \{1 \ldots n\}, i \neq j} R_i \cap R_j}{\textit{Space occupied by } \bigcup_{i \in \{1 \ldots n\}} R_i}$$

(see figure 3.24). The definition of *space occupied by* is taken simply as the total area of a region in the case of uniformly distributed data, but in other distributions is calculated as the number of data elements in the node's subtrees that lie in that region. This is referred to as *weighted overlap*. The authors present results that suggest that the average weighted intra-node overlap in the R*-tree approaches 100% in as few as 10 dimensions, indicating that, whatever the quality


(a) A set of rectangles, $\{r_1, \ldots, r_n\}$

(b) The grey regions constitute $\bigcup_{i,j \in \{1 \ldots n\}, i \neq j} r_i \cap r_j$

(c) The grey regions constitute $\bigcup_{i \in \{1 \ldots n\}} r_i$

Figure 3.24: Components of Berchtold *et al.*'s definition of overlap

of a node split, the curse of dimensionality renders this degree of overlap unavoidable. With this much overlap, answering an exact-match query might require reading a substantial portion of the R*-tree's nodes, while such a query can be answered by reading, on average, only half of the pages of an unstructured data file. The X-tree [8] is a response to this observation; a structure that of itself degrades by degrees into a flatter structure in situations when using a tree-based index would make queries more expensive to answer.

The approach is implemented by providing a 'split policy' that can, in certain cases, choose not to split a node at all. A 'normal' split is first attempted using an appropriate split policy, after which the overlap between the MBRs of the two sets is evaluated. If that overlap exceeds a certain threshold, the node remains unsplit and is allowed to occupy two disk pages as a 'supernode'. If already a supernode of n pages, the node grows to $n + 1$ pages and remains unsplit.

The X-tree is essentially an R-tree with variable-sized nodes, the average size of which increases with dimensionality d. As one might expect, results presented for the X-tree suggest R-tree-like performance at low d and better than R-tree performance at medium d, as lower overlap reduces unnecessary node reading (even overcoming higher average node-read cost). On the basis of the overlap results described above, we would anticipate that in very high d, average exact-match query execution cost in the X-tree would approach that of reading half the tree's leaves and a very small (largely collapsed) directory — essentially a sequential scan — while in the R-tree it might approach that of reading the entire tree.

3.4.2 BV-tree

Freeston's BV-tree [19][2] was proposed in 1995 but has since received little direct attention. We believe this to be because it is rather poorly understood. Descriptions of the BV-tree, throughout the literature ([18, 19, 23, 20, 51]), lack formal algorithms or performance data, suggesting to us that the structure has never been fully implemented. We present the BV-tree here briefly and return to a full treatment in later chapters.

Region descriptors for the BV-tree are described in terms of a required property — *containment* — rather than using an explicit representation. Given any pair of region boundaries, A and B, stored in the BV-tree, either A contains B, B contains A, or A and B are disjoint. Region partitioning consists of the removal of one region from another, as a hole; the BANG file's mode of

[2]Freeston's BV-tree should not be confused with the *bounding-volume* hierarchies proposed by Klosowski *et al.* [32] to support their approach to collision detection in collections of moving objects.

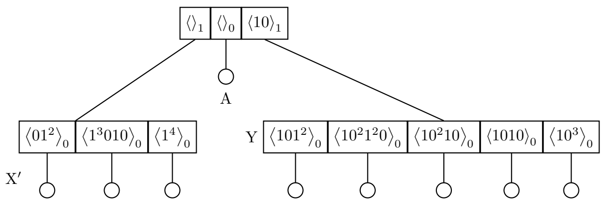


Figure 3.25: BV-tree reorganisation on execution of root split and tree growth in response to overflow shown in figure 3.17. Entry descriptors have been labelled with their original level in the tree.

region description is therefore consistent with the requirements of the BV-tree. Rather than being a single structure, the BV-tree forms, in theory, a class of structures that use this mode of nested region decomposition. In practice, however, the BANG file's region representation is the only one that we know to work, for reasons we describe in section 6.2.

In our presentation of the BANG file in section 3.3.5, we described the use of forced splitting to make the structure correct by avoiding region loss. Forced splitting occurs when an entry belongs partially in one subtree, and partially in another — we described this as the forced split criterion. Instead of splitting an entry satisfying the forced split criterion, the BV-tree idea is to *elevate* it into the level above. We refer to this as *virtual forced splitting.* The elevated entry can then be carried into either subtree during tree descent, so that the fragment of the entry belonging in each subtree respectively can be found therein. We describe this at greater length in chapter 4. Figure 3.25 illustrates the BV-tree approach to the BANG node splitting example in figure 3.18. In the figure, entries are subscripted with a number to indicate the level at which they would belong if the tree were fully balanced.

Elevation of entries causes a degree of deterioration in the tree's structure, by reducing a node's capacity for unelevated (*primary*) entries, reducing the number of leaves that can be reached from the node and requiring a larger tree for a given set of leaves. This requires careful handling if the tree is to continue to function well, or indeed — as we shall see in chapter 4 — at all. The BV-tree's nested region description provides the necessary control: the BANG file's limit of one forced split per level translates into an elevation limit of, for each primary entry in a node, one elevated entry from each level below it in the tree. Despite this, given a fixed node capacity, the average node fanout ratio decreases at higher levels in the tree, to the extent that effective node occupancy guarantees can still be undermined. This is discussed in greater depth in section 4.6.5. Freeston suggests that fanout from primary entries can be maintained by implementing the tree with a node capacity of $F.v$ where F is the required number of primary entries and v is the level of the node's primary entries; it is the introduction of this controlled variability in node size that leads us to classify the BV-tree amongst structures requiring variable node sizes. Note, however, that a node size of $F.v$ is constant in any given level of the tree, so the structure remains IO-balanced. A consequence of this approach is that, in the absence of elevated entries, a node's capacity for primary entries grows, reducing the efficiency of tree search. In section 4.6.5 we introduce an alternative approach to accommodating elevated entries, without allowing a node's primary capacity to increase.

Figure 3.26 shows a spatial decomposition and figure 3.27 a BV-tree indexing that space. The example is adapted from [19], and assumes that a general containment-based region representation is suitable. Notice that both regions b_1 and d_0 appear in the root, because both entries belong

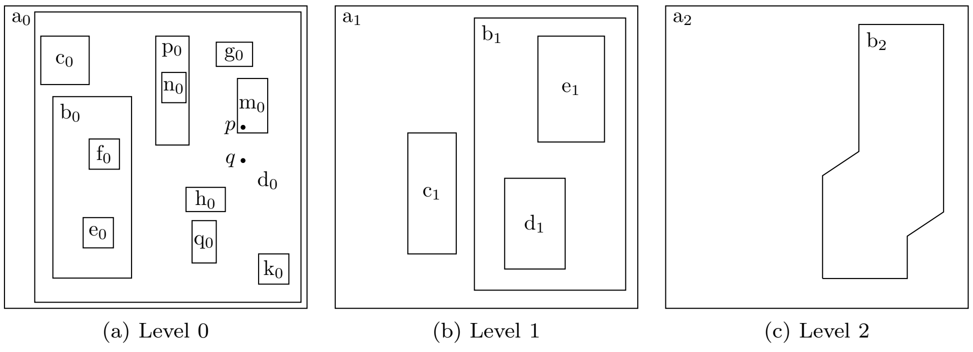


Figure 3.26: Spatial partitioning associated with the BV-tree shown in figure 3.27.

partially in the subtree rooted on a_2, and partially in that rooted on b_2. Exact match search for the point p (shown in figure 3.26a) proceeds as follows:

1. In the root, interpret local, possibly 'holey' predicates $\lambda x.x \in (a_2 \setminus b_2)$, $\lambda x.x \in d_0$, $\lambda x.x \in b_1$ and $\lambda x.x \in b_2$. The latter three are satisfied, so the primary subtree rooted on b_2 is selected for descent. Entries d_0 and b_1 are elevated, and must be carried down into b_2's subtree in a *pending set*, **G**.

2. Entries in the child of b_2 are d_1 and e_1, and when merged with **G** allow calculation of local predicates $\lambda x.x \in (b_1 \setminus (d_1 \cup e_1))$, $\lambda x.x \in d_1$, $\lambda x.x \in e_1$ and $\lambda x.x \in d_0$, of which the latter two are satisfied. The subtree rooted on e_1 is selected for descent with $\mathbf{G} = \{d_0\}$.

3. Merging $\{d_0\}$ with g_0 and m_0 gives local predicates $\lambda x.x \in (d_0 \setminus (g_0 \cup m_0))$, $\lambda x.x \in g_0$ and $\lambda x.x \in m_0$, of which m_0 is satisfied, its subtree selected for descent, and the search terminates.

Point q marked on figure 3.26 provides a second example, briefly:

1. b_2's subtree selected for descent, $\mathbf{G} = \{d_0, b_1\}$;

2. b_1's subtree selected for descent, $\mathbf{G} = \{d_0\}$;

3. d_0's subtree selected for descent.

Notice that, had q been closer to the boundary of m_0 such that it fell within e_1, search would have instead continued down e_1's subtree in step 2, but still with $\mathbf{G} = \{d_0\}$, so the search would still have terminated correctly in the child of d_0. Notice also that the entries required to interpret fully the local predicate for d_0 are scattered throughout the tree. At no time is d_0's local predicate ever fully interpreted; interpretation is restricted to the local region of the subtree being searched.

Abstractly, insertion is, as in other structures, a case of an exact match search potentially followed by node split and entry posting. In practice, however, significant difficulties arise. Consider the insertion of point q. As in the case of search, insertion proceeds, via b_2 and b_1, into the child of d_0. If the leaf node now splits, new parent entries for the two resulting leaves must be posted, but to where? Furthermore, if the split of the child of an elevated entry causes that entry to change, it may no longer be virtually split — maintenance of occupancy guarantees requires such entries to be demoted. These issues are described incompletely in the literature, but as we shall see in chapters 4 and 5, they are non-trivial.

Notions used here in our presentation of the BV-tree — virtual forced splitting, elevated entries, primary entries, pending sets — are features of our own description of the structure. The original

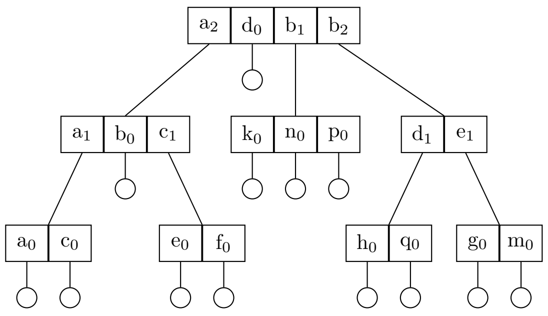


Figure 3.27: BV-tree used to index the spatial partitioning shown in figure 3.26.

presentation [19] refers to elevated entries as *guards* and pending sets as *guard sets*, and does not motivate well the reason for entry elevation. We avoid the guard terminology for elevated entries because it can mislead our intuition. In the case of figure 3.26, notice that a point may lie outside region b_2 but inside d_0, requiring d_0 to be included in the pending set accompanying descent of the subtree rooted on a_2. The BV-tree nomenclature describes d_0 as a guard of b_2, because d_0 contains b_2 and may mislead the reader into making an association solely between d_0 and b_2, and *not* between d_0 and a_2. As we see here, however, an asymmetric association of this kind is incorrect. Similarly, if, for example, entry c_1 ('guarded' by entry b_0) were to undergo a buddy split, say into c_1' and f_1, then descent into either of those would require b_0 in the guard set, so clearly b_0 guards both c_1' and f_1, and cannot be said to be the guard of solely one or the other.

In [51], Samet introduces the notion of *coupling*, the close correlation observed in other structures between a node of (unelevated) entries and the region of space that they index, leading to the description of the BV-tree as a structure in which entry elevation *decouples* the partitioning of space and the grouping of node entries. The notion of decoupling captures accurately the separation of recursive decomposition and its representation in the BV-tree, but falls short of providing an approach to handling decoupled structures. We believe that our concept of virtual forced splitting explains more clearly the motivation for promoting entries out of their 'natural' level, and is easier to understand and reason about. We motivate and introduce more fully the concept of virtual forced splitting in chapter 4.

3.5 Summary

We have, in this chapter, discussed some of the major contributions to hierarchical, external, multidimensional point access methods. Even with such an apparently specialised class of structures, there is a wealth of existing research and proposed structures, although as we have seen, the question of designing IO-balanced structures that preserve locality, guarantee minimum fanout and provide the single path property remains very much open.

We have described many of these structures using the notions of local and global predicates introduced in chapter 2. Most non-SPP structures encode bounded region descriptions on disk, such that predicate interpretation amounts to little more than decoding the region descriptor from its byte-level representation. Conversely, very many of the structures describing disjoint global predicates do so by providing only sufficient information to differentiate subtrees locally; for example the LSD-tree (section 3.3.2) or the BANG file (section 3.3.5). More complex structures of this type, for example the hB-tree (section 3.3.6) or BV-tree (section 3.4.2), require careful handling of this information; in particular, treating nodes as collections of data without clear regard to the predicates that must be interpreted from that data, possibly across multiple nodes, can lead to incorrectness.

Very many members of the SPP class of structures rely on either implicit or explicit kd-tree intranode organisations, principally because the kd-tree decomposition is binary. Quadtree [50] approaches have similar properties of disjointness, but a quadtree-based buddy-tree, for example, would require an overflowing node to be split into four, causing immediate occupancy issues. Furthermore, while attempts to provide external support for the quadtree have been made, the difficulty in providing a balanced structure restricts its use practically to in-memory applications.

Other structures not examined here include those simply outside the class of structures under consideration. These include grid-based approaches like the grid file [41] or EXCELL method [57]. Dimensionality reduction techniques (*i.e.* mappings to spaces of more than one dimension) such as transformations like discrete Fourier transforms (DFT) or principal components analysis (PCA), or structural approaches like the TV-tree [37] also fall outside the scope of our discussion.

Chapter 4

Virtual Forced Splitting

Our discussion in chapter 3 described tree structures in terms of the provision (or otherwise) of four properties: Locality preservation, IO-balance, minimum fanout guarantees and the single path property. We further described local properties of node splits that induce these global properties: Locality preservation through splitting in all spatial dimensions; IO-balance through post-and-grow behaviour and the imposition of maximum occupancy limits; lower limits on fanout by enforcing minimum node occupancy through control of split imbalance; and the SPP by disjointness of split predicates.

We have observed that simultaneous provision of all four desirable properties is rather difficult for multidimensional access methods, and in this chapter we examine why. We begin by revisiting the B+ tree in section 4.1, to illustrate that its application to only one dimension allows it to provide all four desirable properties. This forms a prelude to our discussion of why this becomes difficult in higher numbers of dimensions.

After first considering some issues of predicate interpretation in section 4.2, we proceed in section 4.3 to demonstrate that geometric effects of locality-preserving decompositions, in as few as two dimensions, make it difficult to partition a node's contents in a way that also provides other desired local structural properties of node splits. It is the diminished provision of each of these local properties that in turn gives rise to the classes of structures lacking the corresponding global properties described in sections 3.2–3.4.

In section 4.4 we consider forced splitting (as used in the K-D-B tree and BANG file) in some depth. Forced splitting provides all the required local properties of node partitioning, but fails to induce a global minimum node occupancy because the effect of the forced split is explicitly to split other nodes in the subtrees of the splitting node. Forced splitting remains, nonetheless, a promising approach because it provides locally a 'good' split, and in section 4.5 we introduce *virtual forced splitting*, a way in which to preserve forced split semantics while confining structural reorganisation to the vicinity of the splitting node. This allows locally-balanced splits to provide fanout guarantees without immediately undermining them by downward forced split propagation. The remainder of the chapter is devoted to discussion of issues presented by the virtual forced splitting approach.

4.1 The B+ tree

In this section we examine the B+ tree's provision of all four of the desirable characteristics described in section 2.2. Figure 4.1a shows a B+ tree fragment in which a leaf node can accommodate four integers and an internal node four child page pointers; both nodes M and X are full. For clarity we represent only the *Key* components of M's list of $\langle Key, Value\rangle$ pairs; likewise X is shown as an interleaved list of child page addresses and discriminator key values. Suppose that the value 31 is inserted into the tree. It belongs in node M, which now overflows (figure 4.1b). Because the data items in M form an ordered list (and because we do not permit duplicate values to be held in the leaf), we can always find a split point in the list such that the two resulting sublists are contained in a pair of disjoint interval predicates, required to induce the single path property. In fact, splitting at *any* point in the list gives us a pair of disjoint intervals, so we can always choose a split position with optimum balance, guaranteeing that the split of a list of length $2n$ has balance $n : n$ and that of a list of length $2n + 1$ has balance $n : n + 1$. This gives us a very good minimum occupancy guarantee that induces a global minimum fanout ratio.

Here, we split the overflowing leaf node M's entry list $[\langle 27, \_\rangle, \langle 29, \_\rangle, \langle 30, \_\rangle, \langle 31, \_\rangle, \langle 33, \_\rangle]$ with balance 2:3, producing the sublists $[\langle 27, \_\rangle, \langle 29, \_\rangle]$ and $[\langle 30, \_\rangle, \langle 31, \_\rangle, \langle 33, \_\rangle]$, contained by disjoint predicates $P_L = \lambda x.27 \leqslant x \leqslant 29$ and $P_R = \lambda x.30 \leqslant x \leqslant 33$. Recognising, however, that we can express the intervals' disjointness with a single value, we do so using the left-most value of the right-hand sublist, 30, implying the intervals $(-\infty, 30)$ and $[30, \infty)$. This value, along with the address of the new disk page onto which the second sublist is written (assuming the first list to have overwritten the page originally containing the splitting node), is posted into the level above. Upward growth of this kind induces height balance.

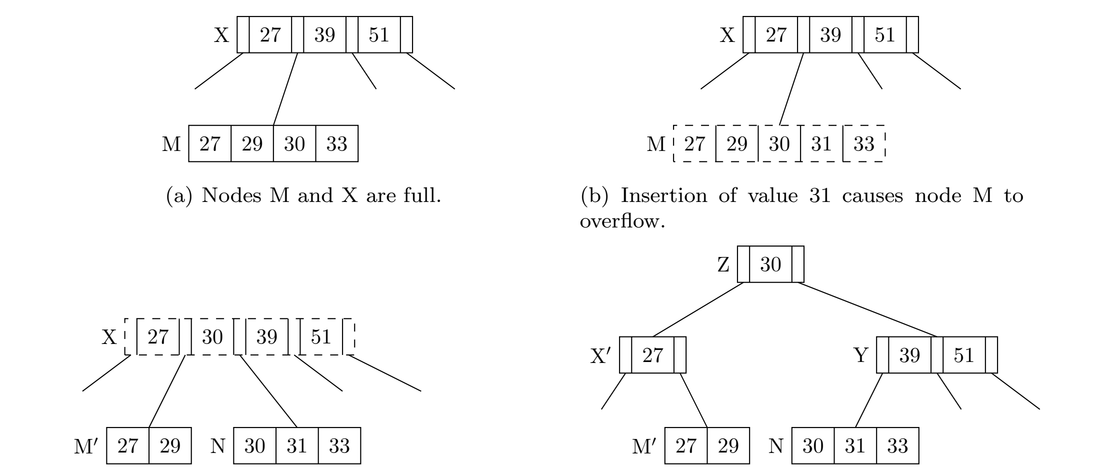
Posting of the new key value and page pointer into node X causes it in turn to overflow (figure 4.1c). The overflowing node contains two lists; an ordered list of key values and a list of associated page pointers, as described in section 2.5.5. The list of key values implies an ordered list of disjoint intervals, and because those interval predicates are both ordered and disjoint, we can, as in the leaf case, split the list into two sublists such that they can be represented by two further disjoint predicates and have lengths that differ, at most, by one (see figure 4.1d). Split of an internal node, therefore, has the same local properties as a leaf node split, required to induce all four desirable properties:

- locality is preserved because a global order is maintained across every entry in the tree;
- all node splits are disjoint, providing the SPP;
- node splits are optimally balanced, providing a high minimum fanout ratio;
- post-and-grow behaviour, in combination with a fixed node size, induces IO-balance.

This makes the B+ tree a very well-behaved structure for the indexing of one-dimensional spaces.

4.2 Handling predicate interpretation

Before proceeding further, we introduce some abstractions for predicate interpretation. In section 2.5 we described our abstract-machine approach to algorithm specification, using the B+ tree as an example in section 2.5.5. B+ tree nodes were described using their decoded node representation; effectively an interleaved list of key values and disk addresses. In section 4.1, we described



(a) Nodes M and X are full.

(b) Insertion of value 31 causes node M to overflow.

(c) Node M splits into M′ and N, posting into node X and causing it to overflow in turn.

(d) Node X splits into X′ and Y, posting into new root Z.

Figure 4.1: B+ tree reorganisation on node overflow.

the B+ tree in terms of predicates — containment of a point in an interval — requiring us explicitly to interpret these predicates from their decoded representation. Because we wish to describe structures in terms of local and global predicates, interpretation of this kind will frequently be required, but we wish to avoid cluttering the discussion. We describe our approach to this here.

4.2.1 Integrating predicate interpretation into the store

In general, we wish to be able to describe nodes in terms of local predicates associated with page pointers, as we suggested in section 2.4, and to do so independently of any specific decoded representation. To do this we introduce the function `interpret`.

`interpret` takes the decoded node representation returned by the store, and constructs from it the logical node structure containing local predicates and associated page pointers. We cannot provide a general implementation of `interpret`, because implementations are, by nature, representation-specific. We can, however, provide examples of the function's operation. Consider a B+ tree, whose internal nodes, as returned by the store, describe an interleaved list of key values and disk page addresses:

$$[a_0 : \mathit{PageId},\ k_1 : \mathit{Key},\ a_1 : \mathit{PageId}, \ldots,\ k_n : \mathit{Key},\ a_n : \mathit{PageId}]$$

The interpreted node structure pairs each page address with a predicate interval, deduced from the interleaved key values. We have:

$$\begin{aligned} & \texttt{interpret}\,([a_0 : \mathit{PageId},\ k_1 : \mathit{Key},\ a_1 : \mathit{PageId}, \ldots,\ k_n : \mathit{Key},\ a_n : \mathit{PageId}]) \\ = \quad & [\langle(\lambda x. -\infty < x < k_1), a_0\rangle, \langle(\lambda x. k_1 \leqslant x < k_2), a_1\rangle, \ldots \langle(\lambda x. k_n \leqslant x < \infty), a_n\rangle] \end{aligned}$$

A different approach to interpretation is taken in the case of the BANG file. Figure 4.2 reprises the exploded BANG file decomposition first presented in figure 2.7. Suppose that rectangles F, G, H, J and K are associated respectively with page pointers f, g, h, j and k, and that these form

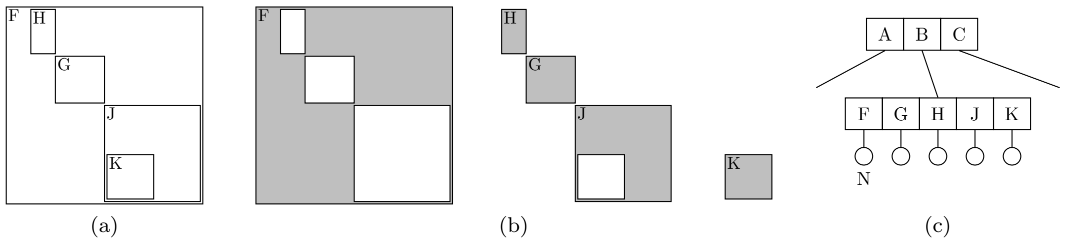


Figure 4.2: A BANG file style holey region decomposition and the associated tree fragment.

the contents of a single, physical BANG file internal node:

$$[\langle \mathrm{F}, f\rangle, \langle \mathrm{G}, g\rangle, \langle \mathrm{H}, h\rangle, \langle \mathrm{J}, j\rangle, \langle \mathrm{K}, k\rangle]$$

Considering the five rectangles collectively permits interpretation of the five holey regions described in figure 4.2b; these form the basis of the local predicates returned by `interpret`:

$$\begin{aligned} & \texttt{interpret}\,([\langle \mathrm{F}, f\rangle, \langle \mathrm{G}, g\rangle, \langle \mathrm{H}, h\rangle, \langle \mathrm{J}, j\rangle, \langle \mathrm{K}, k\rangle]) \\ = \; & [\langle(\lambda x.x \in (\mathrm{F} \setminus (\mathrm{G} \cup \mathrm{H} \cup \mathrm{J})), f\rangle, \langle(\lambda x.x \in \mathrm{G}), g\rangle, \\ & \quad \langle(\lambda x.x \in \mathrm{H}), h\rangle, \langle(\lambda x.x \in (\mathrm{J} \setminus \mathrm{K}), j\rangle, \langle(\lambda x.x \in \mathrm{K}), k\rangle] \end{aligned}$$

As we observed in section 3.2.1, `interpret` bears some similarity to the GiST's **Decompress** function; indeed we suggested that appropriate implementations might permit use of BANG-style region descriptions in the GiST. The principal difference is in intention — GiST compression is intended to provide efficient storage of what we might call a partially-interpreted node structure, while our purpose is to extract a fully-interpreted structure from its decoded representation.

In our description of the GiST we observed that, while the insertion algorithm presented in figure 3.5 does not make calls to **Decompress**, such calls could be integrated into the store's read operation. We go one step further here: we will assume from now on that `interpret` is integrated into the store's read operations. A read from the store therefore returns directly an interpreted node structure of the kind:

$$[\langle P_1 : \mathit{LocalPredicate}, a_1 : \mathit{PageId}\rangle, \ldots, \langle P_n : \mathit{LocalPredicate}, a_n : \mathit{PageId}\rangle]$$

Our picture of reading from the store is therefore as follows. After receiving a request for a given node address, the store retrieves the necessary bytes from disk, decodes them into some tree-specific representation, interprets that representation to yield a node of the structure described above, and returns that node.

Similarly, when writing a node to the store, we will assume that the store translates the interpreted structure into the appropriate tree-specific representation, before encoding it on disk. The necessary 'reverse interpretation' is the inverse of `interpret`, and we do not describe it in further detail here, other than to observe that it bears the same similarity to the GiST's **Compress** as does `interpret` to **Decompress**.

4.2.2 Predicate reinterpretation

Observe that the local predicates returned from a call to `interpret`, *i.e.* those present in a node returned by a read from the store, depend solely on the contents of that node's decoded representation. This is not an issue when each node is considered in isolation, but if the contents of that node change, for example when a new entry is posted into the node, the predicates that should be interpreted from the entries in the node may also change.

Suppose that a point inserted into the BANG file of figure 4.2c causes node N, the child of entry F, to overflow. A split boundary, L, is decided for the region, and let us suppose that it encloses regions G and H. This split boundary, if interpreted as a predicate at this point, would yield simply $\lambda x.x \in \mathrm{L}$. When posted into the parent node, however, two changes take place: firstly, the predicate interpreted for the nascent child of L is now no longer $\lambda x.x \in \mathrm{L}$ but $\lambda x.x \in (\mathrm{L} \setminus (\mathrm{G} \cup \mathrm{H}))$ and secondly, that interpreted for the remnant of node N is no longer $\lambda x.x \in (\mathrm{F} \setminus (\mathrm{G} \cup \mathrm{H} \cup \mathrm{J}))$, but rather $\lambda x.x \in (\mathrm{F} \setminus (\mathrm{J} \cup \mathrm{L}))$.

The practical effect of this is that, despite hiding the abstraction of `interpret` inside the store, some instances will remain in which we must explicitly *reinterpret* entries' predicates. We emphasise that this is not a consequence of our having hidden `interpret`, but rather is an abstraction of the genuinely changing interpretation of entries as they move around a tree. For convenience, we will continue to hide `interpret` in the store, but where necessary will make explicit calls to another function `reinterpret`. This function, given two lists of interpreted entries $\mathbf{E}_1$ and $\mathbf{E}_2$, returns a new list of interpreted entries, $\mathbf{E}_1'$:

$$\mathbf{E}_1' = \texttt{reinterpret}\,(\mathbf{E}_1, \mathbf{E}_2)$$

The disk address components of the elements of $\mathbf{E}_1'$ are exactly the same as those of the elements of $\mathbf{E}_1$, but the associated local predicates are those that would have been interpreted for those disk addresses if $\mathbf{E}_1' \oplus \mathbf{E}_2'$ had been interpreted from a single node (where $\mathbf{E}_2' = \texttt{reinterpret}\,(\mathbf{E}_2, \mathbf{E}_1)$).

4.2.3 Global predicate disjointness

When in section 2.4.3 we introduced the notion of global and local predicates, we remarked that the construction of a global predicate as a conjunction of local predicates permits disjointness of global predicates to be induced, by ensuring that local predicates are disjoint.

Figure 4.3 shows a three-level fragment of an external tree. Node L's local (and global) predicate is $\lambda x.(x \in \mathrm{A})$, but the local and global predicates of nodes M and N are as follows:

$$\begin{aligned}
P_{\mathrm{M}} &= \lambda x.x \in D \\
\mathscr{P}_{\mathrm{M}} &= \lambda x.(x \in A \wedge x \in D) \\
&= P_{\mathrm{L}} \wedge P_{\mathrm{M}} \\
P_{\mathrm{N}} &= \lambda x.x \in E \\
\mathscr{P}_{\mathrm{N}} &= \lambda x.(x \in A \wedge x \in E) \\
&= P_{\mathrm{L}} \wedge P_{\mathrm{N}}
\end{aligned}$$

Note that, in both cases, the nodes' global predicates are formed of the conjunction of their respective local predicates and P_{L}; clearly, if we require $\mathscr{P}_{\mathrm{M}} \parallel \mathscr{P}_{\mathrm{N}}$, it is sufficient that $P_{\mathrm{M}} \parallel P_{\mathrm{N}}$.

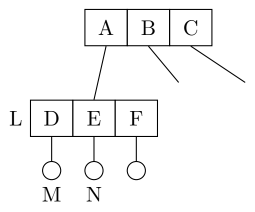
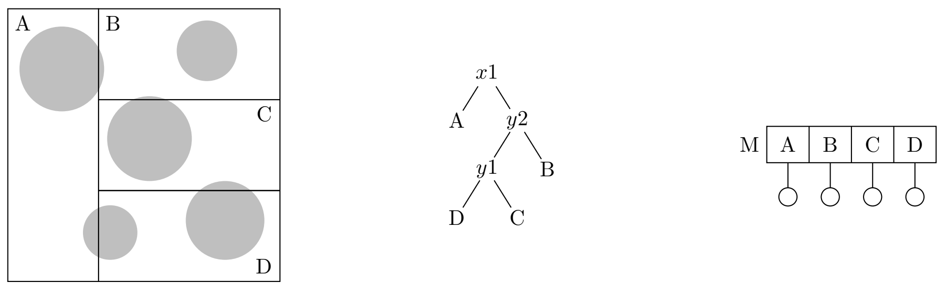


Figure 4.3: Three-level fragment of an external tree structure.



(a) Spatial decomposition showing locations of data clusters

(b) kd-tree representation of decomposition

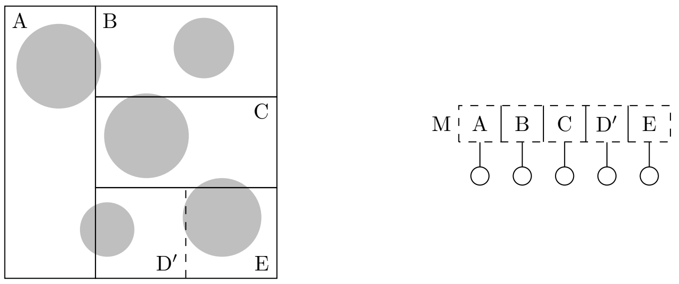
(c) External tree indexing the space as a collection of rectangles

Figure 4.4: Decomposition of a two-dimensional space using both dimensions for partitioning.

4.3 Region description in more than one dimension

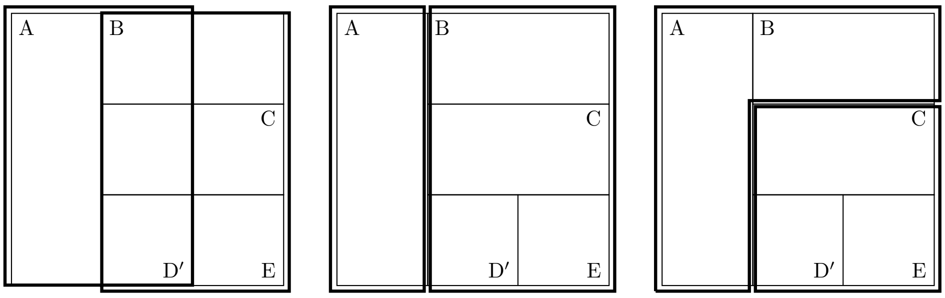
We saw in section 4.1 that the one-dimensional ordering of a B+ tree internal node's keys is used to imply a disjoint predicate for each of its children. In spaces of higher dimensionality, where no such natural ordering exists, achieving disjointness in a node split is often only possible at the expense of losing another desirable property of the split. Effects of higher-dimensional spaces begin to become apparent in as few as two dimensions, and we proceed, without loss of generality, by examining only two dimensions, for convenience of representation and intuition. One way in which to index a two-dimensional space using a B+ tree is to slice it in one dimension only (effectively using the row-order mapping discussed in section 3.1.1), but as described in section 2.2 and shown in figure 2.1, this generally results in poor preservation of locality. Figure 4.4a shows a clustered data distribution and two-dimensional decomposition, described using the kd-tree in figure 4.4b and indexed by the tree given in figure 4.4c. Supposing the child of entry D to be overflowing, it is split into two, corresponding to regions D′ and E (shown in figure 4.5a), posting two new entries into node M to replace D and causing node M to overflow (figure 4.5b).

Before examining some possible partitionings of the list of entries [A,B,C,D′,E], we consider the issue of representation. The key issue in partitioning sets of entries is not that subsets forming disjoint regions of space cannot be found, but rather that they cannot efficiently be represented. For example, the regions A ∪ C ∪ E and B ∪ D′ are disjoint (although not necessarily formed of contiguous parts), but neither can be described using a simple representation that contains all its components without overlapping the other region. To represent either region accurately may require each of its components to be represented explicitly, requiring an index term whose combined size may still approach the capacity of a single node. Typically, an index structure uses a single mode of representation; for example 'rectangles' or 'spheres' or 'kd-trees', and describes partitions solely using that mode. This has profound effects on selection of partitions; some of these effects are illustrated in figure 4.6 and are described below and in section 4.4.



(a) Region D splits into D′ and E. (b) Replacement of entry D with entries D′ and E causes node M to overflow.

Figure 4.5: Further decomposition of the space in figure 4.4, resulting in overflow of node M.



(a) Overlapping (b) Poorly balanced (c) Non-rectangular

Figure 4.6: Approaches to partitioning a set of two-dimensional regions into two subsets.

4.3.1 Overlap

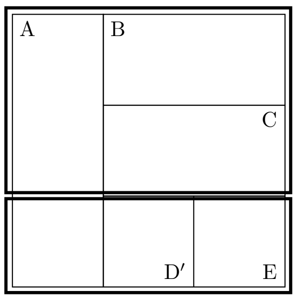
Figure 4.6a shows partitioning of the five rectangles into subsets [A, D′] and [B, C, E], describing each partition with a bounding rectangle. As a result, the two bounding rectangles overlap, the split is not disjoint, and the single path property cannot be maintained. This is a common approach, taken by the class of non-SPP structures described in section 3.2 and the Hybrid tree of section 3.3.4.

4.3.2 Poor split balance

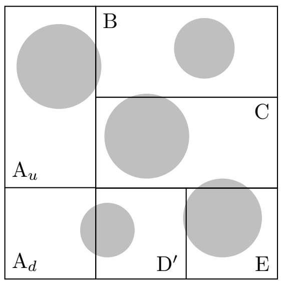
Figure 4.6b shows partitioning of the five rectangles into subsets [A] and [B, C, D′, E], describing each partition with a bounding rectangle. The choice of partitioning is such that the split is disjoint, preserving the SPP but at a cost of poor split balance. The result of this is to allow the formation of underoccupied nodes and to allow the tree's fanout ratio to fall below an acceptable minimum. The LSD- and buddy-trees take this approach.

4.3.3 Holey splitting

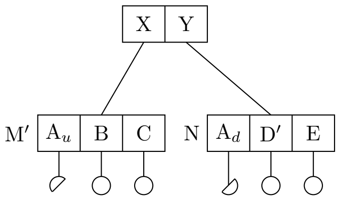
Figure 4.6c shows partitioning of the five rectangles into subsets [A, B] and [C, D′, E], by splitting a corner from the space, as in the hB-tree. As described in sections 3.3.5 and 3.3.6, splitting approaches such as these require either constrained partitioning choices (as in the BANG file's spanning splits or the hB-tree's splitting at decorations) or may result in index terms with an uncertain upper size bound (as when posting full paths in the hB-tree). The former approach may



(a) Imposition of partition boundaries requires region A to split.



(b) Split of region A yields regions A$_d$ and A$_u$.



(c) Forced split of the child of A places an entry in both node M′ and node N.

Figure 4.7: Forced split of region A on partitioning node M's entries

produce poorly-balanced splits, while the latter reduces occupancy in the parent node, with either effect risking node underoccupancy.

4.4 Forced splitting

Forced splitting in external file access methods was introduced in the K-D-B tree, which we discussed in section 3.3.1. The advantage of forced splitting is that desirable local properties of a node split can all be provided: disjoint partitioning in a cyclically-chosen splitting dimension with good balance. In this section we consider a general approach to forced splitting.

Figure 4.7a shows the superposition of two partitioning rectangles on the space, neither of which contains subregion A. The approach here, as in the K-D-B tree, is to choose the best possible partitioning while postponing consideration of 'difficult' entries, then to split each of the remaining entries forcibly between the two partitioning regions. In our example, we select partitions [B, C] and [D′, E], splitting entry A forcibly, resulting in a split ratio of 3:3. This is a perfectly balanced split, into partitions [A$_u$, B, C] and [A$_d$, D′, E] (see figure 4.7b), achieving a disjoint spatial partitioning and by extension the SPP. This behaviour is exhibited by the K-D-B tree and BANG file and, as in those structures, forced splits must propagate downwards in order to preserve the disjoint recursive partitioning of the space, splitting further entries in lower levels of the tree as required to achieve this. We describe structures exhibiting this behaviour as *forced splitting trees* or *FS-trees*.

Section 4.4.1 presents an algorithm for insertion into a general FS-tree, and in section 4.4.2 we discuss briefly the effects of a forced splitting approach.

4.4.1 FS-tree insertion

We now present an algorithm for insertion into a general FS-tree. Insertion proceeds by selecting, in each internal node, the unique entry that contains the point undergoing insertion (recall that FS-trees exhibit the SPP) and descending into that entry's child. When a leaf node overflows, its contents, $\mathbf{E}$, are split into two new lists, $\mathbf{E_L}$ and $\mathbf{E_R}$, with associated predicates p_L and p_R, using a function `splitL`:

$$\texttt{splitL}\,(\mathbf{E}) = \langle p_L, \mathbf{E_L}, p_R, \mathbf{E_R} \rangle$$

with the following properties (given the parent entry of the splitting node, $\langle p, \_ \rangle$):

$$\begin{aligned}
&p_L \vee p_R = p\\
&p_L \wedge p_R = \texttt{false}\\
&\text{elems}\,\mathbf{E_L} = \{\langle k, v\rangle \mid \langle k, v\rangle \in \text{elems}\,\mathbf{E} \wedge p_L(k)\}\\
&\text{elems}\,\mathbf{E_R} = \{\langle k, v\rangle \mid \langle k, v\rangle \in \text{elems}\,\mathbf{E} \wedge p_R(k)\}
\end{aligned}$$

An analogous function, `splitI`, is defined to partition the contents of an overflowing internal node, however, as described in the introduction to this section, there may exist 'difficult' entries whose predicates are not disjoint from either split predicate, *i.e.* any entry $\langle p, s\rangle$ for which $p \nparallel p_L \wedge p \nparallel p_R$. Bearing in mind that local predicate disjointness implies disjointness of the associated global predicates, this is therefore a statement of the forced split criterion introduced in chapter 3: these entries must be split forcibly. Consequently, `splitI` also returns, in a separate list $\mathbf{F}$, those entries requiring a forced split:

$$\texttt{splitI}\,(\mathbf{V}) = \langle p_L, \mathbf{V_L}, p_R, \mathbf{V_R}, \mathbf{F}\rangle$$

Properties of `splitI` are:

$$\begin{aligned}
&p_L \vee p_R = p\\
&p_L \wedge p_R = \texttt{false}\\
&\text{elems}\,\mathbf{V_L} = \{\langle p, s\rangle \mid \langle p, s\rangle \in \text{elems}\,\mathbf{V} \wedge p \subseteq p_L\}\\
&\text{elems}\,\mathbf{V_R} = \{\langle p, s\rangle \mid \langle p, s\rangle \in \text{elems}\,\mathbf{V} \wedge p \subseteq p_R\}\\
&\text{elems}\,\mathbf{V_L} \cup \text{elems}\,\mathbf{V_R} \subseteq \text{elems}\,\mathbf{V}\\
&\text{elems}\,\mathbf{F} = \{e \mid e \in \mathbf{V} \wedge e \notin \mathbf{V_L} \wedge e \notin \mathbf{V_R}\}
\end{aligned}$$

After splitting a leaf node, or after splitting an internal node for which $\mathbf{F}$ is empty (no entries require splitting forcibly), entries are posted into the level above in the usual way. If, however, $\mathbf{F}$ is not empty, each entry, $\langle p, s\rangle$, in $\mathbf{F}$ must be split forcibly as follows:

- two new predicates, $p \wedge p_L$ and $p \wedge p_R$ are calculated, between which the contents of the node stored on page s must be split;
- page s' is allocated, to allow the results of partitioning the contents of s to be written into pages s and s' respectively;
- new entry $\langle p \wedge p_L, s\rangle$ is added to $\mathbf{V_L}$, and $\langle p \wedge p_R, s\rangle$ is added to $\mathbf{V_R}$;
- a 'forced split tuple', *fsTuple*, $\langle p \wedge p_L, s, p \wedge p_R, s'\rangle$ is cached. This provides sufficient information to execute the forced split later; the page at address s requires a forced split, the predicates $p \wedge p_L$ and $p \wedge p_R$ describe that split, and the results of the split will be written onto either page s or s'.

The results of the split of this node can now be written to disk, but entries cannot be posted up until all the list of cached forced split tuples is empty. For each *fsTuple*, $\langle p \wedge p_L, s, p \wedge p_R, s'\rangle$, page s is read, its contents are partitioned using predicates $p \wedge p_L$ and $p \wedge p_R$, and the resulting partitions written onto pages s and s' respectively. Partitioning may require the execution of further forced splits which are, as described here, initially identified and cached before later being executed.

An FS-tree insertion configuration has the form:

$$\langle \mathtt{C}\,,\, r\,,\, \pi\,,\, \sigma \rangle$$

where `C` is a command, r is a page identifier, π is a stack of page identifiers and σ is the store. The grammar for the insertion command is:

$$\begin{array}{lcl} \mathit{fsInsertCommand} & ::= & \mathtt{ins}\,(\mathit{LeafEntry}) \\ & | & \mathtt{S} \\ & | & \mathtt{D}\,([\langle \mathit{LocalPredicate}, \mathit{PageId}\rangle]) \\ & | & \mathtt{spl}\,([\mathit{Entry}], [\mathit{fsTuple}]) \end{array}$$

As previously, the `ins` command is used to descend to the leaf, and the `S` command to return to the root when no posting is required. Note that, as in the GiST implementation in figure 3.5, the `D` command for posting entries contains, for convenience, a sequence of entries of length 2; the `spl` command holds this sequence of entries so that they can be posted into the level above after all forced splits cached as *fsTuple*s have been executed. We use the constants Max_L and Max_I to describe, respectively, the capacity of leaf and internal nodes as a number of entries.

The set of state transitions for insertion into an FS-tree is given in figure 4.8. Transition 3.1 bootstraps the insert operation, rewriting the external `fsInsert` command, which carries a leaf entry for insertion and the address of the root node, into an `ins` command for tree descent using transition 3.2. Transitions 3.3 – 3.5 handle cases of insertion into the leaf, while transitions 3.6 and 3.7 return from an internal node, posting no further entries.

Notice in transition 3.7 that, before the incoming entries replace the single entry that was the parent of the splitting node, both the node contents, $\mathbf{V}$, and the posted pair of entries, $[\langle p_L, r_L\rangle, \langle p_R, r_R\rangle]$, are reinterpreted with respect to one another. Transition 3.8 requires similar reinterpretation after entry posting, after which it splits an overflowing internal node, perhaps initiating forced splits which are executed using transitions 3.9 and 3.10, before upward posting can take place in transition 3.11. Insertion terminates in a split of the root of the FS-tree in transition 3.12, or not, in transition 3.13, rewriting into an external `fs` command containing the (possibly new) FS-tree root.

4.4.2 Loss of guaranteed minimum fanout

The advantage of the FS-tree approach is that enforcement of local predicate disjointness at node splitting provides FS-trees with the SPP. Forced splitting makes this possible while preserving local split balance, the intention being to maintain a global minimum fanout ratio. A problem, however, is that forced split of entries in the splitting node occurs without regard to the contents of the subtrees rooted on those entries. Figure 4.7b shows the geometry of the forced split of region A and suggests that most of its points are in A_u, illustrating that, although the split of node M has excellent balance, that of the child of entry A probably does not. The situation could have been still worse had A been an internal node; figure 4.9a shows a decomposition of the space using a tree of two levels, a fragment of which is shown in figure 4.9b. Applying the same forced split

$\big\langle \texttt{fsInsert}\,(\langle k,v\rangle, r)\,,\,\sigma\big\rangle \quad\rightsquigarrow\quad \big\langle \texttt{ins}\,(\langle k,v\rangle)\,,\,r\,,\,[\,]\,,\,\sigma\big\rangle$ (3.1)

$\big\langle \texttt{ins}\,(\langle k,v\rangle)\,,\,r\,,\,\pi\,,\,\sigma\big\rangle \quad\rightsquigarrow\quad \big\langle \texttt{ins}\,(\langle k,v\rangle)\,,\,s\,,\,r::\pi\,,\,\sigma\big\rangle$ (3.2)

if $\sigma(r) = \texttt{I}\,(\mathbf{V})$

where $\langle \_, s\rangle = \texttt{uniq}\,(\mathbf{V}, P)$ and $P = \lambda x.\text{match } x \text{ as } \langle q,t\rangle \text{ in } q(k)$

$\big\langle \texttt{ins}\,(\langle k,v\rangle)\,,\,r\,,\,\pi\,,\,\sigma\big\rangle \quad\rightsquigarrow\quad \big\langle \texttt{S}\,,\,r\,,\,\pi\,,\,\sigma[r \mapsto \texttt{L}\,(\texttt{repl}\,(\langle k,v\rangle, i, \mathbf{E}))]\big\rangle$ (3.3)

if $i \leqslant |\mathbf{E}|$

where $\sigma(r) = \texttt{L}\,(\mathbf{E})$

and $i = \texttt{first}\,(\mathbf{E}, P)$ and $P = \lambda x.\text{match } x \text{ as } \langle j,w\rangle \text{ in } j = k$

$\big\langle \texttt{ins}\,(\langle k,v\rangle)\,,\,r\,,\,\pi\,,\,\sigma\big\rangle \quad\rightsquigarrow\quad \big\langle \texttt{S}\,,\,r\,,\,\pi\,,\,\sigma[r \mapsto \texttt{L}\,(\texttt{append}\,(\langle k,v\rangle, \mathbf{E}))]\big\rangle$ (3.4)

if $|\mathbf{E}| < Max_L$

where $\sigma(r) = \texttt{L}\,(\mathbf{E})$

$\big\langle \texttt{ins}\,(\langle k,v\rangle)\,,\,r_L\,,\,\pi\,,\,\sigma\big\rangle \quad\rightsquigarrow$ (3.5)

$\big\langle \texttt{D}\,([\langle p_L, r_L\rangle, \langle p_R, r_R\rangle])\,,\,r_L\,,\,\pi\,,\,\sigma[r_L \mapsto \texttt{L}\,(\mathbf{E_L})\,, r_R \mapsto \texttt{L}\,(\mathbf{E_R})]\big\rangle$

where $\sigma(r_L) = \texttt{L}\,(\mathbf{E})$

and $\langle p_L, \mathbf{E_L}, p_R, \mathbf{E_R}\rangle = \texttt{splitL}\,(\texttt{append}\,(\langle k,v\rangle, \mathbf{E}))$

and $r_R = \texttt{fresh}(\sigma)$

$\big\langle \texttt{S}\,,\,r\,,\,s::\pi\,,\,\sigma\big\rangle \quad\rightsquigarrow\quad \big\langle \texttt{S}\,,\,s\,,\,\pi\,,\,\sigma\big\rangle$ (3.6)

$\big\langle \texttt{D}\,(\mathbf{E})\,,\,r_L\,,\,s::\pi\,,\,\sigma\big\rangle \quad\rightsquigarrow\quad \big\langle \texttt{S}\,,\,s\,,\,\pi\,,\,\sigma[s \mapsto \texttt{I}\,(\mathbf{V}'')]\big\rangle$ (3.7)

if $|\mathbf{V}| < Max_I$

where $\sigma(s) = \texttt{I}\,(\mathbf{V})$

and $\mathbf{V}' = \texttt{del}\,(\texttt{first}\,(\mathbf{V}, P), \mathbf{V})$ and $P = \lambda x.\text{match } x \text{ as } \langle v,t\rangle \text{ in } t = r_L$

and $\mathbf{V}'' = \texttt{reinterpret}\,(\mathbf{V}', \mathbf{E}) \oplus \texttt{reinterpret}\,(\mathbf{E}, \mathbf{V}')$

$\big\langle \texttt{D}\,(\mathbf{E})\,,\,r_L\,,\,s_L::\pi\,,\,\sigma\big\rangle \quad\rightsquigarrow$ (3.8)

$\big\langle \texttt{spl}\,([\langle q_L, s_L\rangle, \langle q_R, s_R\rangle], \mathbf{F}')\,,\,s_L\,,\,\pi\,,\,\sigma[s_L \mapsto \texttt{I}\,(\mathbf{V'_L})\,, s_R \mapsto \texttt{I}\,(\mathbf{V'_R})]\big\rangle$

where $\sigma(s_L) = \texttt{I}\,(\mathbf{V})$

and $\mathbf{V}' = \texttt{del}\,(\texttt{first}\,(\mathbf{V}, P), \mathbf{V})$ and $P = \lambda x.\text{match } x \text{ as } \langle v,t\rangle \text{ in } t = r_L$

and $\langle q_L, \mathbf{V_L}, q_R, \mathbf{V_R}, \mathbf{F}\rangle = \texttt{splitI}\,(\texttt{reinterpret}\,(\mathbf{V}', \mathbf{E}) \oplus \texttt{reinterpret}\,(\mathbf{E}, \mathbf{V}'))$

and $\mathbf{F}' = [\ \langle q_L \wedge p, s, q_R \wedge p, s'\rangle \ \mid \langle p, s\rangle \in \mathbf{F} \wedge s' = \texttt{fresh}(\sigma)\,]$

and $\mathbf{V'_L} = \mathbf{V_L} \oplus [\ \langle p, s\rangle \ \mid \langle p, s, \_, \_\rangle \in \mathbf{F}'\,]$

and $\mathbf{V'_R} = \mathbf{V_R} \oplus [\ \langle p, s\rangle \ \mid \langle \_, \_, p, s\rangle \in \mathbf{F}'\,]$

and $s_R = \texttt{fresh}(\sigma)$

$\big\langle \texttt{spl}\,(\mathbf{P}, \langle p_L, s_L, p_R, s_R\rangle :: \mathbf{F})\,,\,r\,,\,\pi\,,\,\sigma\big\rangle \quad\rightsquigarrow$ (3.9)

$\big\langle \texttt{spl}\,(\mathbf{P}, \mathbf{F})\,,\,r\,,\,\pi\,,\,\sigma[s_L \mapsto \texttt{L}\,(\mathbf{E_L})\,, s_R \mapsto \texttt{L}\,(\mathbf{E_R})]\big\rangle$

where $\sigma(s_L) = \texttt{L}\,(\mathbf{E})$

and $\mathbf{E_L} = [\ \langle k, v\rangle \ \mid \langle k, v\rangle \in \mathbf{E} \wedge p_L(k)\,]$

and $\mathbf{E_R} = [\ \langle k, v\rangle \ \mid \langle k, v\rangle \in \mathbf{E} \wedge p_R(k)\,]$

and $s_R = \texttt{fresh}(\sigma)$

$\big\langle \texttt{spl}\,(\mathbf{P}, \langle p_L, s_L, p_R, s_R\rangle :: \mathbf{F})\,,\,r\,,\,\pi\,,\,\sigma\big\rangle \quad\rightsquigarrow$ (3.10)

$\big\langle \texttt{spl}\,(\mathbf{P}, \mathbf{F}' \oplus \mathbf{F})\,,\,r\,,\,\pi\,,\,\sigma[s_L \mapsto \texttt{I}\,(\mathbf{V_L})\,, s_R \mapsto \texttt{I}\,(\mathbf{V_R})]\big\rangle$

where $\sigma(s_L) = \texttt{I}\,(\mathbf{V})$

and $\mathbf{F} = [\ \langle p_L \wedge p, s, p_R \wedge p, s'\rangle \ \mid \langle p, s\rangle \in \mathbf{V} \wedge p \not\parallel p_L \wedge p \not\parallel p_R \wedge s' = \texttt{fresh}(\sigma)\,]$

and $\mathbf{V_L} = [\ \langle p, s\rangle \ \mid \langle p, s\rangle \in \mathbf{V} \wedge p \subseteq p_L\,] \oplus [\ \langle p, s\rangle \ \mid \langle p, s, \_, \_\rangle \in \mathbf{F}\,]$

and $\mathbf{V_R} = [\ \langle p, s\rangle \ \mid \langle p, s\rangle \in \mathbf{V} \wedge p \subseteq p_R\,] \oplus [\ \langle p, s\rangle \ \mid \langle \_, \_, p, s\rangle \in \mathbf{F}\,]$

and $s_R = \texttt{fresh}(\sigma)$

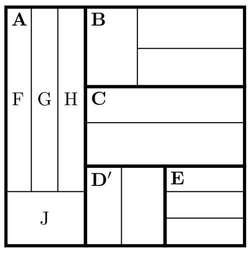
$\big\langle \texttt{spl}\,(\mathbf{P}, [\,])\,,\,r\,,\,\pi\,,\,\sigma\big\rangle \quad\rightsquigarrow\quad \big\langle \texttt{D}\,(\mathbf{P})\,,\,r\,,\,\pi\,,\,\sigma\big\rangle$ (3.11)

$\big\langle \texttt{S}\,,\,r\,,\,[\,]\,,\,\sigma\big\rangle \quad\rightsquigarrow\quad \big\langle \texttt{fs}\,(r)\,,\,\sigma\big\rangle$ (3.12)

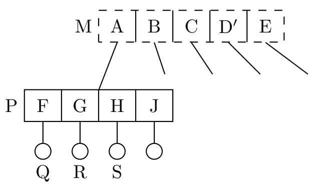
$\big\langle \texttt{D}\,(\mathbf{P})\,,\,r\,,\,[\,]\,,\,\sigma\big\rangle \quad\rightsquigarrow\quad \big\langle \texttt{fs}\,(s)\,,\,\sigma[s \mapsto \texttt{I}\,(\mathbf{P})]\,\big\rangle$ (3.13)

where $s = \texttt{fresh}(\sigma)$

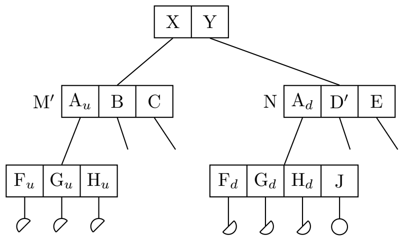
Figure 4.8: The FS-tree `fsInsert` algorithm.



(a) Subdivision of five index partitions.



(b) Node M is overflowing but to split must execute a forced split of entry A.



(c) Forced split of A leads to significant reorganisation.

Figure 4.9: Forced splits of internal nodes may cause significant reorganisations with uncertain effects on occupancy.

geometry in this case produces the tree shown in figure 4.9c, in which the forced split of entry A causes forced split of nodes P, Q, R and S.

In the worst case, a forced split could result in every node in a subtree being split forcibly, with a large proportion of those being underoccupied as a result. The implementation of figure 4.8 may actually write empty pages on occasions, but even an alternative implementation that optimised these out may be required to instantiate pages containing only a single entry. This presents a major disadvantage inherent in FS-trees; because forced split boundaries are entirely independent of the data distribution in affected subtrees, downward forced splitting can produce heavily underoccupied nodes. This reduces tree fanout in underoccupied branches, causing query performance to deteriorate, in the worst case reducing search efficiency from logarithmic to linear.

4.5 Virtual forced splitting

Reconsidering the example partitioning of figure 4.7a, we make two observations:

1. The proposed partitioning has excellent local balance; it is only in downward propagation of the forced split that occupancy guarantees are undermined;

2. Region A is contained in neither of the two disjoint regions proposed to partition the set of entries, and hence meets the forced split criterion; this is what led us to execute a forced split.

This leads us to the suggestion that, if region A could be treated differently so that we can make use of the proposed partitioning without executing the forced split, the situation might improve. More specifically, if the semantics of the forced split of region A could be reproduced using a more local reorganisation, we might avoid the disadvantageous effects of wider split propagation.

An alternative approach, first proposed in Freeston's BV-tree, is to *virtualise* the forced split. The purpose of forced splitting is to separate a subtree into the two portions that belong on either side of a split boundary; this can also be achieved by, at the same time as promoting the parent entries resulting from a node split, *elevating* entries meeting the forced split criterion into the level above. Elevated entries can then, during search execution, be carried into either of the subtrees into which they would have otherwise been forced split, making them available to the search as in the forced split case. We refer to the posting of entries requiring a forced split as *elevation* to distinguish such entries from the parent entries *promoted* at node split; non-elevated entries are

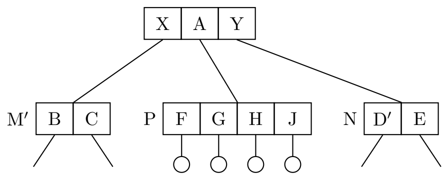


Figure 4.10: The forced split executed in figure 4.9b is virtualised by elevation of entry A into the root node.

referred to as *primary* entries (Freeston refers to elevated entries as *guards*). The result of handling the overflow in figure 4.9b in this way is shown in figure 4.10.

This immediately presents the issue of how to execute a search in a tree with elevated entries. It is not correct simply to treat an elevated entry as a 'normal', primary entry, because the region it describes is not disjoint from those described by the primary entries in the node in which the elevated entry is physically located. This is a consequence of the fact that the tree itself no longer represents directly a recursive decomposition of the space; separation of the decomposition from its physical representation is a phenomenon described by Samet [51] as *decoupling*. In fact, the contents of a subtree rooted on an elevated entry really belong distributed between the subtrees under the primary entries across which it is virtually split; in general, an elevated entry can only correctly be interpreted in the level from which it was elevated, *i.e.* its *natural* level. There are a number of reasons for this:

- In some structures, for example those employing holey-splitting like the BANG file or the BV-tree, a node's local predicate is interpreted from information stored, not only in its parent entry, but also in other entries. Elevation of a given subtree's parent entry separates it from these other entries — correct interpretation of the local predicate therefore requires an elevated entry to be brought down into a position in which it can be considered alongside other relevant entries. Figure 4.11a shows two regions, A and B. B is a hole in A, so region A does not consist of all the space contained in its outer boundary, but excludes the region contained in B. If the entry representing region A's outer boundary were to be elevated and separated from that of B, consideration of A *in situ* would suggest that descent of the associated subtree is required to answer a search for q_1, when in fact it is not.

- Structures like the M-tree employ local predicates that are not contained by the corresponding global predicate. These structures rely on the confining effect of implicit construction of the global predicate as a conjunction of local predicates — consideration of such local predicates without first confining them can lead to error. Figure 4.11b shows two M-tree regions, C = $\langle O_C, r_C \rangle$ and D = $\langle O_D, r_D \rangle$. Suppose that C is the region component of D's parent entry, but that in a (hypothetical) virtual forced splitting M-tree, D has been elevated into the same node as C. A search for point q_2 clearly is not required to descend the subtree rooted on C, but *in situ* consideration of D suggests that the subtree rooted on D is a candidate. D's global predicate, however, is really $\lambda x.(d(x, O_C) \leqslant r_C \wedge d(x, O_D) \leqslant r_D)$, so this is not the case.

- As we described above, the contents of the subtree rooted on an elevated entry really belong partially in each of the subtrees across which it is virtually split. This means that the node should really be considered to be the union of more than one virtual node, each of which has a global predicate of its own. We will consider this in greater detail in chapter 5. As

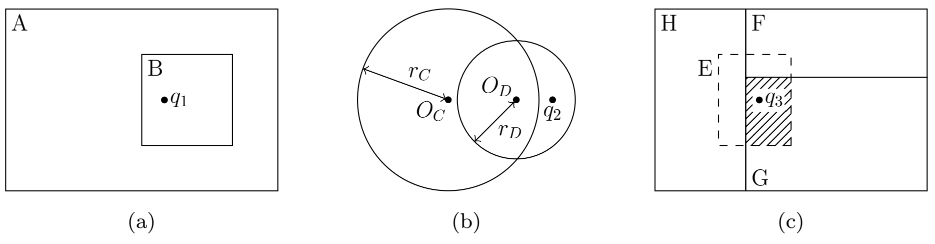


Figure 4.11: Effects of VFS-tree region representation on interpretation of predicates.

in the case of M-tree-like structures, that global predicate can only be identified by carrying the entry into the appropriate subtree, to confine the local predicate to the region indexed by that subtree. Figure 4.11c shows region E, virtually split across regions F, G and H. A search for region q_3 is required to visit the child of the entry associated with E, but it is only the portion of E intersecting G that we require (shaded in the diagram). This is identified by carrying E into the subtree rooted on G, allowing construction of the correct global predicate. We refer to the exclusion of entries from a search of a physical elevated node because they fall outside the relevant virtual node as *filtering*.

For these reasons, during a search of the tree, an elevated entry cannot be considered as a potential branch in the search path at the level in which it occurs physically, but must merely be picked up and carried along as the search progresses down the tree, until the search arrives at the elevated entry's natural level. At this point we say that the elevated entry 'reaches its primary level' or 'becomes primary', and can now be considered as if it had never been elevated from there in the first place. In figure 4.10, this means carrying entry A into the subtree of either entry X or Y pending its consideration as required (we refer to a set of elevated entries carried down the tree in this way as a *pending set*; Freeston calls this a *guard set*). This reconstructs the semantics of the forced split of entry A on-the-fly, and so we also refer to A as being *virtually split* and call trees of this kind *virtual forced split trees* (VFS-trees).

4.5.1 Pending sets and predicate reinterpretation

In FS-trees, we used `reinterpret` to determine the, possibly revised, predicate associated with a node's children after new entries have been posted into the node. Using pending sets to carry entries around the tree for interpretation elsewhere requires a similar approach.

In the discussion at the start of section 4.5, we postulated a situation in which region A of figure 4.11a was elevated out of a node in which region B was represented. Consider the subsequent situation in which the node into which A was elevated is read from disk. The predicate interpreted from A when reading the node from the store could not make reference to region B at all, since B is not also elevated. The predicate interpreted from A in its elevated position might, therefore, be as simple as $\lambda x.x \in \mathrm{A}$. If, however, it were carried, in a pending set, into the node containing B, that predicate would need to be reinterpreted to yield $\lambda x.x \in (\mathrm{A} \setminus \mathrm{B})$.

We present a number of VFS-tree algorithms in chapter 5: these will make considerable use of `reinterpret` to make sense of predicates as entries move around the tree.

4.5.2 The KDB-VFS tree

We introduce, very briefly, the KDB-VFS tree; a VFS-analogue of the forced splitting K-D-B tree. Additional description appears in chapters 6 and 7, but we introduce it here as an example of the class of VFS-structures that is conceptually simpler than the BV-tree. This is principally on grounds of region representation: K-D-B regions are represented explicitly in each entry, enabling consideration of features of virtual forced splitting without further complicating the discussion with the issues of local predicate interpretation presented by the BV-tree. Beyond this we make no assertions for the time being about the usefulness of the KDB-VFS tree as a practical VFS-tree implementation.

Insertion into a KDB-VFS tree proceeds as in the K-D-B case until an internal node split is executed. At this stage, entries meeting the forced split criterion are instead elevated into the parent of the splitting node.

4.5.3 Entry and node level numbers

To assist the identification of elevated entries in VFS-trees, each internal node entry is given a level number, corresponding to the length of the longest path from the entry's child to a leaf node in its subtree. An entry whose child is a leaf node is of level 0, and the level number of an entry whose child is an internal node is one greater than the highest-level entry in that child. Each internal node is also given a level number, equal to that of the highest-level entry stored in the node; for convenience we represent this explicitly. Elevated entries can thus be identified as those of a lower level number than that of the node in which they appear.

The structure of an interpreted VFS-node is as follows:

$$\begin{aligned} \mathit{Node} &::= \mathit{INode} \mid \mathit{LNode} \\ \mathit{INode} &::= \mathtt{I}\,(\mathit{NodeLevelNumber}, [\langle \mathit{EntryLevelNumber}, \mathit{LocalPredicate}, \mathit{PageId}\rangle]) \\ \mathit{LNode} &::= \mathtt{L}\,([\langle \mathit{Key}, \mathit{Value}\rangle]) \end{aligned}$$

We introduced in section 2.4.4 the convention, in diagrams, of taking an entry label to indicate the interpreted predicate component of a node entry, representing the address of its child using a line connecting the entry to that child. We extend that convention in VFS-trees to include the subscripting of predicates with entry level numbers; a VFS-entry $\langle l, P, r\rangle$ will be represented diagrammatically as an entry labelled P_l with a pointer to the node at address r. Figure 4.12 gives a VFS-tree fragment illustrating this convention, which we will observe from now on.

At the beginning of section 4.5, we described elevated entries as being virtually split *across* primary entries; for reasons that will become clear we will also describe elevated entries as being virtually split *at the level* of those primary entries, and therefore at the level number of the node in which they are physically located. Figure 4.12 shows level 0 entry D_0 located in the level 2 node N; entry D_0 is thus virtually split at level 2.

4.6 Issues with the VFS approach

Using elevation to avoid forced splitting presents a number of issues. Each of these affects the BV-tree as much as any other possible VFS-structure, but the BV-tree presentation of these issues is sometimes unclear or incomplete, precisely for the reason that we suggested in section 2.4.3: its

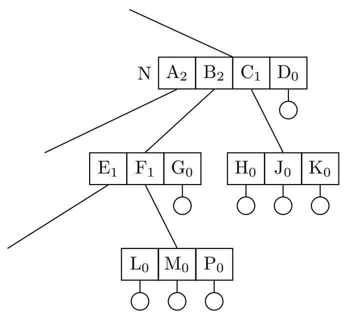


Figure 4.12: VFS-tree fragment showing entry level numbers.

operations are described in terms of data — individual entries in the tree — rather than in terms of the predicates that are interpreted from its entries collectively. We begin by identifying each of these issues before considering if and how they are to be addressed, in VFS-trees in general and in the BV-tree in particular.

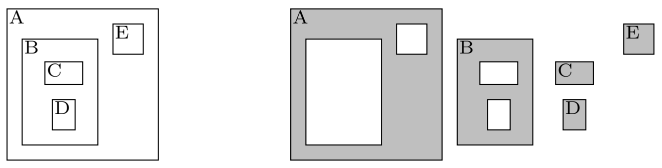
A common measure of the efficiency of stored representation in external hierarchical access methods is that of fanout — the average number of children (entries) in an index node. In a height-balanced tree, the fanout ratio is a statement of the number of leaves reachable from a node; the height of a subtree, h, and the average number of leaves in that subtree, N, are related directly by the fanout ratio, F:

$$h = \lceil \log_F N \rceil$$

In a VFS-node of average occupancy, a number of its F entries will be elevated entries, the subtrees of which are of lower height than its primary entries; these reduce the total number of leaves reachable from the node. To increase the number of leaves reachable from a VFS-node, therefore, we must seek to increase the average number of primary entries in a node while decreasing the number of elevated entries therein.

There are three factors that contribute to the number of elevated entries in a node. Firstly, *direct* elevation describes the elevation of entries accompanying the split of an overflowing node, and is examined in section 4.6.1. Secondly, *indirect* elevation occurs where split of the child of an elevated entry introduces a further elevated entry into a node; this is considered in section 4.6.2. The third contributing factor to the number of elevated entries in a node is the height of that node above the leaf level. We discuss this in section 4.6.3, before considering overall elevation limits in the face of the simultaneous contribution of all three factors in section 4.6.4.

Finally, we note that the sole published search algorithm for the BV-tree is that of exact match; the real advantage, however, of truly spatial structures like VFS-trees is their support for queries with non-zero spatial extent like Range and K-Nearest Neighbour (K-NN). As discussed in section 3.1, although one-dimensional mapping techniques do not preserve locality well, such approaches provide simple and efficient support of exact-match queries; a more complicated approach can only be justified by support of queries with extent. Furthermore, the published description of insertion in the BV-tree, while correct as far as it goes, does not quite give the full story. The key to correct implementation of all these algorithms is the selection of appropriate pending sets; while straightforward in our KDB-VFS example of exact-match search in figure 4.10, insertion into more complex structures like the BV-tree, or algorithms like queries with extent, require a clear understanding of the semantics of the VFS-tree. We introduce this in section 4.6.6 and will discuss it in detail in section 5.1.



(a) Holey region decomposition (b) Exploded to show true region extent

Figure 4.13: Holey regions described by a collection of boundaries.

4.6.1 Limiting direct elevation

The placing of a limit on direct elevation means providing a guaranteed upper bound on the number of entries in a splitting node that can require a virtual split. In the case of the forced splitting K-D-B tree, we cannot guarantee an upper limit on the number of entries in an overflowing node that might require a forced split (short of the number of entries in the node; as Samet [52] remarks, we should be careful not to choose such a partitioning), and analogously neither can we guarantee a limit on the number of entries in an overflowing KDB-VFS node that might require elevation.

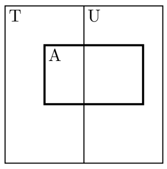
The BV-tree, however, provides the sort of upper limit we require. As described in section 3.4.2, the key to how it does this lies in its approach to region decomposition based on the removal of holes from regions, with a strict containment relationship between resulting regions: namely, that any two regions in a BV-tree decomposition must either be disjoint, or one must strictly contain the other.

When a BV-node overflows, a subregion of the node's region is identified which contains between 1/3 and 2/3 of the node's primary entries. Separation of the entries that fall within this subregion from those outside it corresponds to the formation of a hole in the outer region. As the nodes representing the outer region and those representing holes therein undergo further insertions and node splits, a spatial decomposition of holey regions develops, as shown in figure 4.13.

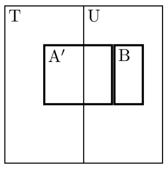
The strict containment requirement means that when a node is split, the split boundary, S divides the region, P, of no more than one primary entry. Every other primary region in the node is either contained by S and P, in which case it will go into the split node corresponding to the hole defined by S, or it contains S and P or is disjoint from S, in which case it will go into the split node corresponding to the remnant region after removal of the hole. (Since S divides P, a different primary entry cannot contain S and be contained by P). The only problem is the entry corresponding to P itself, part of which belongs in the split node corresponding to the hole, and part in the remnant node. This entry, then, is the only primary entry that must be elevated, guaranteeing an upper limit of one primary entry being elevated from a single node split.

4.6.2 Limiting indirect elevation: Demotion

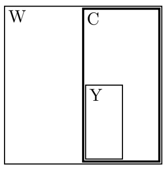
Consider a node, N, containing a number of primary entries, and one elevated entry, E. Suppose that E's child, a node of level l, overflows, causing a new entry to be posted into N (we consider the correctness of this in sections 4.6.6 and 5.1). This new entry is of level l, which must be less than the level of node N from the very fact that entry E is elevated — so a 'normal' node split and post has caused the number of elevated entries in N to increase. We refer to this as *indirect elevation*. Because indirect elevation is a consequence of normal tree growth and development, it cannot be limited in the sense of preventing overflow in children of elevated entries — to do so would compromise the tree's ability to organise itself dynamically. Instead, we must recognise that



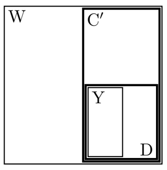
(a) KDB-VFS entry A virtually split across entries T and U



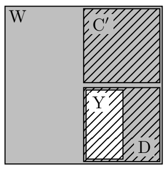
(b) KDB-VFS entry B is not virtually split



(c) BV entry C virtually split across entries W and Y



(d) BV-entry C′ is not virtually split



(e) Space actually represented by BV-regions C′ and D

Figure 4.14: Indirect elevation caused by split of already-elevated entries. Heavier boundary lines indicate elevated entries.

the split resulting from such an overflow may change the elevated entries' requirement for a virtual split, and take advantage of this where possible.

Figure 4.14a shows a KDB-VFS region, A, virtually split across regions T and U; in figure 4.14b, the child of A has overflowed, causing it to split into regions A′ and B — notice that although A′ remains virtually split across T and U, B is contained fully within region U. This is a consequence of the position of the split of region A; it is also possible that the split could occur directly *on* the T/U boundary, in which case neither of the resulting regions would be virtually split. Alternatively, had the split of A been in the horizontal plane, both resulting entries would be virtually split.

Similarly, figure 4.14c shows BV-region C virtually split across regions W and Y. Split of C into C′ and D is shown in figure 4.14d, and is such that D is virtually split, but C′ is not. The true extent of region C′ (as $C \setminus D$) is given in figure 4.14e; clearly C′ does not intersect region Y at all.

These examples illustrate that the actual split of virtually split entries may produce fragments that are not themselves virtually split; such fragments should be *demoted* to reduce the number of elevated entries present in the node. The need for demotion is acknowledged in the BV-tree, although its description is short of complete, and alterations would have to be made to the insertion algorithm as presented in [19] in order to accommodate the process.

4.6.3 Tree height and elevation

Figure 4.15 illustrates the phenomenon that causes tree height to be an issue. Figure 4.15a shows an overflowing VFS-node, M, containing five primary entries and one elevated entry; the associated spatial decomposition is given in figure 4.15b. A well-balanced, disjoint split of node M would be into partitions containing [W,X] and [Y,Z], bounded respectively by S and R, and shown in figure 4.15c, elevating entry V_1; note however that region A is contained in neither S nor R, and so is virtually split across the two, requiring entry A_0 to undergo further elevation. The effect of this is shown in figure 4.15d. In general, higher level nodes are likely to contain more elevated entries, simply because there are a greater number of entries in their subtrees, and by sheer weight of numbers more entries are likely to require a virtual split. This is directly analogous to the forced split situation in which split of a higher level node requires forced splitting of a taller subtree.

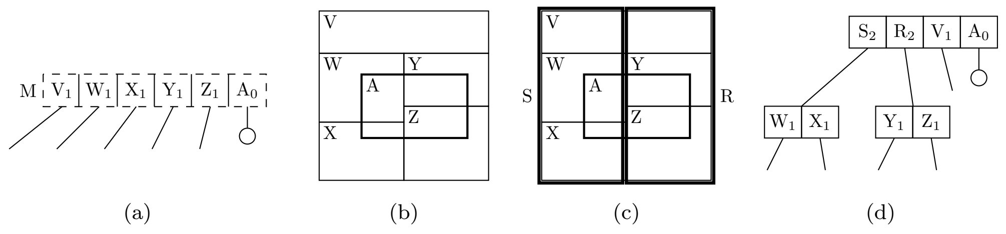


Figure 4.15: Split of node M requires further elevation of the already-elevated entry A_0.

If, as in the case of the KDB-VFS tree, we are unable to place limits on direct elevation, we are clearly also unable to do so on its cumulative effects in higher levels of the tree. This is, however, not the case in the BV-tree. In section 4.6.1, we described how the BV-tree mode of node splitting limits the number of primary entries requiring elevation at node split to one. In a splitting BV-node containing elevated entries, we still have at most one primary entry requiring elevation, but the reason for its elevation — that part of the entry belongs inside the hole formed by the split, and part of it outside — applies equally to entries elevated into the node previously. As in the case of primary entries, given a number of elevated entries of a single natural level, at most one of those entries will require elevation by virtue of the strict containment property. In general, however, split of a node of level L may require as many as $L + 1$ entries to be elevated into the level above: one of each natural level from 0 to L (the natural level of the primary entry to be elevated).

4.6.4 Overall elevation limits

The fact that the number of elevated entries in a node is dependent on these three factors requires us to take care in stating limits on the number of elevated entries in the node. In particular, the BV-tree presentation describes limits in terms of the maximum number of entries that can be elevated directly during node split, rather than in terms of the number of elevated entries present in a node. The key guarantee in the BV-tree case is not that direct elevation is limited but instead that, for every primary entry in the node, there can be only one elevated entry from each of the non-leaf levels beneath that node in the tree.

This ratio comes directly from the holey BV-tree spatial decomposition. An elevated entry, E, can be contained by only one primary entry; if two primary entry boundaries, P_1 and P_2, both contain E, then either P_1 contains P_2 or P_2 contains P_1. Assuming, without loss of generality, the latter, then P_1 forms a hole in P_2, meaning that E falls inside that hole and cannot intersect P_2. Any elevated entry, E, is therefore virtually split across the single primary region that contains it, and one or more holes, contained within the boundary of E, in that primary region.

Our statement of the limit on elevation in the BV-tree relies on the fact that, given a collection of primary entries across which an elevated entry is virtually split, only one of those primary entry boundaries can contain the elevated entry. Recognition of this also makes plain our reasons for avoiding the term ‘guard’ to describe an elevated entry. Figure 4.16a shows a BV-region W which in figure 4.16b has undergone a split into region W′ and X, with region C virtually split across the two. Freeston’s terminology refers to C as the guard of X (but not of W), suggestive of an asymmetric association between C and X (and not C and W), when in fact C is virtually split across both W and X. Furthermore, region W′ might undergo further splits, the results of which might also be contained in C, as in figure 4.16c. In this case, W′ has been split into W′′ and Y without any new virtual forced split of a region across W′′ and Y. In Freeston’s terminology Y

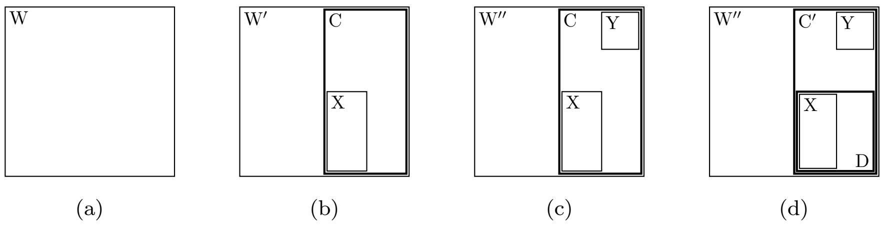


Figure 4.16: Evolution of a BV-node on split of primary and elevated entries. Heavier boundary lines indicate elevated entries.

does not have a guard, however, it is clear from the figure that C is now virtually split across W″, X and Y.

We believe that the one-to-one, asymmetric relationship between 'guard' and 'guarded', suggested by the terminology, and by the description of virtual splitting solely in terms of the process of direct elevation, has also led to misconceptions with demotion. As we described in section 4.6.2, demotion is required to limit elevation due to the split of already-elevated entries, a requirement for the BV-tree recognised by Freeston. However, in [19] it is asserted that such a split will always permit a demotion. This would be correct if an elevated entry were only ever split across two primary entries, but as figure 4.16c illustrates, this is not the case. A subsequent split of entry C into C′ and D is shown in figure 4.16d; in this case neither C′ nor D is demotable, but each is virtually split, across W″ and respectively Y and X.

4.6.5 Handling occupancy effects of elevation

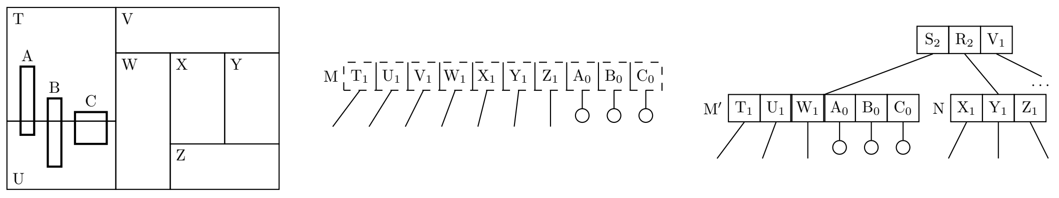
Our discussion about elevation and guaranteed limits allows us to make the following observations:

- A BV-tree node of level L containing P primary entries may contain as many as $L \times P$ elevated entries;
- A KDB-VFS tree node of level L containing P primary entries may contain many elevated entries, since it does not restrict the number of entries that need to be virtually split when partitioning a node's entries. This provides no clear guaranteed upper bound on the number of elevated entries in a node.

This has consequences for considerations of disk occupancy.

We consider first a BV-tree with a fixed node size that can accommodate 10 entries. A full level 0 node contains 10 level 0 (primary) entries, but a level 1 node may need to accommodate as many level 0 (elevated) entries as level 1 (primary) entries — five of each level. In level 2, the node's capacity for primary entries is further reduced, until in level 10 we cannot accommodate a single primary entry and be guaranteed to have space for all the necessary elevated entries, causing the structure to fail. Freeston recognises the effect of reducing a node's capacity for primary entries (although not its potential to cause failure — Samet notes this in [51]) and suggests using larger nodes in higher levels of the tree; if a level 0 node can accommodate F entries, a level n node in general may be required to accommodate $(n + 1)F$ entries. This guarantees that any index node, even if equipped with its full complement of elevated entries, will be able to accommodate F primary entries. We refer to these nodes as being of *level-multiplied* size.

Guaranteeing a node's capacity for primary entries permits VFS-trees in general to guarantee a worst-case split balance of 2:1 between primary entries; *primary* split balance is important in


T
V
A
B
C
W
X
Y
Z
U
M
T_1 U_1 V_1 W_1 X_1 Y_1 Z_1 A_0 B_0 C_0
S_2 R_2 V_1
M′
T_1 U_1 W_1 A_0 B_0 C_0
N
X_1 Y_1 Z_1

(a) Regions represented in node M. (b) Overflowing node M. (c) Split of M, balanced between primary entries only.

Figure 4.17: Distribution of elevated entries during a KDB-VFS node split.

order to guarantee primary fanout, effectively the minimum number of leaves reachable from a subtree. Figure 4.17a, shows a KDB-VFS spatial decomposition into seven regions T–Z with three virtually split regions A–C. Supposing these to occupy a single, overflowing node M (figure 4.17b), node M must undergo a split. One possible partitioning with 5:5 balance is [T_1,U_1,A_0,B_0,C_0] and [V_1,W_1,X_1,Y_1,Z_1], but selection of this partitioning has poor effects on overall fanout, since because entries A_0, B_0 and C_0 are of a lower level number than the node M, their subtrees are likely to contain significantly fewer leaf pages than primary entries T_1-Z_1. A better split for fanout overall in this case might be into [T_1,U_1,W_1,A_0,B_0,C_0] and [X_1,Y_1,Z_1], elevating entry V_1, as in figure 4.17c. Clearly, however, we need to guarantee a node's capacity for primary entries if we are to tolerate such an unbalanced distribution of elevated entries.

Notice, however, that raising the capacity of a node to $(n+1)F$ entries permits the possibility that the node might contain that many primary entries. This is clearly undesirable; we wish to maintain the logarithmic performance of the tree while also ensuring sufficient primary capacity. Note also that the capacity limit of $(n+1)F$ relies heavily on the BV-tree elevation limits — this solution is not appropriate for structures that might elevate less predictably, for example the KDB-VFS tree. Whether to support such structures in practice, or merely their development for evaluation, we require a more flexible approach.

Our solution to the problem is to represent a node as a 'bucket' consisting of a primary page and a chain of linked overflow pages as required. Primary entries may only be written onto the primary page, and the node is full when the primary page is full of primary entries. Elevated entries are written into unused space in the primary page and then into the overflow chain. In the case of unpredictable elevation, this allows elevated entries to be accommodated indefinitely, while in the BV-tree case, individual node sizes range between F and $(n+1)F$ entries, with between 2 and F primary entries.

We assume that, in both our linked-list approach and in the level-multiplied approach, the storage space required to accommodate F entries on disk is a whole number of disk pages. This being so, the linked list-approach guarantees, in the worst case — that of a BV-node containing $(n+1)F$ entries — no more than the same number of pages per node as the level-multiplied approach. We would anticipate better logarithmic performance when fewer entries are elevated, because additional capacity is not given over to primary entries, although, perversely, the linked-list approach may risk increasing the time required to read a node from disk if its pages are allocated non-contiguously.

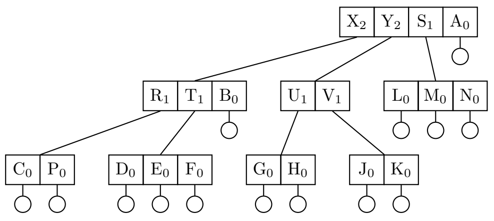


Figure 4.18: BV-tree example of a VFS-tree.

4.6.6 Algorithm design

We have at various points so far referred to virtual splitting as 'preserving the semantics of the forced split' which it replaces. We suggest that the key to understanding and designing algorithms on VFS-trees in general is to recognise that a VFS-tree is actually a compact representation of a simpler, balanced forced split tree. We call this a *reduced VFS-tree* (RVFS-tree). The RVFS-tree is derived from the VFS-tree and is never materialised on disk, but is generated lazily, in memory and on demand. During tree operations, a VFS-tree cannot be searched directly, but is used instead to construct fragments of the RVFS-tree required for the operation in question.

In section 5.1, we introduce a `reduce` operation that converts a full VFS-tree into the associated RVFS-tree. RVFS-trees contain no elevated entries and are fully balanced, although nodes may suffer from under-occupancy or from over-occupancy (*i.e.* more entries than can fit into a normal, single-page node). Reduction consists essentially of the execution of the hitherto avoided forced splits. Note, however, that an RVFS-tree is not a conventional FS-tree, because a full RVFS-node does not split when an elevated entry is reduced into it — it merely becomes 'overfull'. The RVFS-tree is *not* a practical access method implementation, but generalises the conversion required to search a VFS-tree fragment, and hence underlies every VFS-tree operation.

We have already remarked that the only published search algorithm for the BV-tree is exact-match. The key problem in implementing other algorithms is twofold. First, for non-updating algorithms such as searches with spatial extent, how can we choose which elevated entries should be carried down in the pending sets in every step, *i.e.* which entries will be required to construct the correct RVFS-tree? Second, updating algorithms such as insert and delete, and subordinate algorithms such as node-splitting and demotion of indirectly elevated entries, make updates to the lazily constructed, in-memory RVFS-fragment. How is the on-disk VFS-tree to be modified so that, under subsequent reduction, the modified RVFS-tree is correct?

We illustrate one facet of the update problem as follows. Figure 4.18 gives an example of a BV-tree, with the corresponding spatial decomposition in figure 4.19. Consider insertion of a point, lying between regions J and K, into this tree. Search for the insertion location proceeds initially into Y_2, carrying the pending set $\{S_1, A_0\}$, then descends V_1, carrying pending set $\{A_0\}$, before finally inserting the point into A_0. Supposing the child of A_0 to overflow, we must then post into its parent node. But which node is its parent?

The root node is certainly a candidate, for A_0 is physically located there, but we actually descended three levels before reaching the leaf, so the root appears to be its great-grandparent. This suggests that the correct parent should be the child of V_1, as this is the level 0 node through which the insertion passed before descending A_0. However, when one considers that not all the contents of the leaf beneath A_0 belong under V_1, nor even under Y_2, it seems questionable to post to V_1's child.

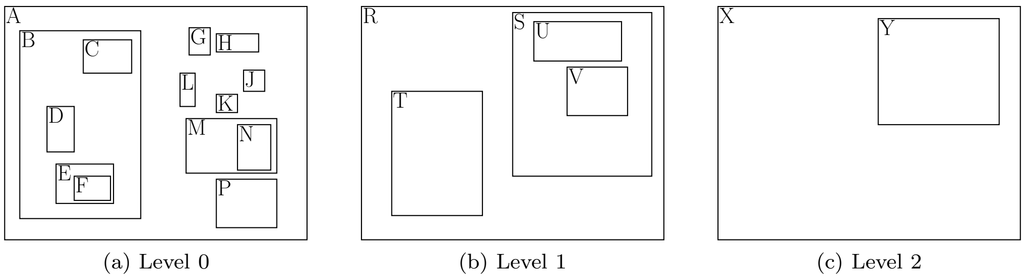


Figure 4.19: Region components of entries of different levels in the BV-tree of Figure 4.18.

$$\begin{tikzcd}
\text{VFS} \arrow[r, "vfsUpdate"] \arrow[d, "\texttt{reduce}"'] & \text{VFS}' \arrow[d, "\texttt{reduce}"] \\
\text{RVFS} \arrow[r, dashed, "rvfsUpdate?"] & \text{RVFS}'
\end{tikzcd}$$

Figure 4.20: A possible relationship between VFS- and RVFS-trees and respective operations. The diagram does not commute because the postulated *rvfsUpdate* operation does not exist.

To resolve this, and more convoluted cases, we use a guiding principle based on the observation that the VFS-tree is merely a representation of the RVFS-tree: any modification to the VFS-tree must preserve search correctness in the corresponding modified RVFS-tree. Note, however, that we cannot insist on a more direct relationship such as that suggested by figure 4.20 — we would like the diagram to commute, but it cannot, because no sensible *rvfsUpdate* exists to correspond to the *vfsUpdate* operation.

We introduce the `reduce` operation, to convert a VFS-tree into the corresponding RVFS-tree, in chapter 5, after which we will answer the question of what happens to the results of splitting A_0, and explain the non-existence of *rvfsUpdate*.

4.7 Summary

In this chapter we introduced VFS-trees; trees that use virtual forced splitting as a compact storage representation of a balanced forced split tree that we call an RVFS-tree. While a conceptually simple idea, correct handling of VFS-trees requires a clear understanding of the nature of virtual splitting and of the fact that search operations are performed exclusively on the associated RVFS-tree. In chapter 5 we describe the `reduce` operation to transform a VFS-tree into the associated RVFS-tree, and use this approach as a basis for the implementation of standard search and update operations in VFS-trees.

Chapter 5

VFS-tree operations

In chapter 4, we introduced virtual forced splitting as a way to represent a forced splitting structure efficiently, and outlined a number of issues with doing so. Recognition that a VFS-tree is a compact representation of an RVFS-tree is key to this approach. In section 5.1, we present a general-purpose transformation from a VFS-tree to the associated RVFS-tree, and in sections 5.2 and beyond, use the insights gained to develop the update and search algorithms required of multidimensional access methods as standard.

5.1 The `reduce` operation and the RVFS-tree

In this section, we introduce `reduce`, an algorithm to transform a VFS-tree into the associated RVFS-tree it represents. This underpins our approach to algorithm design for the VFS-tree on both a conceptual and practical level. By recognising that the VFS-tree is itself not directly searchable, but is an efficient on-disk representation of the searchable RVFS-tree, the purpose of pending sets becomes conceptually clearer. Practically, the lazy generation of the RVFS-tree on-the-fly means that when construction of an RVFS-branch is complete, so is our search — we have reached the RVFS-leaf and (in the case of a query) need simply collect the points we find there. In addition to being a general VFS/RVFS transformation algorithm, `reduce` therefore provides the structural basis for many VFS-tree algorithms.

The intuition behind the `reduce` operation is close to that of forced splitting, but rather than execute locally the split of an entry that may have been elevated through several levels, the entry is inserted into all primary subtrees whose regions it intersects. Figure 5.1 illustrates the RVFS-tree resulting from the reduction of the BV-tree in figure 4.18. Once-elevated entries in the figure are subscripted with the the labels of entries through which they have been reduced; natural levels are not given in RVFS-entries because the RVFS-tree contains no elevated entries. Notice that, although entry S now appears in the children of both entries X and Y, the children of the two S entries differ; as entry L falls completely outside region X it does not appear at all in the subtree rooted on entry X, likewise neither do entries M and N appear in Y's subtree. The contents of leaf nodes are similarly restricted; entry B appears in the subtrees of entries R and T, but the contents of the leaf nodes rooted on B_R and B_T consist of the subsets of points from the original leaf (rooted on B) that are contained in (interpreted) regions R and T respectively. Finally, although entry A_0 appears in the root of the BV-tree in figure 4.18, observe that it does not appear in every level 0 node; because region T is contained by region B, T falls into a hole in A and contains no space

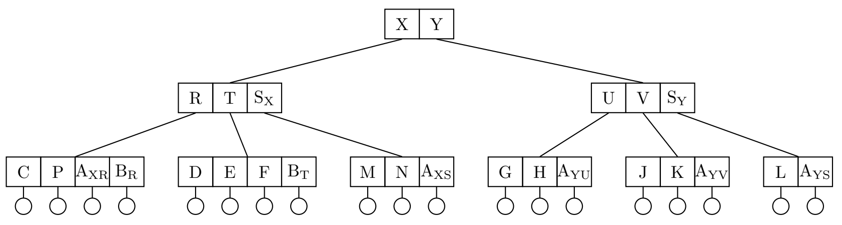


Figure 5.1: Reduced BV-tree example of an RVFS-tree.

also contained by A.

We can now revisit the example introduced in section 4.6.6, in which we considered the destination of entries posted as a result of splitting A_0. Using the reduction of figure 4.18 given in figure 5.1, we see immediately that our confusion was caused by a misinterpretation of A_0. What we actually inserted into was not A_0, but A_{YS}, the part of A_0 that belongs in that leaf position. However, what has to be split is not A_{YS}, but the whole A_0. This can only be split at the root (although one or both of the split parts might then be demotable down X_2 or Y_2, depending on the split boundary chosen), so entries resulting from the split of A_0 must be posted into the root.

This example also illustrates the reason for the non-existence of *rvfsUpdate* operations that would allow the diagram of figure 4.20 to commute. Because insertion into the VFS-tree causes a split of the whole of A_0, any of the RVFS-regions of which it is constituted might also be affected by the split. For an RVFS-tree insertion (*rvfsUpdate*) operation to allow figure 4.20 to commute, the operation would have to:

- recognise if the corresponding VFS-tree insertion would have caused the corresponding VFS-leaf to overflow. This would be required despite the fact that the RVFS-leaf into which insertion proceeds is unlikely to contain as many entries as the overflowing leaf (and that in the RVFS-tree the concept of node overflow does not exist).

- reproduce the effects of the split of A_0 in other affected RVFS-branches.

This would be difficult or impossible to achieve independently of the VFS-tree route, forcing figure 4.20 to commute rather artificially — under these circumstances the construction of an *rvfsUpdate* operation seems rather meaningless.

We now describe the detail of the `reduce` operation. We use node types $\mathtt{I}_R$ and $\mathtt{L}_R$ to denote RVFS-nodes, and to assist unambiguous specification of `reduce` will use $\mathtt{I}_V$ and $\mathtt{L}_V$ to denote VFS-nodes. In later, VFS-specific discussions, we will, however, revert to the usual `I` and `L`. As in the case of `fsInsert`, the RVFS-tree is written top-down, writing a VFS-node's primary entries (either local or from an incoming pending set) into an RVFS-node, and at the same time allocating child pages. Those pages are not written directly, but instead a *reduction tuple* is cached for each child, consisting of four components required to write the RVFS-node later:

- r, the address of the VFS-node to be reduced;

- r', the address into which the corresponding RVFS-node is to be written (this address will already have been written into the RVFS-node's parent);

- $\mathbf{G}$, the pending set of elevated entries that must be inserted into the RVFS-node and its subtrees;

- $\mathscr{Q}$, the global predicate for the RVFS-node.

The operation begins by pushing a single reduction tuple, for the root page of the VFS-tree, onto an empty stack. A tuple on the stack is consumed by converting it into either an RVFS-leaf, in the case of a tuple containing the address of a VFS-leaf, or into an internal RVFS-node if the tuple addresses an internal VFS-node. In the second case, the contents of the VFS-node are converted into yet more reduction tuples which are then also pushed onto the stack. The operation terminates when the stack is empty. A single reduction tuple $\langle r, r', \mathbf{G}, \mathscr{Q} \rangle$ is handled as follows:

1. Read page r.

2. If the VFS-node at address r is a leaf node, copy its contents into the empty RVFS-leaf at r', filtered using the global predicate in the reduction tuple, $\mathscr{Q}$. Filtering means that only those points that satisfy $\mathscr{Q}$ are copied into the RVFS-leaf; points that do not satisfy $\mathscr{Q}$ belong elsewhere in the RVFS-tree and are not copied into the RVFS-leaf.

3. Otherwise, the VFS-node at address r is an internal node, $\mathtt{I}_V\,(L, \mathbf{V})$. Our first task is to reinterpret the predicates in the node and those in the incoming pending set, with respect to one another, and to merge the results. Let:

$$\mathbf{E} = \mathtt{reinterpret}\,(\mathbf{V}, \mathbf{G}) \oplus \mathtt{reinterpret}\,(\mathbf{G}, \mathbf{V})$$

   We can now decide, for each primary entry $\langle L, p, s \rangle \in \mathbf{E}$, whether it belongs in this RVFS-node or elsewhere in the RVFS-tree: only if $p \not\parallel \mathscr{Q}$ does the entry belong in this RVFS-node. For each one of the primary entries whose interpreted predicate intersects $\mathscr{Q}$:

   (a) Form an RVFS-entry consisting of the entry's predicate, p, and a new page ID, s', to become the address of the child of the RVFS-entry.

   (b) Form a new reduction tuple consisting of the address of the VFS-entry's child, s, the address newly-allocated for the corresponding RVFS-entry's child, s', and a new pending set, $\mathbf{G}'$. $\mathbf{G}'$ is specific to this entry and contains all elevated entries from $\mathbf{E}$ intersecting the global predicate of the new RVFS-node, $\mathscr{Q} \wedge p$. Include this predicate as the final component of the new tuple.

4. Write out collected RVFS-entries into a new RVFS-node and push all newly-formed reduction tuples onto the stack.

In the same way as, in section 4.5, we described entries becoming available for descent only when they become primary, note that reduction tuples are only formed for elevated entries when those entries have themselves become primary (in step 3). Note also that the effect of step 3b is to add an elevated entry to the pending set for every primary entry that it intersects; when one of these entries becomes primary the approach is not then to split it forcibly between a number of RVFS branches, but rather to exclude from its subtree any elements that do not belong in the current RVFS branch. This is described at the beginning of step 3 by including only primary entries 'intersecting $\mathscr{Q}$'. Execution of the reduction of a VFS-entry that may have been virtually split across many subtrees is better described, not as the forced splitting of the entry between RVFS branches, but as the confinement of that entry's subtree to each RVFS branch in turn.

The formal structure of an RVFS-node is as follows:

$$\begin{aligned} \textit{Node} &::= \textit{INode} \mid \textit{LNode} \\ \textit{INode} &::= \mathtt{I}_R\,([\textit{LocalPredicate}, \textit{Node}]) \\ \textit{LNode} &::= \mathtt{L}_R\,([\langle \textit{Key}, \textit{Value}\rangle]) \end{aligned}$$

Note that the *INode* structure no longer requires entry or node level numbers, since elevated entries no longer exist. The `reduce` ruleset uses a single, zero-argument command, `red`, and has the following configuration structure:

$$\big\langle \mathtt{red}\,,\,\pi\,,\,f\,,\,\sigma \big\rangle$$

where π is a stack of reduction tuples, f is the address of the new RVFS-root and σ is the store.

We introduce two ancillary functions used to aid concision in the `reduce` ruleset (and other subsequent rulesets). `filter` takes a predicate and a list of interpreted node entries, returning only those entries in the list whose predicates intersect the predicate argument to `filter`:

$$\mathtt{filter}\,(\mathscr{Q}, \mathbf{E}) = [\ \langle l, p, r\rangle\ \mid \langle l, p, r\rangle \in \mathbf{E} \wedge p \between \mathscr{Q}]$$

$\mathtt{lev}_{<L}$ takes a list of entries and returns only those of level number strictly less than L:

$$\mathtt{lev}_{<L}\,(\mathbf{E}) = [\ \langle l, p, r\rangle\ \mid \langle l, p, r\rangle \in \mathbf{E} \wedge l < L]$$

In a node of level L, $\mathtt{lev}_{<L}$ can be used to separate elevated entries from the mixed list of primary and elevated entries in the node.

The `reduce` algorithm is given in figure 5.2. Transition 4.1 bootstraps the operation from the external `reduce` command (which takes as a parameter the page identifier of the VFS-root) into a `red` command with the root reduction tuple on the stack. Transition 4.2 reduces a leaf node; this does not require insertion of pending entries as none exist, but may require the confinement of the leaf's contents to this subtree if the leaf was once elevated. Transition 4.3 contains the heart of the operation, described above as the treatment of a single reduction tuple, while transition 4.4 rewrites from a `red` command with no remaining reduction tuples into an external `rvfs` command containing the root of the RVFS-tree.

Unlike the FS-tree algorithm, `reduce` makes no change to VFS-entry local predicates; it merely ensures that no data points and no node predicates in an RVFS subtree lie outside that subtree's global predicate. This is sufficient for search correctness, since if we reduce predicate P_N into a node with global predicate $\mathscr{P}$, the reduced node's new global predicate, $\mathscr{P}_N = \mathscr{P} \wedge P_N$ is confined to $\mathscr{P}$. Notice, also, that in transition 4.3:

1. the pending set for each primary entry $\langle L, p, s\rangle$ is filtered using $\mathscr{Q} \wedge p$;

2. *all* primary entries, both local and from the incoming pending set, are filtered using the node's global predicate $\mathscr{Q}$.

This means that elevated entries that become primary in a node have already been filtered with that node's global predicate — why is it then necessary to do so again as in step 2? The reason is the fact that the predicates interpreted previously for elements selected into the pending set,

$$\big\langle \texttt{reduce}\,(v)\,,\,\sigma \big\rangle \quad\rightsquigarrow\quad \big\langle \texttt{red}\,,\,[\langle v,f,[\,],\texttt{True}\rangle]\,,\,f\,,\,\sigma \big\rangle \tag{4.1}$$

$$\text{where } f = \texttt{fresh}(\sigma)$$

$$\big\langle \texttt{red}\,,\,\langle r,r',\_,\mathscr{Q}\rangle :: \pi\,,\,f\,,\,\sigma \big\rangle \quad\rightsquigarrow\quad \big\langle \texttt{red}\,,\,\pi\,,\,f\,,\,\sigma[r' \mapsto \texttt{L}_R\,(\mathbf{E}')] \big\rangle \tag{4.2}$$

$$\text{if } \sigma(r) = \texttt{L}_V\,(\mathbf{E})$$

$$\text{where } \mathbf{E}' = [\ \langle k,v\rangle \ \mid \langle k,v\rangle \in \mathbf{E} \wedge \mathscr{Q}(k)\,]$$

$$\big\langle \texttt{red}\,,\,\langle r,r',\mathbf{G},\mathscr{Q}\rangle :: \pi\,,\,f\,,\,\sigma \big\rangle \quad\rightsquigarrow\quad \big\langle \texttt{red}\,,\,\mathbf{T} \oplus \pi\,,\,f\,,\,\sigma[r' \mapsto \texttt{I}_R\,(\mathbf{F})] \big\rangle \tag{4.3}$$

$$\text{where } \sigma(r) = \texttt{I}_V\,(L,\mathbf{V})$$

$$\text{and } \mathbf{E} = \texttt{reinterpret}\,(\mathbf{V},\mathbf{G}) \oplus \texttt{reinterpret}\,(\mathbf{G},\mathbf{V})$$

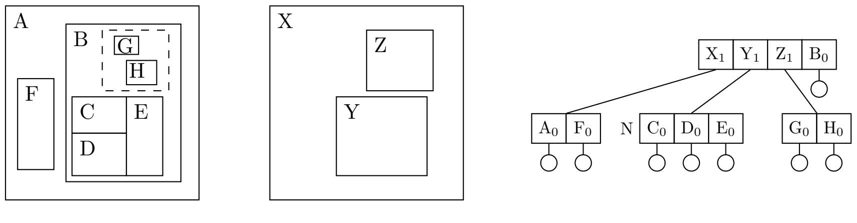
$$\text{and } \mathbf{P} = [\ \langle p,s,\texttt{fresh}(\sigma),\mathbf{G}'\rangle \ \mid \langle L,p,s\rangle \in \texttt{filter}\,(\mathscr{Q},\mathbf{E}) \wedge \mathbf{G}' = \texttt{filter}\,(\mathscr{Q} \wedge p, \texttt{lev}_{<L}\,(\mathbf{E}))\,]$$

$$\text{and } \mathbf{F} = [\ \langle p,s'\rangle \ \mid \langle p,\_,s',\_\rangle \in \mathbf{P}\,]$$

$$\text{and } \mathbf{T} = [\ \langle s,s',\mathbf{G}',\mathscr{Q} \wedge p\rangle \ \mid \langle p,s,s',\mathbf{G}'\rangle \in \mathbf{P}\,]$$

$$\big\langle \texttt{red}\,,\,[\,]\,,\,f\,,\,\sigma \big\rangle \quad\rightsquigarrow\quad \big\langle \texttt{rvfs}\,(f)\,,\,\sigma \big\rangle \tag{4.4}$$

Figure 5.2: The VFS-tree `reduce` algorithm.



(a) Level 0 (b) Level 1 (c) BV-tree

Figure 5.3: On examination of the BV-tree root, entry B_0 is included in the pending set for reduction of Y_1, because its interpreted predicate is $\lambda x.x \in \mathrm{B}$. It is only on descent into node N that interpretation of $\lambda x.x \in (\mathrm{B} \setminus (\mathrm{C} \cup \mathrm{D} \cup \mathrm{E}))$ permits B_0 to be pruned.

and those that can now be interpreted for incoming primary entries in the pending set, may differ. Figure 5.3a shows a collection of level 0 regions, and figure 5.3b their partitioning at level 1. The dashed line in figure 5.3a indicates the position of the boundary between regions X and Z; the boundary between X and Y is coincident with the outer boundary of the block of regions formed by C, D and E. The resulting BV-tree is shown in figure 5.3c. Consider now the action of `reduce` on the subtree of entry Y_1. In the root, entry B_0 is selected for the pending set, as its outer boundary contains Y, however, when inside node N, it is found that, together, C, D and E span region Y — Y actually coincides with a hole in B_0, so B_0 should be pruned from the search. In general, when an elevated entry is selected for a pending set, it does not mean that the entry will not ultimately be pruned — merely that it cannot be pruned at this time.

We illustrate the action of `reduce` in detail with a further example. Figure 5.4a shows a VFS-tree fragment with relevant associated region predicates shown in figure 5.4b, and figure 5.4c the RVFS-tree fragment resulting from the reduction of this VFS-fragment. We assume these predicates to be simple rectangles, such that they are already fully interpreted, having the effect that $\texttt{reinterpret}\,(\mathbf{E}_1,\mathbf{E}_2) = \mathbf{E}_1$. We unroll some of the detail of reducing this fragment, illustrating, at various points, the state of the RVFS-tree and of the stack of reduction tuples. In the tree diagrams of figure 5.4c, dashed node pointers indicate that the child page has been allocated by `reduce` but not yet written; solid pointers not terminated with an explicit child node indicate merely that the detail of the child has been omitted. In the figure, we use the following convention to specify page identifiers in reduction tuples: if a VFS-node entry has predicate P, the primary page of its child is located at page address p (lowercase italicised) and the associated RVFS-node will be written out into a bucket whose primary page is at address p' (lowercase italicised primed). The VFS-entry $\langle l, \mathrm{P}, p\rangle$ is abbreviated to P_l as in the VFS-tree diagram.

- The operation is bootstrapped using transition 4.1, allocating page v' to hold the root of the new RVFS-tree:
$$\left\langle \mathtt{reduce}\,(v)\,,\,\sigma\right\rangle \rightsquigarrow \left\langle \mathtt{red}\,,\,[\langle v, v', [\,], \mathtt{True}\rangle]\,,\,v'\,,\,\sigma\right\rangle$$
Rewrite of this `red` configuration proceeds using transition 4.3, writing the root node of the RVFS-tree and pushing reduction tuples for its entries onto the stack. The output configuration is:
$$\left\langle \mathtt{red}\,,\,[\langle x, x', [\mathrm{R}_1], \mathrm{X}\rangle, \langle y, y', [\mathrm{S}_1], \mathrm{Y}\rangle, \langle z, z', [\mathrm{R}_1, \mathrm{S}_1], \mathrm{Z}\rangle]\,,\,v'\,,\,\sigma'\right\rangle$$
and is shown in figure 5.5a, where $\sigma' = \sigma[v' \mapsto \mathtt{I}_R\,([\langle \mathrm{X}, x'\rangle, \langle \mathrm{Y}, y'\rangle, \langle \mathrm{Z}, z'\rangle])]$.

- Reduction tuple $\langle x, x', [\mathrm{R}_1], \mathrm{X}\rangle$ is popped off the stack and the VFS-node at address x read, yielding $\mathtt{I}_V\,(1, [\langle 1, \mathrm{J}, j\rangle, \langle 1, \mathrm{K}, k\rangle])$. Pending entry $\mathrm{R}_1 = \langle 1, \mathrm{R}, r\rangle$ becomes primary at this level, so after RVFS-node page allocation, the node $\mathtt{I}_R\,([\langle \mathrm{J}, j'\rangle, \langle \mathrm{K}, k'\rangle, \langle \mathrm{R}, r'\rangle])$ is written to page x' and associated reduction tuples pushed onto the stack (see figure 5.5b). Note that the global predicate is formed by taking the conjunction of each local predicate with X; although $\mathrm{J} \subseteq \mathrm{X}$ and $\mathrm{K} \subseteq \mathrm{X}$, conversely $\mathrm{R} \nsubseteq \mathrm{X}$, so the global predicate will be required later to exclude elements from the RVFS-subtree to be rooted on r'.

- Because no further pending elements were present in the previous step, the three reduction tuples pushed onto the stack have empty pending sets. Reduction of the VFS-subtrees rooted at addresses j and k therefore amounts to a copy of the contents of their subtrees into new RVFS nodes. We omit the detail here; figure 5.5c shows the tree and stack immediately after this.

- In figure 5.5d, following pop of $\langle r, r', [\,], \mathrm{X} \wedge \mathrm{R}\rangle$ from the stack, the node rooted on r has been read, yielding $\mathtt{I}_V\,(0, [\langle 0, \mathrm{F}, f\rangle, \langle 0, \mathrm{G}, g\rangle, \langle 0, \mathrm{H}, h\rangle])$. Note, however, that since $\mathrm{H} \nparallel (\mathrm{X} \wedge \mathrm{R})$, the RVFS-node written to page r' is $\mathtt{I}_R\,([\langle \mathrm{F}, f'\rangle, \langle \mathrm{G}, g'\rangle])$, and only two reduction tuples are placed on the stack.

- Similarly, in figure 5.5e, $\langle f, f', [\,], \mathrm{F} \wedge \mathrm{X} \wedge \mathrm{R}\rangle$ has been popped from the stack and the node at f read to yield $\mathtt{L}_V\,([p, q])$. Of the two points in the page, only p satisfies $\mathrm{F} \wedge \mathrm{X} \wedge \mathrm{R}$, so only p is written to the corresponding RVFS-node at f', using transition 4.2. Execution will continue with reduction of the node at address g.

5.2 Query algorithms

5.2.1 Queries of fixed extent: `rQuery`

We use the term ‘fixed extent’ to refer to queries that are posed in terms of a fixed query region, described either explicitly, for example as a query window or point, or implicitly as a range query around a given query point. A K Nearest Neighbour query can be described as a range query with an initially infinite radius which contracts during the search; its final radius is not known until

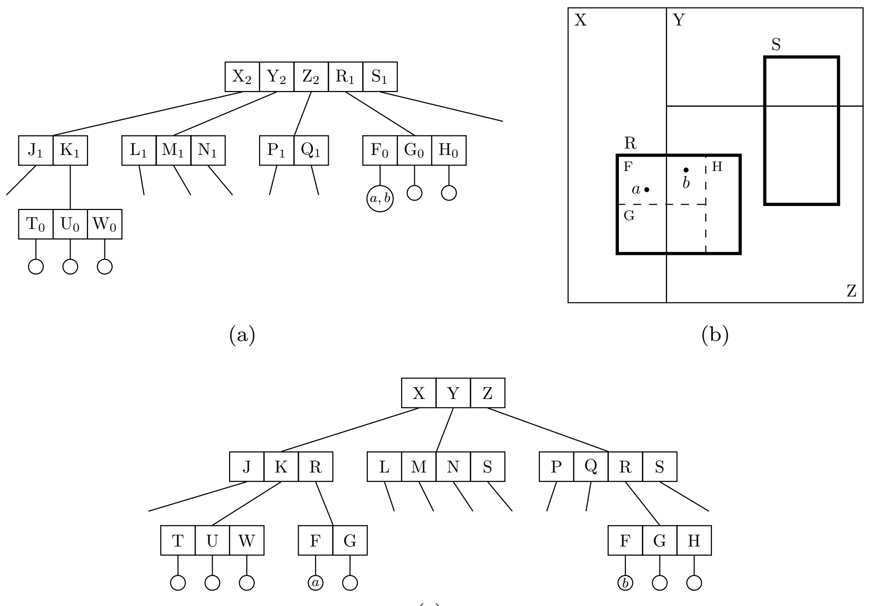


Figure 5.4: A VFS-tree fragment and the RVFS-fragment resulting from its reduction.

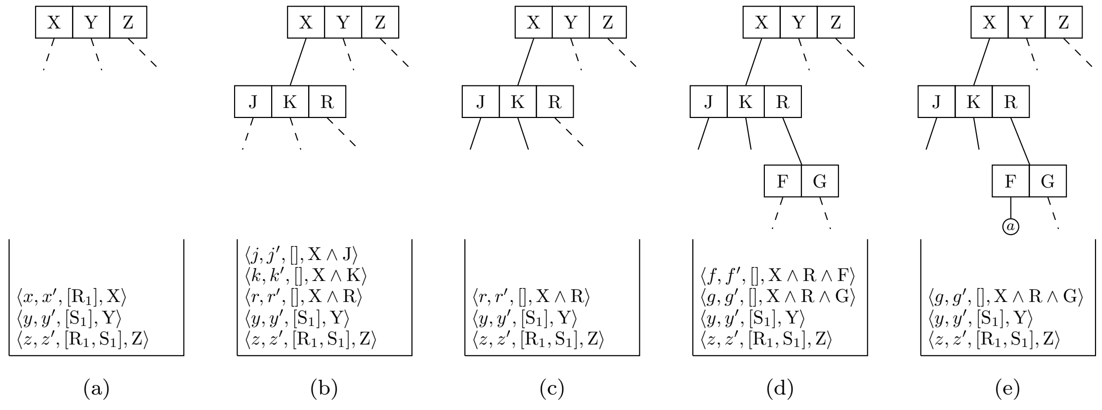

$\langle x, x', [R_1], X\rangle$
$\langle y, y', [S_1], Y\rangle$
$\langle z, z', [R_1, S_1], Z\rangle$
$\langle j, j', [], X \wedge J\rangle$
$\langle k, k', [], X \wedge K\rangle$
$\langle r, r', [], X \wedge R\rangle$
$\langle f, f', [], X \wedge R \wedge F\rangle$
$\langle g, g', [], X \wedge R \wedge G\rangle$
(a)
(b)
(c)
(d)
(e)

Figure 5.5: Development of the RVFS-tree and the stack of reduction tuples during reduction of the VFS-tree of figure 5.4a.

query execution is complete; we postpone discussion of queries such as these, with variable extent, until section 5.2.2. The discussion here includes queries of arbitrary but fixed region shape and size. As we consider a class of point access methods, the result set of a fixed-extent query consists of every point, stored in a tree's leaves, and satisfying query predicate $\lambda x.x \in R$, where R describes the fixed extent region. Exact-match queries are a special case of this type of query, in which the query region is a point.

A fixed-extent region query on an RVFS-tree must descend all branches of the tree whose regions intersect the query region, pruning, as we proceed, any subtrees that do not intersect the query region. In the VFS-tree case we do the same, but must decide which elevated entries to carry in the pending sets as we descend. In fact, any elevated entry that does not intersect the query region is one that would eventually be merged, by `reduce`, into a branch of the RVFS-tree that will be pruned from the search. Hence we need to carry down all and only the elevated entries that intersect the query region.

Note, however, that because virtual splitting is used to preserve the forced split semantics, an elevated subtree can only be searched correctly in the context of the primary subtree into which it has been carried, excluding entries that belong outside that primary subtree. If the search requires those excluded entries, they will be examined when the search proceeds down the primary branch in which they really belong. This is semantically correct in that we are reconstituting the effect of a forced split in the RVFS-tree, subdividing a VFS-node into two or more RVFS-nodes; a physical VFS-node representing a number of RVFS-nodes must be read once for each RVFS-node accessed. For example, consider a search of the VFS-tree in figure 5.4a for points contained in region F of figure 5.4b. The RVFS-tree in figure 5.4c illustrates that point a belongs in the subtree rooted on X, while b belongs in that rooted on Z. A search of the RVFS-tree therefore requires us to visit node F twice; once for each subtree across which F is virtually split. This is similar to the approach taken by `reduce` to confine a once-elevated subtree's contents to the subtree into which it has been reduced, and we take the same approach here, by confining the query predicate to the subtree into which the search proceeds to prevent our 'finding' of results that lie outside the RVFS-node but inside the VFS-node.

Practically, this makes `rQuery` structurally similar to `reduce`. Reduction is controlled using the stack of reduction tuples, each of which contains the page ID of the node to be reduced, a pending set of entries to be carried into that node, and the node's global predicate (confined to the subtree in which the node is found). `rQuery` is controlled in the same way, the only difference in a query tuple being that a second page ID (the address of the associated RVFS-node in `reduce`) is not required. In the `rQuery` configuration, we substitute reduction tuples containing RVFS-node addresses for query tuples which do not, and we replace the RVFS-root address (the output of `reduce`) with a result set. Configurations for the internal states of the `rQuery` algorithm have the form:

$$\left\langle \mathtt{qry}\,,\, \pi\,,\, \mathbf{R}\,,\, \sigma \right\rangle$$

where `qry` is the query command, π is a stack of query tuples, $\mathbf{R}$ is the query result set and σ is the store containing the VFS-tree. A query tuple has the form $\langle r, \mathbf{G}, \mathscr{Q}' \rangle$, where r is a VFS-node address and $\mathbf{G}$ is a pending set of elevated entries intersecting $\mathscr{Q}'$, the conjunction of the node's global predicate with the query predicate.

Figure 5.6 provides a set of rules for fixed extent queries, based on a suitably modified `reduce` rule set. Search is bootstrapped in transition 5.1, in which an `rQuery` command containing the

$$\big\langle \texttt{rQuery}\,(v,\mathscr{Q})\,,\,\sigma\big\rangle \;\rightsquigarrow\; \big\langle \texttt{qry}\,,\,[\langle v,[\,],\mathscr{Q}\rangle]\,,\,[\,]\,,\,\sigma\big\rangle \tag{5.1}$$

$$\big\langle \texttt{qry}\,,\,\langle r,\_,\mathscr{Q}\rangle :: \pi\,,\,\mathbf{R}\,,\,\sigma\big\rangle \;\rightsquigarrow\; \big\langle \texttt{qry}\,,\,\pi\,,\,\mathbf{R}\oplus\mathbf{E}'\,,\,\sigma\big\rangle \tag{5.2}$$
$$\text{if } \sigma(r) = \texttt{L}\,(\mathbf{E})$$
$$\text{where } \mathbf{E}' = [\ \langle k,v\rangle \mid \langle k,v\rangle \in \mathbf{E} \wedge \mathscr{Q}(k)\,]$$

$$\big\langle \texttt{qry}\,,\,\langle r,\mathbf{G},\mathscr{Q}\rangle :: \pi\,,\,\mathbf{R}\,,\,\sigma\big\rangle \;\rightsquigarrow\; \big\langle \texttt{qry}\,,\,\mathbf{T}\oplus\pi\,,\,\mathbf{R}\,,\,\sigma\big\rangle \tag{5.3}$$
$$\text{where } \sigma(r) = \texttt{I}\,(L,\mathbf{V})$$
$$\text{and } \mathbf{E} = \texttt{reinterpret}\,(\mathbf{V},\mathbf{G}) \oplus \texttt{reinterpret}\,(\mathbf{G},\mathbf{V})$$
$$\text{and } \mathbf{T} = [\ \langle s,\mathbf{G}',\mathscr{Q}\wedge p\rangle \mid \langle L,p,s\rangle \in (\texttt{filter}\,(\mathscr{Q},\mathbf{E})) \wedge \mathbf{G}' = \texttt{filter}\,(\mathscr{Q}\wedge p, \texttt{lev}_{<L}\,(\mathbf{E}))\,]$$

$$\big\langle \texttt{qry}\,,\,[\,]\,,\,\mathbf{R}\,,\,\sigma\big\rangle \;\rightsquigarrow\; \big\langle \texttt{results}\,(\mathbf{R})\,,\,\sigma\big\rangle \tag{5.4}$$

Figure 5.6: The VFS-tree `rQuery` algorithm.

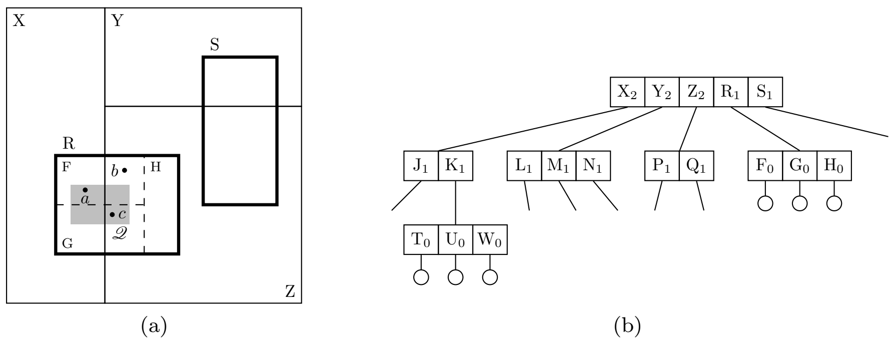


Figure 5.7: Spatial decomposition and VFS-tree for the query examples of figures 5.8 and 5.9.

VFS-root address and a query predicate is written into the first `qry` configuration. Transition 5.2 adds points from a leaf node, found to satisfy the query $\mathscr{Q}$, to the result set $\mathbf{R}$, while transition 5.3 assembles primary entries requiring exploration with the relevant pending sets and pushes them onto the stack as query tuples. When the stack contains no further query tuples, query execution is complete and the search terminates in an external `results` configuration via transition 5.4.

Figure 5.7 gives the same spatial decomposition and VFS-tree as that used in our reduction example, showing points a, b and c, and a query region, $\mathscr{Q}$. As we described at the beginning of this section, region query search collapses to exact-match search when the query region is a point; figure 5.8 illustrates the sequence of configurations encountered in executing an exact-match search of this tree for point a. We refer to this as a *trace*.

A second trace, given in figure 5.9, illustrates the execution of search of the same tree for query region $\mathscr{Q}$. This illustrates the possibility of multiple visits to a single VFS-node — but multiple RVFS-nodes — as described above. Pages r, f and g are each visited twice, once for each portion of the associated region that lies on either side of the X/Z boundary. This is specified by the inclusion in the query tuple's global predicate of either X and Z as required. For example, the tuple at the top of the stack in line 6 describes accessing the VFS-node at page g to retrieve points in the RVFS-node lying within G and X, while that at the top of the stack in line 10 describes accessing the same page to retrieve points in the RVFS-node lying within G and Z. It is for this reason that the first visit to page g adds no points to the result set; it is not until line 11 that point c is added.

Line	
1	$\left\langle \texttt{rQuery}\,(v,a)\,,\,\sigma \right\rangle$
2	$\left\langle \texttt{qry}\,,\,[\langle v,[\,],a\rangle]\,,\,[\,]\,,\,\sigma \right\rangle$
3	$\left\langle \texttt{qry}\,,\,[\langle x,[\mathrm{R}_1],a\rangle]\,,\,[\,]\,,\,\sigma \right\rangle$
4	$\left\langle \texttt{qry}\,,\,[\langle r,[\,],a\rangle]\,,\,[\,]\,,\,\sigma \right\rangle$
5	$\left\langle \texttt{qry}\,,\,[\langle f,[\,],a\rangle]\,,\,[\,]\,,\,\sigma \right\rangle$
6	$\left\langle \texttt{qry}\,,\,[\,]\,,\,[a]\,,\,\sigma \right\rangle$
7	$\left\langle \texttt{results}\,([a])\,,\,\sigma \right\rangle$

Figure 5.8: Trace of an exact-match search for point a in the VFS-tree of figure 5.7b.

Line	
1	$\left\langle \texttt{rQuery}\,(v,\mathscr{Q})\,,\,\sigma \right\rangle$
2	$\left\langle \texttt{qry}\,,\,[\langle v,[\,],\mathscr{Q}\rangle]\,,\,[\,]\,,\,\sigma \right\rangle$
3	$\left\langle \texttt{qry}\,,\,[\langle x,[\mathrm{R}_1],\mathscr{Q}\wedge \mathrm{X}\rangle,\langle z,[\mathrm{R}_1],\mathscr{Q}\wedge \mathrm{Z}\rangle]\,,\,[\,]\,,\,\sigma \right\rangle$
4	$\left\langle \texttt{qry}\,,\,[\langle r,[\,],\mathscr{Q}\wedge \mathrm{X}\wedge \mathrm{R}\rangle,\langle z,[\mathrm{R}_1],\mathscr{Q}\wedge \mathrm{Z}\rangle]\,,\,[\,]\,,\,\sigma \right\rangle$
5	$\left\langle \texttt{qry}\,,\,[\langle f,[\,],\mathscr{Q}\wedge \mathrm{X}\wedge \mathrm{R}\wedge \mathrm{F}\rangle,\langle g,[\,],\mathscr{Q}\wedge \mathrm{X}\wedge \mathrm{R}\wedge \mathrm{G}\rangle,\langle z,[\mathrm{R}_1],\mathscr{Q}\wedge \mathrm{Z}\rangle]\,,\,[\,]\,,\,\sigma \right\rangle$
6	$\left\langle \texttt{qry}\,,\,[\langle g,[\,],\mathscr{Q}\wedge \mathrm{X}\wedge \mathrm{R}\wedge \mathrm{G}\rangle,\langle z,[\mathrm{R}_1],\mathscr{Q}\wedge \mathrm{Z}\rangle]\,,\,[a]\,,\,\sigma \right\rangle$
7	$\left\langle \texttt{qry}\,,\,[\langle z,[\mathrm{R}_1],\mathscr{Q}\wedge \mathrm{Z}\rangle]\,,\,[a]\,,\,\sigma \right\rangle$
8	$\left\langle \texttt{qry}\,,\,[\langle r,[\,],\mathscr{Q}\wedge \mathrm{Z}\wedge \mathrm{R}\rangle]\,,\,[a]\,,\,\sigma \right\rangle$
9	$\left\langle \texttt{qry}\,,\,[\langle f,[\,],\mathscr{Q}\wedge \mathrm{Z}\wedge \mathrm{R}\wedge \mathrm{F}\rangle,\langle g,[\,],\mathscr{Q}\wedge \mathrm{Z}\wedge \mathrm{R}\wedge \mathrm{G}\rangle]\,,\,[a]\,,\,\sigma \right\rangle$
10	$\left\langle \texttt{qry}\,,\,[\langle g,[\,],\mathscr{Q}\wedge \mathrm{Z}\wedge \mathrm{R}\wedge \mathrm{G}\rangle]\,,\,[a]\,,\,\sigma \right\rangle$
11	$\left\langle \texttt{qry}\,,\,[\,]\,,\,[a,c]\,,\,\sigma \right\rangle$
12	$\left\langle \texttt{results}\,([a,c])\,,\,\sigma \right\rangle$

Figure 5.9: Trace of a search of region $\mathscr{Q}$ in the VFS-tree of figure 5.7b.

It could be argued that, for a real implementation, one would wish to optimise this situation by 'reading-ahead' all entries from a VFS-node when accessing the first RVFS-node included therein. We do not consider this further here, being more concerned with the explanation of the underlying approach. Note, however, that the store component of a query configuration is, in a real DBMS, implemented as a buffer pool; when answering queries over small regions, even without such an optimisation, subsequent reads of an already-visited physical page are likely to find that page still present in the pool.

Finally, observe that in figure 5.8 we have not specified the explicit construction of the global predicate as search descends the tree, but that in all cases that predicate evaluates to a. Because the query predicate, a, is a point, conjunction with other predicates cannot confine it any more closely; for any predicate P for which $P(a)$, $P \wedge a = a$. This description of exact match search causes our specification of pending sets to collapse precisely to that described by Freeston for the BV-tree.

5.2.2 K Nearest Neighbour Queries

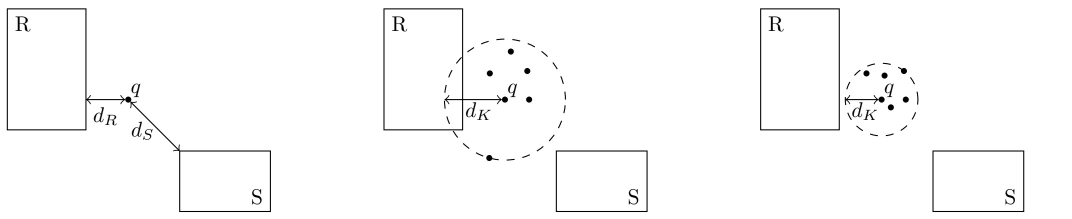
K Nearest Neighbour (K-NN) queries return, given a query object drawn from the data space, the K objects in the index structure closest to the query object. We refer to such queries as not being of fixed extent because the query region or radius will vary from query to query for a given value of K. Furthermore, as described in section 5.2.1, the radius of any given query remains unknown until query execution is complete; one can picture a K-NN query as beginning with an infinite radius that gradually contracts as results are found, until those results include the K nearest neighbours and the query radius is established. The notion of 'nearest' relies implicitly on a distance function between two points, which we denote with `dist`, returning a real number greater than or equal to zero.

There are two common approaches to K-NN queries in hierarchical structures: *depth-first* and *best-first* [29] search. In both cases, pruning of subtrees from the search relies on satisfaction of a *pruning criterion*: that the distance between the query object and the Kth neighbour found so far is less than the distance between the query object, q, and the closest point satisfying the predicate, P, of a subtree under consideration. We refer to this as the minimum distance between q and P, and calculate it using a function, `mindist`:

$$\texttt{mindist}(q, P) = \min\{\texttt{dist}(q, x) \mid P(x)\}$$

Both approaches to K-NN search consist of maintaining a list of the K results found so far, and the distance of the Kth most distant result. When a leaf node is visited, its points are added to the result set, and only the K closest results of the combined set retained. Discarding the outermost results in this situation causes the query radius to contract, so before descent of any subtree, the current query radius (d_K, the distance to the Kth nearest result found so far) is compared to the minimum distance to the subtree predicate under consideration. If d_K is less than this minimum distance, the subtree is certain not to contain any of the K nearest neighbours and can be pruned from the search. Figures 5.10b and 5.10c show the two situations when the region R, representing the predicate $\lambda x.(x \in R)$, respectively cannot and can be pruned from the search for the five nearest neighbours of q.

A sensible modification to the depth-first approach to K-NN searching is to order descent from



(a) $d_R = \texttt{mindist}\,(q, \lambda x.x \in \mathrm{R})$; $d_S = \texttt{mindist}\,(q, \lambda x.x \in \mathrm{S})$

(b) $d_R < d_K < d_S$; S can be pruned from the search.

(c) $d_K < d_R$ and $d_K < d_S$; both R and S can be pruned from the search.

Figure 5.10: Use of `mindist` to prune regions from a K-NN search.

entries within a node, so that those more likely to contain a required result, *i.e.* those of lowest `mindist`, are visited first, allowing the search radius to contract more rapidly. The best-first approach extends this ordering over a greater set of entries than those in a single node; the set of all entries in all nodes visited so far. This means that, if on visiting a node it is found that some of its entries are actually less promising than others already encountered but not explored, their processing can be postponed, rather than requiring the entire subtree to be processed at once. This approach was shown to be optimal in [6], in the sense that it visits exactly the set of nodes that would be visited if d_K were known in advance and the query implemented as that of a fixed region.

The implementation of best-first K-NN searching in the RVFS-tree is effectively the same as that seen in other non-VFS structures, for example the R- or R*-tree. The algorithm for the VFS-tree is almost exactly the same, except that each tree entry that is enqueued must be accompanied by sufficient information to construct (lazily) the corresponding RVFS branch when it is dequeued and explored. The necessary information that must be enqueued is given by the parameters to the `reduce` operation: the address of a VFS-node, its global predicate, and the pending set of elevated entries to be carried down with that entry.

The technique of placing an entry from a tree into an auxiliary structure until it can later be processed is familiar here from our use of the stack for reduction tuples and query tuples. Use of a stack in combination with the K-NN pruning criterion described here gives us the depth-first query algorithm, while replacement of the stack with a priority queue yields the best-first algorithm; the best-first algorithm is given in figure 5.11. We use the expression $\mathbf{T} \blacktriangleright \mathbf{S}$ to mean "the priority queue resulting from enqueueing all the elements of list $\mathbf{T}$ in priority queue $\mathbf{S}$", and maintain two queues, one of the (up to) K nearest neighbours found so far, and another of query tuples containing VFS-entries yet to be explored.

Given a query point q, the priority queue $\mathbf{R}$ of query results is a list, $[\langle k_1, v_1\rangle, \ldots, \langle k_n, v_n\rangle]$ with the property that:

$$\forall \langle k_i, v_i\rangle \in \mathbf{R}_{1 \cdot\cdot |\mathbf{R}|-1}.\ \texttt{dist}\,(q, k_i) \leqslant \texttt{dist}\,(q, k_{i+1})$$

The 'enqueue all' operation, denoted $\blacktriangleright$, enqueues a list of points $\mathbf{P}$ into priority queue $\mathbf{R}$ such that $\mathbf{R}' = \mathbf{P} \blacktriangleright \mathbf{R}$ has the same property. Similarly, a priority queue, $\mathbf{S}$, is a list of query tuples, $[\langle r_1, \mathbf{G}_1, \mathscr{Q}_1\rangle, \ldots, \langle r_n, \mathbf{G}_n, \mathscr{Q}_n\rangle]$, with the property that:

$$\forall \langle r_i, \mathbf{G}_i, \mathscr{Q}_i\rangle \in \mathbf{S}_{1 \cdot\cdot |\mathbf{S}|-1}.\ \texttt{mindist}\,(q, \mathscr{Q}_i) \leqslant \texttt{mindist}\,(q, \mathscr{Q}_{i+1})$$

and given a list of entries $\mathbf{E}$, $\mathbf{S}' = \mathbf{E} \blacktriangleright \mathbf{S}$ shares this property.

$$\left\langle \mathtt{knnQuery}\,(v, K, q)\,,\, \sigma \right\rangle \rightsquigarrow \left\langle \mathtt{knq}\,(K, q)\,,\, [\langle v, [\,], \mathtt{True}\rangle]\,,\, [\,]\,,\, \sigma \right\rangle \tag{6.1}$$

$$\left\langle \mathtt{knq}\,(K, q)\,,\, [\,]\,,\, \mathbf{R}\,,\, \sigma \right\rangle \rightsquigarrow \left\langle \mathtt{results}\,(\mathbf{R})\,,\, \sigma \right\rangle \tag{6.2}$$

$$\begin{aligned} &\left\langle \mathtt{knq}\,(K, q)\,,\, \langle \_, \_, \mathscr{Q}\rangle :: \mathbf{S}\,,\, \mathbf{R}\,,\, \sigma \right\rangle \rightsquigarrow \left\langle \mathtt{results}\,(\mathbf{R})\,,\, \sigma \right\rangle \\ &\qquad \text{if } |\mathbf{R}| \geqslant K \wedge \mathtt{dist}\,(k, q) \leqslant \mathtt{mindist}\,(q, \mathscr{Q}) \\ &\qquad \text{where } \langle k, \_\rangle = \mathbf{R}_K \end{aligned} \tag{6.3}$$

$$\begin{aligned} &\left\langle \mathtt{knq}\,(K, q)\,,\, \langle \_, \_, \mathscr{Q}\rangle :: \mathbf{S}\,,\, \mathbf{R}\,,\, \sigma \right\rangle \rightsquigarrow \left\langle \mathtt{knq}\,(K, q)\,,\, \mathbf{S}\,,\, \mathbf{R}'_{1\cdot\cdot\min(K,|\mathbf{R}'|)}\,,\, \sigma \right\rangle \\ &\qquad \text{if } \sigma(r) = \mathtt{L}\,(\mathbf{E}) \\ &\qquad \text{where } \mathbf{R}' = \mathtt{filter}\,(\mathscr{Q}, \mathbf{E}) \blacktriangleright \mathbf{R} \end{aligned} \tag{6.4}$$

$$\begin{aligned} &\left\langle \mathtt{knq}\,(K, q)\,,\, \langle r, \mathbf{G}, \mathscr{Q}\rangle :: \mathbf{S}\,,\, \mathbf{R}\,,\, \sigma \right\rangle \rightsquigarrow \left\langle \mathtt{knq}\,(K, q)\,,\, \mathbf{T} \blacktriangleright \mathbf{S}\,,\, \mathbf{R}\,,\, \sigma \right\rangle \\ &\qquad \text{where } \sigma(r) = \mathtt{I}\,(L, \mathbf{V}) \\ &\qquad \text{and } \mathbf{E} = \mathtt{reinterpret}\,(\mathbf{V}, \mathbf{G}) \oplus \mathtt{reinterpret}\,(\mathbf{G}, \mathbf{V}) \\ &\qquad \text{and } \mathbf{T} = [\ \langle s, \mathbf{G}', \mathscr{Q} \wedge p\rangle \mid \langle L, p, s\rangle \in (\mathtt{filter}\,(\mathscr{Q}, \mathbf{E})) \wedge \mathbf{G}' = \mathtt{filter}\,(\mathscr{Q} \wedge p, \mathtt{lev}_{<L}\,(\mathbf{E}))\,] \end{aligned} \tag{6.5}$$

Figure 5.11: The VFS-tree `knnQuery` algorithm

The `knnQuery` configuration is of the form:

$$\left\langle \mathtt{knq}\,(K, q)\,,\, \mathbf{S}\,,\, \mathbf{R}\,,\, \sigma \right\rangle$$

Note that the single query command `knq` has two arguments; K, required to trim the list of results to size, and q, the query point. This is required here in a way that it was not for `rQuery` because the query region must frequently be updated. The ruleset for `knnQuery` is given in figure 5.11.

Transition 6.1 bootstraps the operation in the same was as does transition 5.1 for `rQuery`, with three differences:

- the query tuple for the VFS-root is placed in a priority queue, rather than in a stack;
- the result set is initialised as an empty priority queue;
- the input predicate, as in the case of `reduce`, is `True`. This reflects the fact that the query starts with an infinite radius and region, only contracting as results are found.

Unlike `rQuery`, termination here is not usually determined by emptiness of the query tuple queue, but when the first entry in the tuple queue is at a greater distance from the query point than the Kth entry in the result queue. For this reason, the conditional ordering requires transition 6.3 to be placed before the rules for tree descent. (Transition 6.2 gives the termination case when the queue is empty, and is placed here for clarity, to keep the terminating transitions together). Both terminating transitions rewrite into an external `results` configuration. Transition 6.4 collects leaf entries (points) into the result set and trims it to length K, while transition 6.5 specifies the reading of an internal node and the enqueueing of its entries with the appropriate pending sets. Note that, for simplicity, we describe the pending set as being composed of entries that intersect the local predicate of the primary entry,

$$\mathtt{filter}\,(\mathscr{Q} \wedge p, \mathtt{lev}_{<L}\,(\mathbf{E}))$$

although this could be limited further as the query region contracts during search execution, by replacing $\mathscr{Q}$ with the region described by the query point q and the current query radius, $d_K = \mathtt{dist}\,(q, \mathbf{R}_K)$. This does not mean, however, that subtrees are explored unnecessarily, merely that enqueued pending sets may turn out to contain entries that ultimately are not required.

5.2.3 Lazy RVFS-tree generation

In section 4.6.6 we suggested that approaches to search in VFS-trees consist of lazy generation of the corresponding RVFS-tree so that it can be searched. Generation of the RVFS-tree is lazy in the sense that the VFS-tree is reduced only across the region of space required for the search, but also in the sense that the RVFS-tree is never generated physically. RVFS-nodes are instead constructed as required, during descent of the VFS-tree, from the primary entries in each VFS-node and those introduced by the incoming pending set.

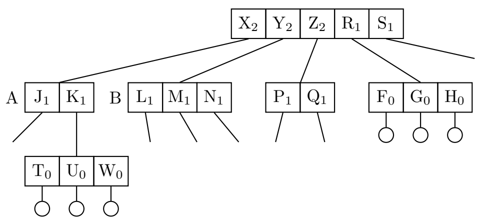
Figure 5.12a shows the VFS-tree used in the search of region $\mathscr{Q}$, described by the query trace of figure 5.9, and figure 5.12b gives the RVFS-fragment required to answer that search. Figure 5.12c illustrates the lazy generation of this RVFS-fragment, at runtime, from the underlying VFS-tree; we refer to this as a *runtime VFS-tree.* In runtime VFS-tree diagrams, a node's contents are shown in blocks of common entry type, from left to right, as follows:

- Local primary entries are shown in the usual way — an entry's label appears in a solid box, subscripted with the entry level number;
- Any primary entries introduced from the incoming pending set are shown in braces;
- Local elevated entries appear is dashed boxes and without children shown — this is merely to indicate their physical location;
- If any non-primary entries are found in the incoming pending set, they are shown, without children, in braces.

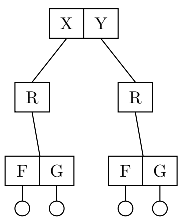
In the root of figure 5.12c, we see three local primary entries, an empty incoming pending set (empty because it is the root of the tree) and two local elevated entries. Node A', the child of entry X_2, has two local primary entries, one incoming primary entry, and no elevated entries, local or otherwise. The explicit RVFS-fragment required to answer a search can be extracted from the runtime VFS-tree by reading upwards from the leaves required to answer the search. In the search for region $\mathscr{Q}$, the children of entries F_0 and G_0 were required — these each appear twice in the leaf level of figure 5.12c. Reading upwards, we find the parent of each $\{\mathrm{F}_0, \mathrm{G}_0\}$ block to be an R_1 entry, and the parent of each of these to be entries X_2 and Y_2 respectively; this is precisely the RVFS-fragment given in figure 5.12b.

The effect of this approach is that the set of entries in a runtime VFS-tree generated during a search is often larger than the set of entries contained in the RVFS-fragment formally required to answer the search — the runtime VFS-tree consists of every entry required for the search and every primary entry found locally in each VFS-node visited. It is important to recognise, however, that the primary entries in the runtime VFS-tree do not necessarily constitute a full reduction of the on-disk VFS-tree:

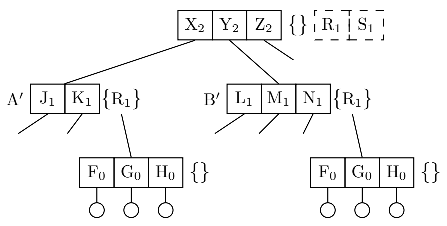
- A runtime VFS-node contains all primary entries from the corresponding static VFS-node, but `reduce` might physically remove some of these if the VFS-node were a descendant of an elevated entry. Although such entries remain physically in the runtime VFS-node, they are still not explored — they are pruned logically using the search predicate. In our example, notice that the runtime VFS-node A' contains entries J_1, K_1 and R_1, while the child of X in the RVFS-fragment of figure 5.12b contains only entry R.



(a) VFS-tree searched for region $\mathscr{Q}$



(b) RVFS-fragment required to answer search for region $\mathscr{Q}$



(c) Lazy RVFS-fragment produced when generating the formal RVFS-fragment in figure 5.12a

Figure 5.12: Lazy RVFS-tree generation during query execution.

- The runtime VFS-node may not contain all primary entries that would be introduced in the pending set for `reduce`, but only those that intersect the search predicate. Observe that the root of figure 5.12c contains elevated entries R_1 and S_1, but only entry R_1 appears as a primary pending entry in level 1.

As we shall see in section 5.3.1, the second point is critical — the fact that the runtime VFS-node contains more entries than the formal RVFS-fragment must not be taken to mean that it is a true RVFS-fragment covering a larger region.

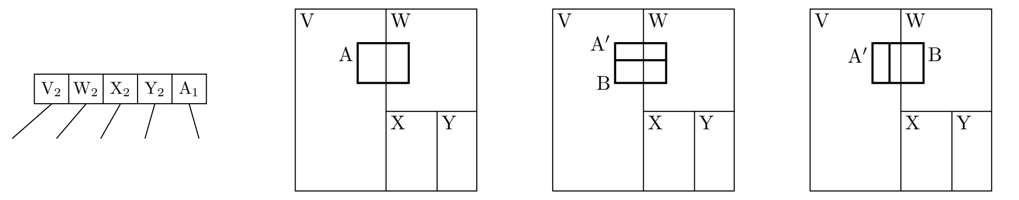
The advantage of this representation is that it combines the physical VFS-structure and the logical RVFS-structure in a single diagram. This enables us to see clearly how a query is answered by the RVFS-tree, whilst also indicating how modifications to the RVFS-structure must be translated back into the underlying VFS-tree.

5.3 Demotion

We have postponed our discussion of the VFS-tree insertion algorithm so far to allow prior presentation of the fixed-extent query approach. This is because, as we described in section 4.6, insertion has elements of exact-match searching, when first inserting a point, and fixed-region searching, when split of an elevated node requires one or both parent entries to be demoted from their elevated position(s). Before we assemble these and other components into the full insertion algorithm, we describe some features of demotion that require additional care.

Figure 5.13a shows a root fragment of a VFS-tree and figure 5.13b the associated spatial decomposition. We assume a primary node capacity of 4 entries, and observe that entry A_1 has been elevated into the root on account of its being virtually split across entries V_2 and W_2. If the child of A_1 overflows, the node may be split, horizontally or vertically, in one of four ways:

- horizontally, as in figure 5.13c, in which case both entries resulting, A_1' and B_1, remain virtually split at level 2;



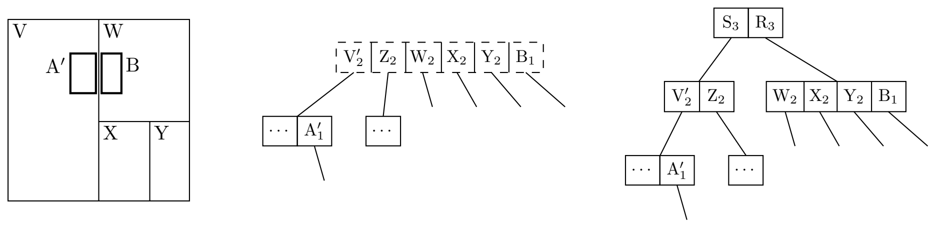
(a) Level 2 root node contains one elevated entry

(b) Geometry of entries in the root node

(c) Possible horizontal split of the child of entry A_1, permitting no demotions.

(d) Possible vertical split of the child of entry A_1, permitting one demotion.

Figure 5.13: Two possible split geometries for an elevated KDB-VFS tree entry.



(a) Possible vertical split of the child of entry A_1, permitting two demotions.

(b) Demotion of entry A'_1 causes node split and promotion, leading to root overflow, indicated by the node's dashed boundary.

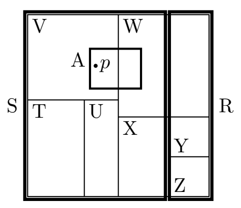
(c) Split of the overflowing root, deferring demotion of entry B_1.

Figure 5.14: Split of the child of entry A_1 requires two demotions, sequential execution of which requires care.

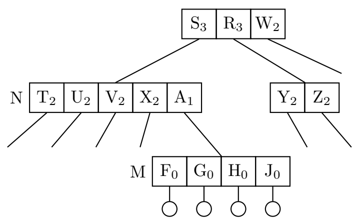
- vertically, to the left of the V_2/W_2 boundary (see figure 5.13d), in which case entry A'_1 is no longer virtually split and should be demoted into the child of V_2;
- vertically, to the right of the V_2/W_2 boundary, in which case entry B_1 is no longer virtually split and should be demoted into the child of W_2;
- vertically, and exactly on the V_2/W_2 boundary, as in figure 5.14a, in which case neither A'_1 nor B_1 are virtually split. Both should be demoted.

When demoting either A'_1 or B_1, because we are in the root, and because the entry or entries undergoing demotion belong in the level immediately below the root (so can be demoted no further than that), we clearly require no pending entries. If demoting a single entry, we do so directly, perhaps causing the node into which the demotion proceeds to overflow as in figure 5.14b after the demotion of A'_1, in which case primary entries V'_2 and W_2 are posted back into the root causing it in turn to overflow and split in the usual way. If, however, *both* A'_1 and B_1 require demotion, care must be taken: if demotion of A'_1 causes root overflow (as in figure 5.14b — the dashed boundary here indicates that the node is overflowing), the root must either be permitted temporarily to exceed its primary capacity while the second demotion is performed (perhaps resulting in a further split), or the demotion of B_1 must be deferred until after the root split is complete. Figure 5.14c shows the result of splitting the root first; entry B_1 is still awaiting demotion.

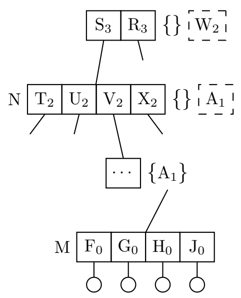
When dealing with demotion from an internal (*i.e.* non-root, directory) node, the situation can become still more complicated. Figure 5.15a shows a spatial decomposition, with the associated VFS-tree fragment given in figure 5.15b. The exact decomposition of region A into subregions F, G H and J is not shown, but we assume that the point p, also marked on figure 5.15a, is contained



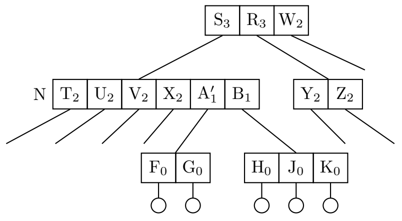
(a) Spatial decomposition prior to the insertion of point p.



(b) VFS-tree before the insertion of p.



(c) Runtime VFS-fragment generated in exact-match search for p.



(d) Insertion of p has caused split of node M — one or both of entries A'_1 or B_1 may now be demotable.

Figure 5.15: Insertion of point p causes split of an elevated node, possibly permitting entry demotion.

in region F. We further assume internal nodes to have a primary capacity of four entries; node N is full. Entry A_1 is virtually split at level 2, across entries V_2 and W_2; note, however, that entry W_2 is itself virtually split at level 3, so has been elevated into the root node.

Consider now the insertion of point p into the VFS-tree of figure 5.15b. Exact-match search for the node into which p should be inserted proceeds from the root into node N (the child of entry S_3), into the child of entry V_2 (carrying pending entry A_1), then into node M (the child of A_1) and finally into the leaf child of entry F_0. This is shown in the runtime VFS-tree fragment given in figure 5.15c. We assume that insertion of p into the child of F_0 then causes that leaf to overflow, posting into node M, which itself also overflows. This causes two new entries, A'_1 and B_1, to be posted into node N, replacing entry A_1 and resulting in the VFS-tree of figure 5.15d. This presents two questions for consideration; firstly, which (if either) of A'_1 and B_1 are demotable from N? Secondly, if both are demotable, what happens if one or both demotions causes further overflow? In particular, notice that one demotion would be made into the child of entry W_2 — but this entry is above the VFS-node from which the demotions are to be made. How do we organise a demotion, from node N, but that might be required to return entries into the root of the tree? We examine these issues in sections 5.3.1 and 5.3.2.

5.3.1 Demotability

Decisions of demotability depend ultimately on having the correct pending set available for consideration. Consider once again the insertion of point p into the VFS-tree of figure 5.15b. This provoked the split of node M, causing entries A'_1 and B_1 to replace A_1 in node N, and producing the VFS-tree of figure 5.15d. We must now establish which of these two new level 1 entries is still virtually split.

We demonstrate here that to attempt this by examination of local primary entries alone is

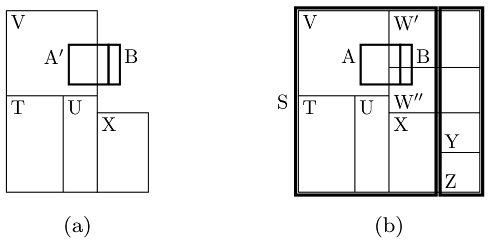
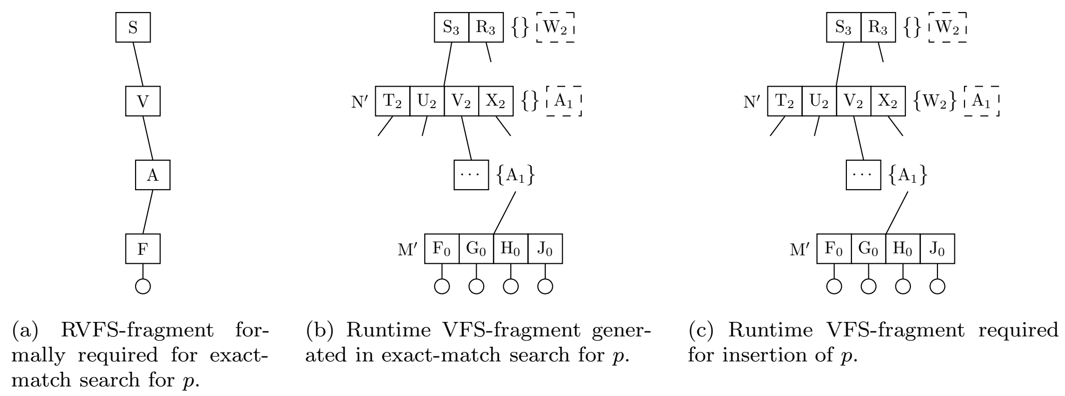


Figure 5.16: Region B may appear not to be virtually split from local examination of node N's entries, but with full region information can be seen to be virtually split across regions W′ and W″.



(a) RVFS-fragment formally required for exact-match search for p.

(b) Runtime VFS-fragment generated in exact-match search for p.

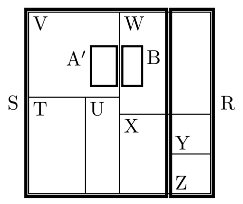
(c) Runtime VFS-fragment required for insertion of p.

Figure 5.17: Insertion of a point into a VFS-tree requires construction of a larger RVFS-fragment than would be required, or even generated, merely to search for the same point.

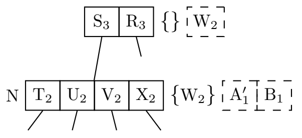
insufficient. Assuming a possible split geometry for region A, figure 5.16a shows the regions represented in node N prior to demotion (A′ and B are the regions associated with entries A'_1 and B_1 respectively). From the figure we can see that region A′ remains virtually split, but we cannot say either way about B — because the level 2 regions across which it may (or may not) be virtually split are not available for examination. Figure 5.16b shows the result of a possible split of region W, prior to the insertion of p, such that region B remains virtually split at level 2. This illustrates that deciding the demotability of an entry from a node requires more primary entries than those that are available locally, or provided in the pending set used for the insertion of p. The presentation of the BV-tree in [19] does not allude to this.

The correct choice of pending set can be understood directly in terms of the runtime VFS-tree constructed in support of the insertion of p. Figure 5.17a shows the RVFS-fragment formally required for exact-match search for p, and figure 5.17b repeats the runtime VFS-fragment actually produced by the generation of 5.17a. Entries T_2, U_2 and X_2 are absent from the formal RVFS-fragment, but appear in runtime VFS-node N′ because they are present locally in VFS-node N. Notice, however, that N′ is not a valid reduction of node N across its entire region (S), because region W is not represented.

Insertion of p into the runtime VFS fragment in figure 5.17b causes the child of entry F_0 to overflow, which in turn causes VFS-node M to overflow, splitting, and posting entries A'_1 and B_1. These are posted first into the parent of node M, and then posted again into node N, because this is the physical location of entry A_1, which they are to replace. It is at this point that we must establish their demotability, and it is clear from figure 5.17b that entry W is unavailable for



(a) Split of region A into regions A′ and B.



(b) Runtime VFS-fragment after promotion of A'_1 and B_1 and prior to their demotion.

Figure 5.18: Split of region A into regions A′ and B renders both associated elevated entries demotable.

examination. Runtime VFS-node N represents, in total, the space $T \cup U \cup V \cup X$; region A is not contained in this space. We therefore cannot consider the demotion of entry A'_1 or B_1 into the children of node N, because the node does not represent the space occupied by regions A′ and B. The problem is that, when selecting pending entry A_1 from VFS-node N, we were only actually interested in the portion of region A that lies within region V — $V \cap A$ — and consequently generated the runtime VFS-fragment for that region only. When the child of A_1 splits, however, we are required to consider the whole of region A, so require a different runtime VFS-fragment, one in which region A is represented.

In general, when inserting a point, we are not able to predict which subregions of the RVFS-branch into which insertion proceeds might require reorganisations of this kind — when encountering entry W_2 on inserting p into the root of the VFS-tree in figure 5.15b, we did not know that it would later be required to cover region A. We must therefore include in a pending set for insertion, not only each entry that intersects the point undergoing insertion, but every elevated entry that intersects the entire subtree region of each node into which we descend. This permits us to generate, in full, the RVFS-tree branch down which the insertion proceeds (rather than merely the fragment into which the insertion takes place). Figure 5.17c illustrates the full runtime VFS-fragment to be generated during insertion of p — notice that because region W intersects region S, it was brought down from the VFS-root in a pending set and is now being considered as an incoming primary entry at node N. This means that when entry A_1 splits, region W is available in node N to assess correctly the demotability of A'_1 and B_1. By definition, the region indexed by a node must contain every entry in the node — including elevated entries — so by populating the RVFS-node with its full complement of primary entries, demotability of any elevated entry therein can correctly be assessed.

5.3.2 The demote queue

Continuing with our example from the previous section, we assume that the insertion of point p into the VFS-tree of figure 5.15b has caused node M to split, replacing entry A_1 with entries A'_1 and B_1. We now assume that the geometry of the split of region A is such that the boundary between the regions into which it is split, A′ and B, lies along that between regions V and W, as shown in figure 5.18a. Having collected pending sets as appropriate to construct the runtime VFS-fragment given in figure 5.17c, we are now able to establish that entries A'_1 and B_1 are both demotable, into the subtrees rooted on entries V_2 and W_2 respectively.

This presents some potential difficulties related to the fact that, although we are operating on a runtime VFS-tree fragment, changes to that fragment must be incorporated into the static, on-disk VFS-tree. Figure 5.18b shows the upper levels of the runtime VFS-tree after the replacement of entry A_1 in node N. The original demotion has returned entries A'_1 and B_1 into node N, so could we now, from node N, commence demotion of the two new entries, embedding the demotion routines within the over-arching insertion of p?

The difficulty here is that either demotion could cause further overflow, and node N is already full. Suppose we choose to demote entry B_1 into the subtree rooted on W_2. If the child of W_2 overflows, this posts into the root of the tree — the physical location of W_2 — so has stepped outside the tree traversal of the original insertion of p, which has yet to terminate. Worse still, one or both of the entries replacing W_2 in the root may also be demotable, potentially requiring us to demote an entry into N, which will itself then overflow. Do we then split node N, keeping track of entry A'_1, which is still awaiting demotion? Or should we allow node N to become overfull (to exceed its primary capacity temporarily), pending demotion of A_1? Furthermore, recall that we have interrupted our return from the insertion of p to execute these demotions. The structure of this algorithm is far from clear if, having paused the return from insertion at a node, we first promote entries above that node, and then demote entries through the same node.

Because node N is full, demoting entry A'_1 first poses a similar set of questions; it too could cause overflow of the child of V_2, which will subsequently cause node N to overflow. The critical point is that demotion of either entry causes a change to the static VFS-tree — irrespective of the nature of that change, we must recognise immediately that the runtime VFS tree in figure 5.18b is no longer an accurate lazy reduction of the VFS-tree on disk. Demotion of the first entry must therefore either modify the static VFS-tree *and* the runtime VFS-tree so that they remain synchronised, or we must call a halt after the first demotion, allowing the insertion call to return to the root, and starting the second demotion from scratch by demoting it from the root of the VFS-tree. Given that even some single demotions — such as that of B_1 into W_2 — are fraught with complexities (like interference with the interrupted insertion routine that first called the demotion), and that we must in any case make provision for an 'offline' approach to the second demotion of a pair, we take the offline approach as standard, using an auxiliary structure we call a *demote queue*.

When an entry is found to be demotable, it is copied to the demote queue, but no further action is taken. The original operation (for example, an insertion or another demotion) is permitted to finish normally, after which we attempt to demote the entry at the head of the demote queue. The insert operation is only considered to be complete when the demote queue is empty. Entries in the queue are described as having been 'scheduled for demotion', and processing of each scheduled entry proceeds as follows:

- Remove the copy entry from the demote queue;
- Use the information in the copy entry to find the original entry in the tree;
- If it is found, and if it is still demotable, begin physical demotion of the original entry.

We say 'if it is still demotable', because the presence of an entry in the demote queue does not necessarily indicate its demotability — merely that it was demotable at the time at which it was enqueued. Suppose, as in our example, that insertion of point p causes entries B_1 and A'_1 to be scheduled for demotion, in that order. We assume now that B_1 is demoted, into the child of R_3 and thence into that of W_2, causing that node to overflow. W_2 is then replaced in the root with two new entries, W'_2 and W''_2, both of which are also scheduled for demotion. The demote queue now contains $[A'_1, W'_2, W''_2]$; A'_1 and one other element of the queue, say W'_2, are scheduled for demotion into node N, rooted on entry S_3. The demotion of entry A'_1, however, may cause node N to split, replacing entry S_3 with two new entries. When we then come to demote W'_2, therefore, we may find that it is, once again, virtually split at level 3.

In the case of the BV-tree, the demote queue approach may require a node to accommodate more elevated entries than the BV-tree elevation limit guarantee would permit, while one or more of those entries awaits demotion. This is acceptable, from the standpoint that the tree is still mid-insertion in this state, but requires a node to have the physical capacity to accommodate those entries. The linked-list approach to node implementation provides such capacity, but a level-multiplied node implementation would be at risk of failure.

5.3.3 Demotion termination and insertion cost

Finally, we consider the issue of termination and insertion cost: firstly, a question about when an individual demotion will terminate, and secondly, can we be sure that an insertion-triggered cascade of demotions will not continue indefinitely? Recall from section 4.6.3 that an elevated entry may require further elevation at subsequent node splits; figure 5.19 illustrates a case in which already-elevated entry A_1 is elevated a second time, into level 3. Notice, however, that the entry remains also virtually split at level 2. If the child of A_1 splits, it is therefore possible that the resulting pair of entries will no longer be split at level 3, but still be virtually split at level 2. This means that demotion of such entries is possible from level 3, but that it may be required to terminate in level 2 — the entries may not be demoted as far as their natural level, 1. When demoting an entry into a node, demotion terminates either:

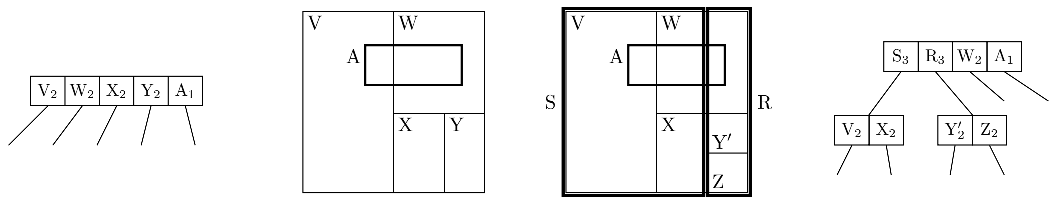
- when the demotee has reached its natural level, or;
- when the demotee is found to be virtually split at the level of the node into which it has been demoted.

Deciding the latter at a given level requires us to examine every primary entry that intersects the demotee, but this needs no special treatment. As we described in section 5.3.1, we must always construct the full RVFS-branch into which an insertion (or demotion) takes place, in case subsequent overflow triggers further potential demotions. This means that the primary entries, local or elevated, necessary to enable us to decide if a demotee remains virtually split above its natural level, are already made available to us in the node into which the demotion takes place.

The possibility that each insertion may trigger demotions causes the average cost of inserting a point into a VFS-tree to rise. Freeston [20] suggests that demotions in the BV-tree might be postponed until a subsequent insertion passes through the node containing the prospective demotee(s). This strategy is intended to allow the cost of demotions to be amortised over future insertions, but suffers from two principal problems.

First, delaying demotions in this way may break the BV-tree guarantee on the number of elevated entries in a node. Unlike the case in which entries are awaiting a scheduled demotion, however, this means that the guarantee may no longer apply even after an insertion operation is complete. (Notice that this too could cause a BV-tree implementation using nodes of level-multiplied size to fail).

Second, although identification of demotees would be possible during tree descent for an insertion, any demotion required could not be executed in parallel with the insertion for the same



(a) Level 2 root node contains one elevated entry.

(b) Geometry of entries in the root node.

(c) Selected split geometry after split of region Y into Y′ and Z requires a root split.

(d) Execution of selected split requires elevation of entry W_2 and further elevation of already-elevated entry A_1.

Figure 5.19: Split of a node containing elevated entries may require those entries to be elevated further.

reason that each demotion must proceed from the root — one operation may modify the underlying VFS-tree, causing the runtime VFS-fragment to need to be refreshed from disk.

The cost of insertion followed by demotions is not, however, unbounded. A cascade of demotions executed sequentially after an initial insertion is certain to terminate for a simple reason: demotion of an entry of level n can trigger, at most, two further demotions, and only of entries of level $n+1$. This is because the demotion can only cause overflow if the entry reaches its natural level, n — it is only then that a node split will be caused, posting two entries of level $n+1$. If the node into which the demotion takes place is also elevated, then one or both of the posted entries may, in turn, be demotable. If every insertion and demotion caused a further pair of demotions in this way, in the worst case a single insertion into a tree of height h could cause 2^{h-2} demotions (assuming that the root does not overflow during the sequence of insertion and demotions). This would require 2 demotions into level 0, 4 into level 1, 8 into level 2 and so on; neither the leaf level, nor that of the root, can be a demotee's natural level.

5.3.4 Demotability in the BV-tree

Our discussion of demotion so far has been in the context of general VFS-trees, but the BV-tree's mode of region representation introduces an issue specific to that structure: If both entries resulting from the (non-buddy) split of an elevated node are scheduled for demotion, their demotability is affected by the order in which they are demoted.

Figure 5.20a shows a collection of holey region descriptions drawn from a single BV-node shown in figure 5.20b, together indicating that entry B_0 is virtually split across entries Y_1 and Z_1. Suppose that the child of entry B_0 overflows, requiring it to split, and that the split of the associated region B is exactly coincident with the boundary between regions Y and Z (*i.e.* the outer boundary of region Z). This means that both entries resulting from the split of the overflowing node (call them B_0^{ou} and B_0^{in}) can be demoted, leaving no elevated entries in node N, as shown in figure 5.20c.

As we observed in section 5.3.2, the presence of an entry in the demote queue indicates that it was demotable at the point at which the demotion was scheduled, but it may no longer be by the time the demote is actually executed. For this reason, an entry undergoing demotion must have its demotability reassessed in every node through which it passes. Suppose now, in our BV-tree example, that entry B_0^{in} is demoted first. Demotion proceeds into node N, at which point the demotability of the entry is checked. Having been found to be demotable, B_0^{in} is then demoted into the child of Z_1, and demotion terminates. B_0^{ou} is now ready for demotion. Once again,

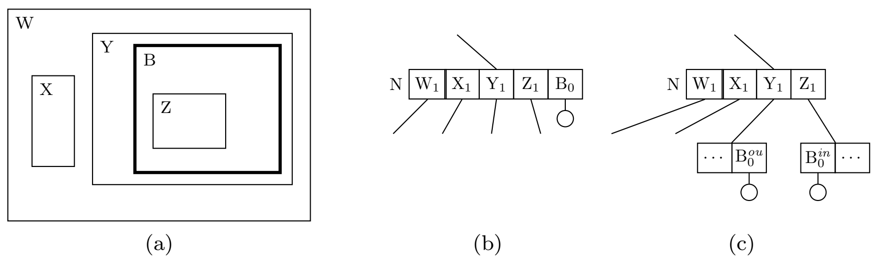


Figure 5.20: Spatial decomposition and associated BV-node. If region B splits exactly along the Z boundary, both entries resulting from the split can be demoted.

demotion proceeds into node N, but when the demotability of B_0^{ou} is checked, the hole that makes it demotable — B^{in} — is no longer represented in the node. This means that entry B_0^{ou} appears not to be demotable, and its demotion terminates in node N, the node from which it originated.

The reverse, however, is not true. If B_0^{ou} is demoted first, entry B_0^{in} is present in N, and B_0^{ou} is found to be demotable into the child of Y_1. When B_0^{in} is demoted subsequently, B_0^{ou} is no longer present in N, but this has no effect; entry B_0^{in} is still found not to be virtually split at level 1, allowing demotion to continue into the child of Z_1. This comes down to the fact that 'holeyness' is not symmetric; region B^{in} is a hole in B^{ou}, but not the reverse.

A BV-specific optimisation for demotion would be to order the demote queue, $\mathbf{S}$, such that, given $\mathbf{S}_i = \langle l, p, r\rangle$ and $\mathbf{S}_j = \langle l, q, s\rangle$, and where $q \subseteq p$, we have $i < j$ — informally, ensuring that holes are demoted last. However, we observe that this does not affect the BV-tree's node occupancy guarantee, for the very reason that if one entry of a pair remains undemoted, it is because it *appears* to be virtually split.

5.4 Insertion

The considerations described for demotion in section 5.3 provide us with the final remaining components required for implementation of a full VFS-tree insertion algorithm. These consist of:

1. a reduction-based exact-match search algorithm to find the location in which a point is to be inserted, described in section 5.2.1 (with points as regions of zero extent), but constructing the full RVFS-branch in case wider modifications are required;

2. a standard post-and-grow approach to handling node overflow and split, but posting into the physical parent node in the case of splitting of an elevated entry, described at the beginning of section 5.1;

3. the determination, in a node into which entries are posted, if one or both entries are elevated and no longer virtually split, in which case one or both are scheduled for demotion, described in section 5.3;

4. the demotion of entries in the demote queue, using the reduction-based region search of section 5.2.1, including checks for continued demotability (described in section 5.3.2) and 'early' termination before the demotee has reached its natural level (described in section 5.3.3);

5. handling overflow caused by demotion, and any subsequent demotions required, as described above.

The algorithm is structurally similar to that of the B+ tree and GiST described in chapters 2 and 3, with a few changes as described above:

- An entry is selected for descent and some information pushed into the stack (in this case including a pending set and global predicate);
- When reaching the leaf, the entry is inserted, producing a `S` or `D` configuration, depending on whether or not the node overflows;
- `S` configurations step back up the tree, while `D` configurations propagate node splits (in this case managing posting of split elevated entries past one or more levels, further overflow or demote scheduling as required);
- When the stack is empty, the first scheduled demotion begins; if the demote queue is also empty, the process terminates.
- Demotions are handled like region insertions, with the difference that each demotion has two phases — one in which the enqueued entry is used to find the original demotee, and a second phase in which that demotee is (potentially) physically demoted.

The `vfsInsert` configuration has the form:

$$\left\langle \mathtt{C} \, , T \, , \pi \, , \mathbf{D} \, , \sigma \right\rangle$$

where `C` is a *vfsInsertCommand*, T is a *vfsInsertTuple*, π is a stack of such tuples, $\mathbf{D}$ is the demote queue and σ is the store. A *vfsInsertTuple*, $\langle r, \mathbf{G}, Q\rangle$, is a reduction tuple used during the insertion algorithm and augments the page identifier information placed on the stack in other insertion algorithms (B+ tree, figure 2.9; GiST, figure 3.5; FS-tree, figure 4.8) with a pending set and global predicate required in the case of VFS-trees.

The grammar for a *vfsInsertCommand* is:

$$\mathit{vfsInsertCommand} \ ::= \ \mathtt{ins}\,(\mathit{LeafEntry}) \mid \mathtt{S} \mid \mathtt{D}\,([\,\mathit{VfsEntry}]) \mid \mathtt{fDem}\,(\,\mathit{VfsEntry}) \mid \mathtt{dem}\,(\,\mathit{VfsEntry})$$

The `ins` command is used to descend to the leaf, and the `S` command to return to the root when no posting is required. The `D` command posts a sequence of entries of minimum length 2 (greater than 2 when elevating virtually split entries). The `fDem` and `dem` commands are equivalent to the `ins` command, but for demotion of a *VfsEntry.* `fDem` carries the enqueued demotee while searching for its physical location on disk; `dem` carries the physical demotee, after its removal from its original physical location.

We introduce the function $\mathtt{lev}_{=L}$, a direct analogue of the function $\mathtt{lev}_{<L}$ introduced in section 5.1; given a list of VFS-entries, $\mathtt{lev}_{=L}$ returns only those of level number equal to L:

$$\mathtt{lev}_{=L}\,(\mathbf{E}) = [\ \langle l, p, r\rangle \ \mid \langle l, p, r\rangle \in \mathbf{E} \wedge l = L\,]$$

In a node of level L, $\mathtt{lev}_{=L}$ identifies the node's primary entries.

Figures 5.21 and 5.22 contain the ruleset for `vfsInsert`. Transition 7.1 bootstraps an insertion, rewriting from an external `vfsInsert` configuration containing the entry undergoing insertion, the address of the root page and the store, into a `ins` configuration, with the root address, its (empty) pending set and global predicate (`True`) pushed onto the stack in an insertion tuple $\langle r, [\,], \mathtt{True}\rangle$.

Transition 7.2 reads an internal node and selects for descent the unique primary entry whose local predicate is satisfied by the point undergoing insertion. Transitions 7.3–7.5 handle insertion of a point into a leaf.

Return to the root of the tree (in the event of no entries being posted) is handled by the `S` $\leadsto$ `S` transition 7.6, while transitions 7.7–7.10 handle the respective possibilities in the event of entries being posted into a node:

- posting from split of an elevated subtree into a node above this one: post again (7.7);
- entries are primary and can be accommodated: rewrite to `S` configuration (7.8);
- entries are primary and node is full: split and post (7.9);
- entries are elevated: accommodate them here and schedule demotions as required (7.10). The $\texttt{lev}_{=L}$ function is used here to count the number of primary entries whose predicates intersect that of an elevated entry; if greater than one, the elevated entry is virtually split and cannot be demoted.

Transitions 7.11 and 7.12 handle termination of the initial insertion or a single demotion in an `S` or `D` term respectively, initiating demotion of the entry at the head of the demote queue. If the demote queue is empty, the entire operation terminates by rewriting to an external `vfs` configuration, containing the root address of the VFS-tree, in either transition 7.13 or 7.14.

Execution of the demotion operation itself is analogous to insertion of a point, but has two distinct phases; one in which the previously-enqueued entry is used to find the demotee's physical location, and a second in which the physical demotion takes place. The net result is as if the demotee had been inserted into the tree from the root; indeed, if the demotion causes a node to split, entries may end up being posted all the way back up the tree and into the root. Transition 7.15 is a step in the search for the demotee on disk — neither the now-dequeued entry, nor any entry that may have replaced it, are found in the node, and a single path for insertion of the demotee is found. Search for the on-disk demotee continues down that single path. In transition 7.16, the demotee is found and physical demotion begins. Notice, in this case, that the demotee is matched against the dequeued entry on the basis of its child node address, but that identification of the demotion path is made using the on-disk predicate for the demotee. This reflects the fact that the demotee may have been altered on disk since the original demotion was scheduled. Transition 7.17 handles the case in which demotion cannot continue, because a prospective demotee is found no longer to be demotable.

Having commenced physical demotion (from transition 7.16), the situation becomes rather more straightforward. Transition 7.16 is a step in the demotion of a physical entry into the tree. This entry has, by now, been found in and removed from a node in a higher level, so demotion looks much more similar to insertion than in transition 7.15. The three possible ways in which this demotion might terminate are handled by transitions 7.7–7.10:

- The demotee has been demoted through at least one level, but is now found to be virtually split at a lower level (still above its own natural level). The entry remains elevated, so overflow is not possible;
- the demotee has reached its natural level, and can be accommodated in the node without overflow;

$$\big\langle \mathtt{vfsInsert}\,(\langle k,v\rangle, r)\,,\,\sigma \big\rangle \quad\rightsquigarrow\quad \big\langle \mathtt{ins}\,(\langle k,v\rangle)\,,\,\langle r, [\,], \mathtt{True}\rangle\,,\,[\,]\,,\,[\,]\,,\,\sigma \big\rangle \tag{7.1}$$

$$\big\langle \mathtt{ins}\,(\langle k,v\rangle)\,,\,\langle r,\mathbf{G},Q\rangle\,,\,\pi\,,\,\mathbf{D}\,,\,\sigma \big\rangle \quad\rightsquigarrow\quad \big\langle \mathtt{ins}\,(\langle k,v\rangle)\,,\,\langle s,\mathbf{G}',Q\wedge p\rangle\,,\,\langle r,\mathbf{G},Q\rangle :: \pi\,,\,\mathbf{D}\,,\,\sigma \big\rangle \tag{7.2}$$

if $\sigma(r) = \mathtt{I}\,(L,\mathbf{V})$

where $\mathbf{E} = \mathtt{reinterpret}\,(\mathbf{V},\mathbf{G}) \oplus \mathtt{reinterpret}\,(\mathbf{G},\mathbf{V})$

and $\langle \_, p, s\rangle = \mathtt{uniq}\,(\mathtt{lev}_{=L}\,(\mathbf{E}), P)$ and $P = \lambda x.$match x as $\langle l,q,t\rangle$ in $q(k)$

and $\mathbf{G}' = \mathtt{filter}\,(Q\wedge p, \mathtt{lev}_{<L}\,(\mathbf{E}))$

$$\big\langle \mathtt{ins}\,(\langle k,v\rangle)\,,\,\langle r,\mathbf{G},Q\rangle\,,\,\pi\,,\,\mathbf{D}\,,\,\sigma \big\rangle \quad\rightsquigarrow\quad \big\langle \mathtt{S}\,,\,\langle r,\mathbf{G},Q\rangle\,,\,\pi\,,\,\mathbf{D}\,,\,\sigma[r\mapsto \mathtt{L}\,(\mathtt{repl}\,(\langle k,v\rangle, i, \mathbf{E}))] \big\rangle \tag{7.3}$$

if $i \leqslant |\mathbf{E}|$

where $\sigma(r) = \mathtt{L}\,(\mathbf{E})$

and $i = \mathtt{first}\,(\mathbf{E},P)$ and $P = \lambda x.$match x as $\langle j,w\rangle$ in $j = k$

$$\big\langle \mathtt{ins}\,(\langle k,v\rangle)\,,\,\langle r,\mathbf{G},Q\rangle\,,\,\pi\,,\,\mathbf{D}\,,\,\sigma \big\rangle \quad\rightsquigarrow\quad \big\langle \mathtt{S}\,,\,\langle r,\mathbf{G},Q\rangle\,,\,\pi\,,\,\mathbf{D}\,,\,\sigma[r\mapsto \mathtt{L}\,(\mathtt{append}\,(\langle k,v\rangle, \mathbf{E}))] \big\rangle \tag{7.4}$$

if $|\mathbf{E}| < \mathit{Max}_L$

where $\sigma(r) = \mathtt{L}\,(\mathbf{E})$

$$\big\langle \mathtt{ins}\,(\langle k,v\rangle)\,,\,\langle r_L,\mathbf{G},Q\rangle\,,\,\pi\,,\,\mathbf{D}\,,\,\sigma \big\rangle \quad\rightsquigarrow \tag{7.5}$$

$$\big\langle \mathtt{D}\,(0, [\langle 0,p_L,r_L\rangle, \langle 0,p_R,r_R\rangle])\,,\,\langle r_L,\mathbf{G},Q\rangle\,,\,\pi\,,\,\mathbf{D}\,,\,\sigma[r_L\mapsto \mathtt{L}\,(\mathbf{E_L})\,, r_R\mapsto \mathtt{L}\,(\mathbf{E_R})] \big\rangle$$

where $\sigma(r_L) = \mathtt{L}\,(\mathbf{E})$

and $\langle p_L, \mathbf{E_L}, p_R, \mathbf{E_R}\rangle = \mathtt{splitL}\,(\mathtt{append}\,(\langle k,v\rangle, \mathbf{E}))$

and $r_R = \mathtt{fresh}(\sigma)$

$$\big\langle \mathtt{S}\,,\,T\,,\,T' :: \pi\,,\,\mathbf{D}\,,\,\sigma \big\rangle \quad\rightsquigarrow\quad \big\langle \mathtt{S}\,,\,T'\,,\,\pi\,,\,\mathbf{D}\,,\,\sigma \big\rangle \tag{7.6}$$

$$\big\langle \mathtt{D}\,(L',\mathbf{E})\,,\,\langle s,\mathbf{G}',Q'\rangle\,,\,\langle r,\mathbf{G},Q\rangle :: \pi\,,\,\mathbf{D}\,,\,\sigma \big\rangle \quad\rightsquigarrow\quad \big\langle \mathtt{D}\,(L',\mathbf{E})\,,\,\langle s,\mathbf{G}',Q'\rangle\,,\,\pi\,,\,\mathbf{D}\,,\,\sigma \big\rangle \tag{7.7}$$

if $\langle \_, \_, s\rangle \notin \mathbf{V}$

where $\sigma(r) = \mathtt{I}\,(L,\mathbf{V})$

$$\big\langle \mathtt{D}\,(L',\mathbf{E})\,,\,\langle s,\mathbf{G}',Q'\rangle\,,\,\langle r,\mathbf{G},Q\rangle :: \pi\,,\,\mathbf{D}\,,\,\sigma \big\rangle \quad\rightsquigarrow\quad \big\langle \mathtt{S}\,,\,\langle r,\mathbf{G},Q\rangle\,,\,\pi\,,\,\mathbf{D}\,,\,\sigma[r\mapsto \mathtt{I}\,(L,\mathbf{V}'')] \big\rangle \tag{7.8}$$

if $L' = L \wedge |\mathtt{lev}_{=L}\,(\mathbf{V})| < \mathit{Max}_I$

where $\sigma(r) = \mathtt{I}\,(L,\mathbf{V})$

and $\mathbf{V}' = \mathtt{del}\,(\mathtt{first}\,(\mathbf{V},P), \mathbf{V})$ and $P = \lambda x.$match x as $\langle m,q,t\rangle$ in $t = s$

and $\mathbf{V}'' = \mathtt{reinterpret}\,(\mathbf{V}',\mathbf{E}) \oplus \mathtt{reinterpret}\,(\mathbf{E},\mathbf{V}')$

$$\big\langle \mathtt{D}\,(L',\mathbf{E})\,,\,\langle s,\mathbf{G}',Q'\rangle\,,\,\langle s_L,\mathbf{G},Q\rangle :: \pi\,,\,\mathbf{D}\,,\,\sigma \big\rangle \quad\rightsquigarrow \tag{7.9}$$

$$\big\langle \mathtt{D}\,(L+1, [\langle L+1,q_L,s_L\rangle, \langle L+1,q_R,s_R\rangle] \oplus \mathbf{E}')\,,\,\langle s_L,\mathbf{G},Q\rangle\,,\,\pi\,,\,\mathbf{D}\,,\,\sigma[s_L\mapsto \mathtt{I}\,(L,\mathbf{V_L})\,, s_R\mapsto \mathtt{I}\,(L,\mathbf{V_R})] \big\rangle$$

if $L' = L$

where $\sigma(s_L) = \mathtt{I}\,(L,\mathbf{V})$

and $\mathbf{V}' = \mathtt{del}\,(\mathtt{first}\,(\mathbf{V},P), \mathbf{V})$ and $P = \lambda x.$match x as $\langle m,q,t\rangle$ in $t = s$

and $\langle q_L, \mathbf{V_L}, q_R, \mathbf{V_R}, \mathbf{E}'\rangle = \mathtt{splitI}\,(\mathtt{reinterpret}\,(\mathbf{V}',\mathbf{E}) \oplus \mathtt{reinterpret}\,(\mathbf{E},\mathbf{V}'))$

and $s_R = \mathtt{fresh}(\sigma)$

$$\big\langle \mathtt{D}\,(L',\mathbf{E})\,,\,\langle s,\mathbf{G}',Q'\rangle\,,\,\langle r,\mathbf{G},Q\rangle :: \pi\,,\,\mathbf{D}\,,\,\sigma \big\rangle \quad\rightsquigarrow \tag{7.10}$$

$$\big\langle \mathtt{S}\,,\,\langle r,\mathbf{G},Q\rangle\,,\,\pi\,,\,\mathbf{D}\oplus\mathbf{E_D}\,,\,\sigma[r\mapsto \mathtt{I}\,(L,\mathbf{V}'\oplus\mathbf{E}')] \big\rangle$$

where $\sigma(r) = \mathtt{I}\,(L,\mathbf{V})$

and $\mathbf{V}' = \mathtt{reinterpret}\,(\mathtt{del}\,(\mathtt{first}\,(\mathbf{V},P), \mathbf{V})\,, \mathbf{G}\oplus\mathbf{E})$ and $P = \lambda x.$match x as $\langle m,q,t\rangle$ in $t = s$

and $\mathbf{G}' = \mathtt{reinterpret}\,(\mathbf{G}, \mathbf{V}'\oplus\mathbf{E})$

and $\mathbf{E}' = \mathtt{reinterpret}\,(\mathbf{E}, \mathbf{V}'\oplus\mathbf{G})$

and $\mathbf{E_D} = [\ \langle m,q,t\rangle\ \mid \langle m,q,t\rangle \in \mathbf{E}' \wedge |\mathtt{filter}\,(q, \mathtt{lev}_{=L}\,(\mathbf{V}'\oplus\mathbf{G}'))| = 1]$

$$\big\langle \mathtt{S}\,,\,\langle r,\_,\_\rangle\,,\,[\,]\,,\,d :: \mathbf{D}\,,\,\sigma \big\rangle \quad\rightsquigarrow\quad \big\langle \mathtt{fDem}\,(d)\,,\,\langle r, [\,], \mathtt{True}\rangle\,,\,[\,]\,,\,\mathbf{D}\,,\,\sigma \big\rangle \tag{7.11}$$

$$\big\langle \mathtt{D}\,(L,\mathbf{E})\,,\,\langle r,\_,\_\rangle\,,\,[\,]\,,\,d :: \mathbf{D}\,,\,\sigma \big\rangle \quad\rightsquigarrow\quad \big\langle \mathtt{fDem}\,(d)\,,\,\langle s, [\,], \mathtt{True}\rangle\,,\,[\,]\,,\,\mathbf{D}\,,\,\sigma[s\mapsto \mathtt{I}\,(L,\mathbf{E})] \big\rangle \tag{7.12}$$

where $s = \mathtt{fresh}(\sigma)$

$$\big\langle \mathtt{S}\,,\,\langle r,\_,\_\rangle\,,\,[\,]\,,\,[\,]\,,\,\sigma \big\rangle \quad\rightsquigarrow\quad \big\langle \mathtt{vfs}\,(r)\,,\,\sigma \big\rangle \tag{7.13}$$

$$\big\langle \mathtt{D}\,(L,\mathbf{E})\,,\,\langle r,\_,\_\rangle\,,\,[\,]\,,\,[\,]\,,\,\sigma \big\rangle \quad\rightsquigarrow\quad \big\langle \mathtt{vfs}\,(s)\,,\,\sigma[s\mapsto \mathtt{I}\,(L,\mathbf{E})] \big\rangle \tag{7.14}$$

where $s = \mathtt{fresh}(\sigma)$

Figure 5.21: `vfsInsert` algorithm. Demotion transitions are given in figure 5.22.

$$\big\langle \texttt{fDem}\,(\langle l_d, p_d, r_d\rangle)\,,\, \langle r, \mathbf{G}, Q\rangle\,,\, \pi\,,\, \mathbf{D}\,,\, \sigma \big\rangle \quad \rightsquigarrow \tag{7.15}$$
$$\begin{aligned}
&\big\langle \texttt{fDem}\,(\langle l_d, p_d, r_d\rangle)\,,\, \langle s, \mathbf{G}', Q \wedge p\rangle\,,\, \langle r, \mathbf{G}, Q\rangle :: \pi\,,\, \mathbf{D}\,,\, \sigma \big\rangle \\
&\quad \text{if } \langle \_, \_, r\rangle \notin \mathbf{V} \wedge l_d < L \wedge |\mathbf{P}| = 1 \\
&\quad \text{where } \sigma(r) = \texttt{I}\,(L, \mathbf{V}) \\
&\quad \text{and } \mathbf{E} = \texttt{reinterpret}\,(\mathbf{V}, \mathbf{G}) \oplus \texttt{reinterpret}\,(\mathbf{G}, \mathbf{V}) \\
&\quad \text{and } \mathbf{P} = \texttt{filter}\,(p_d, \texttt{lev}_{=L}\,(\mathbf{E})) \\
&\quad \text{and } \langle \_, p, s\rangle = \mathbf{P}_1 \\
&\quad \text{and } \mathbf{G}' = \texttt{filter}\,(Q \wedge p, \texttt{lev}_{<L}\,(\mathbf{E}))
\end{aligned}$$

$$\big\langle \texttt{fDem}\,(\langle l_d, p_d, r_d\rangle)\,,\, \langle r, \mathbf{G}, Q\rangle\,,\, \pi\,,\, \mathbf{D}\,,\, \sigma \big\rangle \quad \rightsquigarrow \tag{7.16}$$
$$\begin{aligned}
&\big\langle \texttt{dem}\,(\langle \_, p'_d, r_d\rangle)\,,\, \langle s, \mathbf{G}', Q \wedge p\rangle\,,\, \langle r, \mathbf{G}, Q\rangle :: \pi\,,\, \mathbf{D}\,,\, \sigma[r \mapsto \texttt{I}\,(L, \texttt{del}\,(i, \mathbf{V}'))] \big\rangle \\
&\quad \text{if } l_d < L \wedge |\mathbf{P}| = 1 \\
&\quad \text{where } \sigma(r) = \texttt{I}\,(L, \mathbf{V}) \\
&\quad \text{and } \mathbf{V}' = \texttt{reinterpret}\,(\mathbf{V}, \mathbf{G}) \\
&\quad \text{and } i = \texttt{first}\,(\langle l, q, t\rangle, \mathbf{V}')\, t = r_d \\
&\quad \text{and } \langle \_, p'_d, r_d\rangle = \mathbf{V}'_i \\
&\quad \text{and } \mathbf{E} = \mathbf{V}' \oplus \texttt{reinterpret}\,(\mathbf{G}, \mathbf{V}) \\
&\quad \text{and } \mathbf{P} = \texttt{filter}\,(p'_d, \texttt{lev}_{=L}\,(\mathbf{E})) \\
&\quad \text{and } \langle \_, p, s\rangle = \mathbf{P}_1 \\
&\quad \text{and } \mathbf{G}' = \texttt{filter}\,(Q \wedge p, \texttt{lev}_{<L}\,(\mathbf{E}))
\end{aligned}$$

$$\big\langle \texttt{fDem}\,(\langle l_d, p_d, r_d\rangle)\,,\, \langle r, \mathbf{G}, Q\rangle\,,\, \pi\,,\, \mathbf{D}\,,\, \sigma \big\rangle \quad \rightsquigarrow \tag{7.17}$$
$$\big\langle \texttt{S}\,,\, \langle r, \mathbf{G}, Q\rangle\,,\, \pi\,,\, \mathbf{D}\,,\, \sigma \big\rangle$$

$$\big\langle \texttt{dem}\,(\langle l_d, p_d, r_d\rangle)\,,\, \langle r, \mathbf{G}, Q\rangle\,,\, \pi\,,\, \mathbf{D}\,,\, \sigma \big\rangle \quad \rightsquigarrow \tag{7.18}$$
$$\begin{aligned}
&\big\langle \texttt{dem}\,(\langle l_d, p_d, r_d\rangle)\,,\, \langle s, \mathbf{G}', Q \wedge p\rangle\,,\, \langle r, \mathbf{G}, Q\rangle :: \pi\,,\, \mathbf{D}\,,\, \sigma \big\rangle \\
&\quad \text{if } l_d < L \wedge |\mathbf{P}| = 1 \\
&\quad \text{where } \sigma(r) = \texttt{I}\,(L, \mathbf{V}) \\
&\quad \text{and } \mathbf{E} = \texttt{reinterpret}\,(\mathbf{V}, \mathbf{G}) \oplus \texttt{reinterpret}\,(\mathbf{G}, \mathbf{V}) \\
&\quad \text{and } \mathbf{P} = \texttt{filter}\,(p_d, \texttt{lev}_{=L}\,(\mathbf{E})) \\
&\quad \text{and } \langle \_, p, s\rangle = \mathbf{P}_1 \\
&\quad \text{and } \mathbf{G}' = \texttt{filter}\,(Q \wedge p, \texttt{lev}_{<L}\,(\mathbf{E}))
\end{aligned}$$

$$\big\langle \texttt{dem}\,(\langle l_d, p_d, r_d\rangle)\,,\, \langle r, \mathbf{G}, Q\rangle\,,\, \pi\,,\, \mathbf{D}\,,\, \sigma \big\rangle \quad \rightsquigarrow \tag{7.19}$$
$$\begin{aligned}
&\big\langle \texttt{S}\,,\, \langle r, \mathbf{G}, Q\rangle\,,\, \pi\,,\, \mathbf{D}\,,\, \sigma[r \mapsto \texttt{I}\,(L, \texttt{append}\,(\langle l_d, p_d, r_d\rangle, \mathbf{V}))] \big\rangle \\
&\quad \text{if } l_d < L \\
&\quad \text{where } \sigma(r) = \texttt{I}\,(L, \mathbf{V})
\end{aligned}$$

$$\big\langle \texttt{dem}\,(\langle l_d, p_d, r_d\rangle)\,,\, \langle r, \mathbf{G}, Q\rangle\,,\, \pi\,,\, \mathbf{D}\,,\, \sigma \big\rangle \quad \rightsquigarrow \tag{7.20}$$
$$\begin{aligned}
&\big\langle \texttt{S}\,,\, \langle r, \mathbf{G}, Q\rangle\,,\, \pi\,,\, \mathbf{D}\,,\, \sigma[r \mapsto \texttt{I}\,(L, \texttt{append}\,(\langle l_d, p_d, r_d\rangle, \mathbf{V}))] \big\rangle \\
&\quad \text{if } |\texttt{lev}_{=L}\,(\mathbf{V})| < \mathit{Max}_I \\
&\quad \text{where } \sigma(r) = \texttt{I}\,(L, \mathbf{V})
\end{aligned}$$

$$\big\langle \texttt{dem}\,(\langle l_d, p_d, r_d\rangle)\,,\, \langle r, \mathbf{G}, Q\rangle\,,\, \pi\,,\, \mathbf{D}\,,\, \sigma \big\rangle \quad \rightsquigarrow \tag{7.21}$$
$$\begin{aligned}
&\big\langle \texttt{D}\,(L+1, [\langle L+1, p_L, r\rangle, \langle L+1, p_R, s\rangle] \oplus \mathbf{E})\,,\, \langle r, \mathbf{G}, Q\rangle\,,\, \pi\,,\, \mathbf{D}\,,\, \sigma[r \mapsto \texttt{I}\,(L, \mathbf{V_L})\,, s \mapsto \texttt{I}\,(L, \mathbf{V_R})] \big\rangle \\
&\quad \text{where } \sigma(r) = \texttt{I}\,(L, \mathbf{V}) \\
&\quad \text{and } \mathbf{E} = \texttt{reinterpret}\,([\langle l_d, p_d, r_d\rangle], \mathbf{V} \oplus \mathbf{G}) \\
&\quad \text{and } \mathbf{V}' = \texttt{reinterpret}\,(\mathbf{V}, \mathbf{G} \oplus \mathbf{E}) \oplus \mathbf{E} \\
&\quad \text{and } \langle p_L, \mathbf{V_L}, p_R, \mathbf{V_R}, \mathbf{E}'\rangle = \texttt{splitI}\,(\mathbf{V}') \\
&\quad \text{and } s = \texttt{fresh}(\sigma)
\end{aligned}$$

Figure 5.22: Demotion transitions for the `vfsInsert` algorithm.

- the demotee has reached its natural level, and now causes the node to overflow.

In figure 5.23, we present part of an insertion trace. We adopt the same convention as before, denoting region descriptors with capital letters and the associated page identifiers using the same letter in lowercase. As execution proceeds, progressively more updates are made to the store, which for concision in the figure we do not repeat — in this figure only, we use the following convention for store notation. Lines in the figure are numbered, and we use the notation $\sigma_{\mathtt{n}}$ to indicate the store, including every update made to it, by the end of line `n`. For example, we see the first updates made in line 7:

$$\sigma[f \mapsto \mathtt{L}\,(\cdots)\,, f' \mapsto \mathtt{L}\,(\cdots)]$$

and in line 8 make further updates, subscripting the store symbol with the number 7. We mean

$$\sigma_7 = \sigma[f \mapsto \mathtt{L}\,(\cdots)\,, f' \mapsto \mathtt{L}\,(\cdots)]$$

and therefore

$$\begin{aligned} & \sigma_7[a \mapsto \mathtt{I}\left(0, \left[\mathrm{F}'_0, \mathrm{F}''_0, \mathrm{G}_0\right]\right) b \mapsto \mathtt{I}\,(0, [\mathrm{H}_0, \mathrm{J}_0])] \\ = \quad & \sigma[f \mapsto \mathtt{L}\,(\cdots)\,, f' \mapsto \mathtt{L}\,(\cdots)\,, a \mapsto \mathtt{I}\left(0, \left[\mathrm{F}'_0, \mathrm{F}''_0, \mathrm{G}_0\right]\right) b \mapsto \mathtt{I}\,(0, [\mathrm{H}_0, \mathrm{J}_0])] \end{aligned}$$

The figure describes the insertion of the point p into the VFS-tree of figure 5.15. We assume, as in figure 5.15, that the insertion causes both the child of entry F_0, and subsequently node M, to overflow. We further assume the split of node M to have the geometry of figure 5.18, such that both entries A'_1 and B_1 are scheduled for demotion.

Lines 2–6 are concerned with the insertion of p as far as the leaf level. The selected leaf node (at page f) overflows, posting entries F'_0 and F''_0 in line 7, and causing node M to overflow and post entries in line 8. (The exact partitionings selected in lines 7, 8 and 16 are not important and we omit the detail). These entries, A'_1 and B_1, are posted again as they are to replace the elevated entry A_1 in node N. Both are subsequently scheduled for demotion in line 10. Lines 12–17 handle the demotion of A'_1, and the demotion of B_1 commences in line 18.

5.5 Summary

We began this chapter by introducing `reduce`, an algorithm to transform a VFS-tree into the associated RVFS-tree that it represents. This allowed us to describe a suite of operations on VFS-trees, always using the `reduce` operation to provide the necessary link between an RVFS-tree and its VFS-tree representation. Placed into the VFS-framework, the correct approach to implementing BV-trees — the only published VFS-tree — becomes clear, and has enabled us to explain many features that are unclear or incorrect in its original presentation. The specification of algorithms using an abstract machine allowed us to provide clear examples of operations' execution using traces, essentially a sequence of operational states that can be mapped, without difficulty, back to the transitions that produced them.

The presentation in this chapter is rather abstract, in that we explicitly hide implementation details of possible VFS-trees, by integrating `interpret` into the store, and by our use of `splitL` and `splitI` for node splitting. This removes clutter from the presentation here, but the hidden details are necessary for concrete VFS-tree implementations. In chapter 6 we describe issues of

Line	
1	$\big\langle \texttt{vfsInsert}\,(\langle p, \_\rangle, q)\,,\ \sigma\big\rangle$
2	$\big\langle \texttt{ins}\,(\langle p, \_\rangle)\,,\ \langle q, [\,], \texttt{True}\rangle\,,\ [\,]\,,\ [\,]\,,\ \sigma\big\rangle$
3	$\big\langle \texttt{ins}\,(\langle p, \_\rangle)\,,\ \langle s, [\mathrm{W}_2], \mathrm{S}\rangle\,,\ [\langle q, [\,], \texttt{True}\rangle]\,,\ [\,]\,,\ \sigma\big\rangle$
4	$\big\langle \texttt{ins}\,(\langle p, \_\rangle)\,,\ \langle v, [\mathrm{A}_1], \mathrm{S}\wedge\mathrm{V}\rangle\,,\ [\langle s, [\mathrm{W}_2], \mathrm{S}\rangle, \langle q, [\,], \texttt{True}\rangle]\,,\ [\,]\,,\ \sigma\big\rangle$
5	$\big\langle \texttt{ins}\,(\langle p, \_\rangle)\,,\ \langle a, [\,], \mathrm{S}\wedge\mathrm{V}\wedge\mathrm{A}\rangle\,,\ [\langle v, [\mathrm{A}_1], \mathrm{S}\wedge\mathrm{V}\rangle, \langle s, [\mathrm{W}_2], \mathrm{S}\rangle, \langle q, [\,], \texttt{True}\rangle]\,,\ [\,]\,,\ \sigma\big\rangle$
6	$\big\langle \texttt{ins}\,(\langle p, \_\rangle)\,,\ \langle f, [\,], \mathrm{S}\wedge\mathrm{V}\wedge\mathrm{A}\wedge\mathrm{F}\rangle\,,\ [\langle a, [\,], \mathrm{S}\wedge\mathrm{V}\wedge\mathrm{A}\rangle, \langle v, [\mathrm{A}_1], \mathrm{S}\wedge\mathrm{V}\rangle, \langle s, [\mathrm{W}_2], \mathrm{S}\rangle, \langle q, [\,], \texttt{True}\rangle]\,,\ [\,]\,,\ \sigma\big\rangle$
7	$\big\langle \texttt{D}\,(0, [\mathrm{F}'_0, \mathrm{F}''_0])\,,\ \langle f, [\,], \mathrm{S}\wedge\mathrm{V}\wedge\mathrm{A}\wedge\mathrm{F}\rangle\,,\ [\langle a, [\,], \mathrm{S}\wedge\mathrm{V}\wedge\mathrm{A}\rangle, \langle v, [\mathrm{A}_1], \mathrm{S}\wedge\mathrm{V}\rangle, \langle s, [\mathrm{W}_2], \mathrm{S}\rangle, \langle q, [\,], \texttt{True}\rangle]\,,\ [\,]\,,\ \sigma[f \mapsto \texttt{L}\,(\cdots)\,, f' \mapsto \texttt{L}\,(\cdots)]\big\rangle$
8	$\big\langle \texttt{D}\,(1, [\mathrm{A}'_1, \mathrm{B}_1])\,,\ \langle a, [\,], \mathrm{S}\wedge\mathrm{V}\wedge\mathrm{A}\rangle\,,\ [\langle v, [\mathrm{A}_1], \mathrm{S}\wedge\mathrm{V}\rangle, \langle s, [\mathrm{W}_2], \mathrm{S}\rangle, \langle q, [\,], \texttt{True}\rangle]\,,\ [\,]\,,\ \sigma_7[a \mapsto \texttt{I}\,(0, [\mathrm{F}'_0, \mathrm{F}''_0, \mathrm{G}_0])\, b \mapsto \texttt{I}\,(0, [\mathrm{H}_0, \mathrm{J}_0])]\big\rangle$
9	$\big\langle \texttt{D}\,(1, [\mathrm{A}'_1, \mathrm{B}_1])\,,\ \langle v, [\mathrm{A}_1], \mathrm{S}\wedge\mathrm{V}\rangle\,,\ [\langle s, [\mathrm{W}_2], \mathrm{S}\rangle, \langle q, [\,], \texttt{True}\rangle]\,,\ [\,]\,,\ \sigma_8\big\rangle$
10	$\big\langle \texttt{S}\,,\ \langle s, [\mathrm{W}_2], \mathrm{S}\rangle\,,\ [\langle q, [\,], \texttt{True}\rangle]\,,\ [\mathrm{A}'_1, \mathrm{B}_1]\,,\ \sigma_8[s \mapsto \texttt{I}\,(2, [\mathrm{T}_2, \mathrm{U}_2, \mathrm{V}_2, \mathrm{W}_2, \mathrm{A}'_1, \mathrm{B}_1])]\big\rangle$
11	$\big\langle \texttt{S}\,,\ \langle q, [\,], \texttt{True}\rangle\,,\ [\,]\,,\ [\mathrm{A}'_1, \mathrm{B}_1]\,,\ \sigma_{10}\big\rangle$
12	$\big\langle \texttt{fDem}\,(\mathrm{A}'_1)\,,\ \langle q, [\,], \texttt{True}\rangle\,,\ [\,]\,,\ [\mathrm{B}_1]\,,\ \sigma_{10}\big\rangle$
13	$\big\langle \texttt{fDem}\,(\mathrm{A}'_1)\,,\ \langle s, [\mathrm{W}_2], \mathrm{S}\rangle\,,\ [\langle q, [\,], \texttt{True}\rangle]\,,\ [\mathrm{B}_1]\,,\ \sigma_{10}\big\rangle$
14	$\big\langle \texttt{dem}\,(\mathrm{A}'_1)\,,\ \langle v, [\,], \mathrm{S}\wedge\mathrm{V}\rangle\,,\ [\langle s, [\mathrm{W}_2], \mathrm{S}\rangle, \langle q, [\,], \texttt{True}\rangle]\,,\ [\mathrm{B}_1]\,,\ \sigma_{10}[s \mapsto \texttt{I}\,(2, [\mathrm{T}_2, \mathrm{U}_2, \mathrm{V}_2, \mathrm{W}_2, \mathrm{B}_1])]\big\rangle$
15	$\big\langle \texttt{D}\,(2, [\mathrm{V}'_2, \mathrm{V}''_2])\,,\ \langle v, [\,], \mathrm{S}\wedge\mathrm{V}\rangle\,,\ [\langle s, [\mathrm{W}_2], \mathrm{S}\rangle, \langle q, [\,], \texttt{True}\rangle]\,,\ [\mathrm{B}_1]\,,\ \sigma_{14}[v \mapsto \texttt{I}\,(1, [\cdots, \mathrm{A}'_1])\,, v' \mapsto \texttt{I}\,(1, [\cdots])]\big\rangle$
16	$\big\langle \texttt{D}\,(3, [\mathrm{S}'_3, \mathrm{S}''_3])\,,\ \langle s, [\mathrm{W}_2], \mathrm{S}\rangle\,,\ [\langle q, [\,], \texttt{True}\rangle]\,,\ [\mathrm{B}_1]\,,\ \sigma_{15}[s \mapsto \texttt{I}\,(2, [\mathrm{V}'_2], [\mathrm{V}''_2])\,, s' \mapsto \texttt{I}\,(2, [\mathrm{T}_2, \mathrm{U}_2, \mathrm{X}_2])]\big\rangle$
17	$\big\langle \texttt{S}\,,\ \langle q, [\,], \texttt{True}\rangle\,,\ [\,]\,,\ [\mathrm{B}_1]\,,\ \sigma_{16}[q \mapsto \texttt{I}\,(3, [\mathrm{S}'_3, \mathrm{S}''_3, \mathrm{R}_3, \mathrm{W}_2])]\big\rangle$
18	$\big\langle \texttt{fDem}\,(\mathrm{B}_1)\,,\ \langle q, [\,], \texttt{True}\rangle\,,\ [\,]\,,\ [\,]\,,\ \sigma_{17}\big\rangle$
19	$\cdots$

Figure 5.23: Partial trace of insertion of point p into the VFS-tree of figure 5.15.

predicate interpretation and split policies for implementations of the BV-tree and KDB-VFS tree. Our BV-tree implementation is, we believe, the first working example of the structure.

Chapter 6

Implementation

In chapter 5 we presented algorithms for a number of core operations on a class of virtual forced splitting (VFS) trees. This presentation was made at a level above the usual approach to access method implementation, in that we handle nodes as a collection of fully-described predicates, irrespective of their description decoded from disk. In this chapter, we close the gap between this higher-level description and practical implementations of the BV-tree and KDB-VFS tree, by providing the structure-specific details of features abstracted out in algorithms presented in chapter 5.

6.1 Implementation framework

All structures were implemented in the Java programming language; both the VFS-structures under investigation and the R- and R*-trees against which their performance is compared in chapter 7. The implementation framework has three layers:

- database interface;
- generic tree architecture;
- specific tree implementation.

The database interface layer manages tree files on disk and provides an interface to those files for higher-level code through a buffer pool. Higher layers do not access files directly, but request pages from the buffer pool and write them back to the buffer pool. This is intended to provide a simple, DBMS-like interface for tree development.

The tree architecture layer specifies a standard interface for a tree structure, including the specification of common operations required of all implementations, for example `insert` or `rQuery`. It also includes a standard representation for leaf and internal nodes and their entries, and implementations of core functions, for example, given a buffer frame, to read a page of internal node entries. This has a number of advantages:

- reuse of core code provides consistency between tree implementations' primitives;
- tree implementation layer code can be pared down to structure-specific routines specified in higher-level terms like 'read a page of entries' or 'initialise a new page';

- a common interface for all trees allows common instrumentation for performance evaluation and debugging. For example, all trees built on two-dimensional datasets can be represented graphically using shared tree-viewing software that interfaces with structures at the tree architecture layer; figure 6.1 provides screenshots of small BV- and R*-tree examples.

The tree implementation layer contains structure-specific code; essentially the implementation of specified operations with any required subordinate functions (for example, split policies). The degree of abstraction offered by the architecture layer is chosen to permit relatively clean structure-specific code, without tying a tree to a particular algorithmic paradigm. Within the framework, tree operations can equally well be implemented recursively, iteratively, or using a direct abstract-machine style approach such as that used to describe tree operations in earlier chapters.

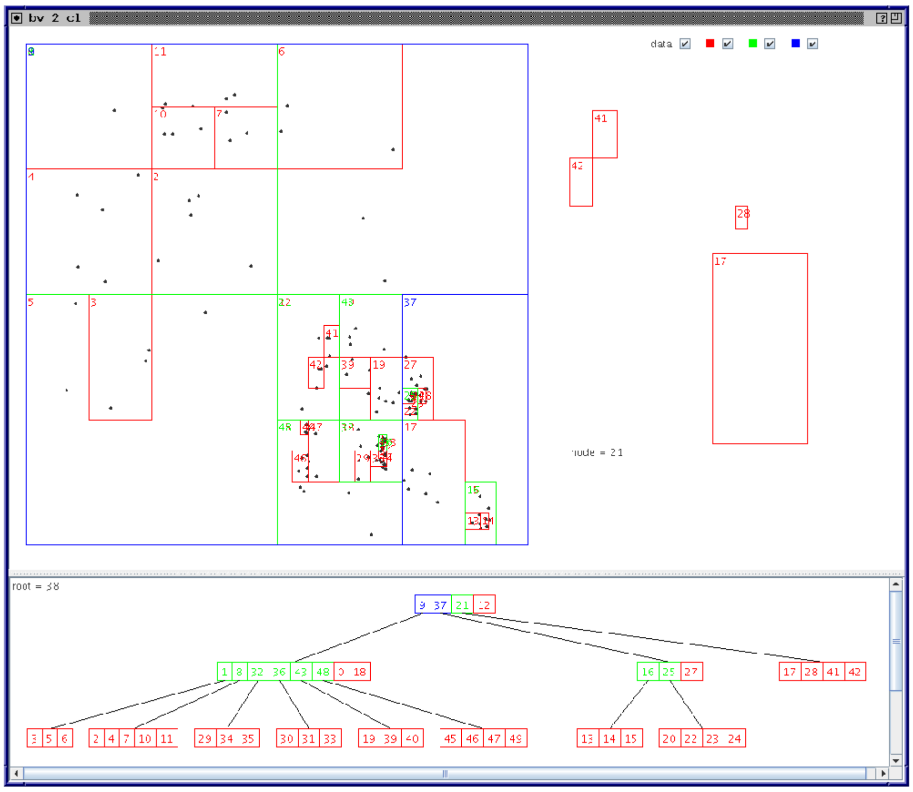
Figure 6.2 gives a schematic view of the three layers atop the filesystem. An example call to read a node is shown. Tree-specific code in the implementation layer calls for a node at page address `pageId`, which is passed by the tree architecture layer to the database interface as a call for a page. The database interface, acting as a buffer pool, retrieves, from file, the bytes that form that page and returns a reference to a buffer page frame. The node's entries are then read from the page and returned to the calling tree routine. This is a slight simplification of what happens in practice — in fact the page frame is returned to the top layer, which retrieves the node type (leaf or internal) from the page directly. However, it then calls an architecture layer routine to extract the entries from the leaf or internal accordingly, so this is a reasonable view of the system.

Because this implementation was written in Java, we are unable to pass predicates around as first-class objects in the way in which we do so using the abstract machine language. Instead, the objects treated as 'predicates' are actually the underlying region representations; where in previous chapters we might have handled $\lambda x.(x \in R)$ as a node predicate, here we simply use R. Similarly, invocations of `first` or `uniq` are inlined, rather than called with a predicate parameter. The internal node returned by the tree architecture layer in figure 6.2 therefore has the structure:

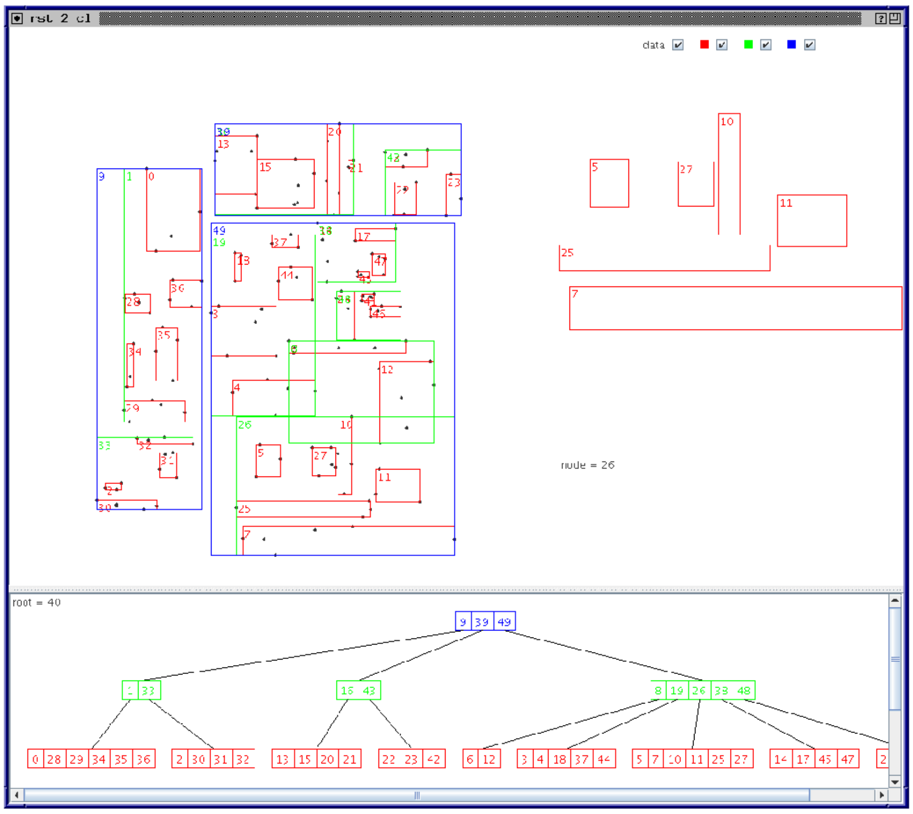
$$\mathtt{I}\,(L : \mathtt{int}, [\langle l_1 : \mathtt{int}, R_1 : \mathtt{Rectangle}, s_1 : \mathtt{int}\rangle, \ldots])$$

The predicate component of an entry in this structure is an explicit rectangle, corresponding to the decoded description of the on-disk representation for all of the structures examined here. Node entries, described above as ternary tuples, are implemented as instances of the `NodeEntry` class; the class's field definitions appear in figure 6.3. Instances of `NodeEntry` are direct translations of their representation on disk.

The rectangle predicate component of a `NodeEntry` is encoded as $2.d$ floating point numbers (where d is the dimensionality of the space), wrapped in a `Rectangle` object. Because local predicates in the R-tree, R*-tree and KDB-VFS tree are of the form $\lambda x.(x \in R)$, where R is an explicit description of a rectangle, they do not require further interpretation; these structures can therefore treat `NodeEntry` objects directly as interpreted entries. The BV-tree offers an example in which direct translation of an on-disk representation into a list of `NodeEntry` objects is insufficient for the interpretation of local predicates; we shall consider two approaches to predicate interpretation in the BV-tree in sections 6.3 and 6.4.



(a) BV-tree



(b) R*-tree

Figure 6.1: Tree viewer used to visualise structures in two dimensions.

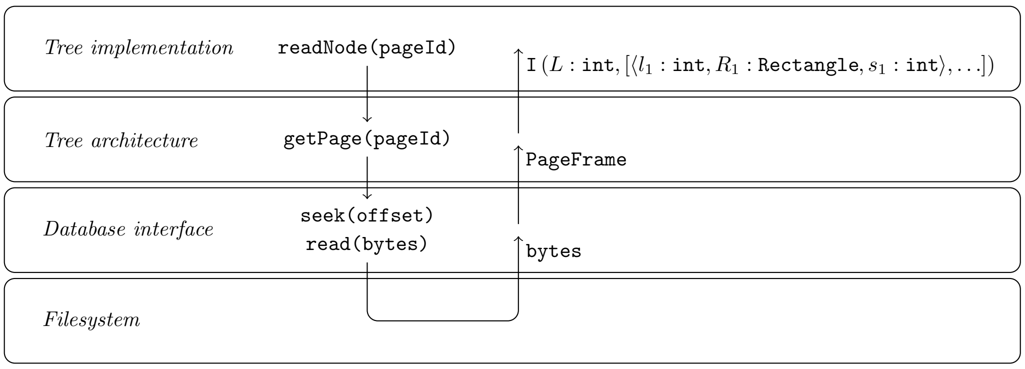


Figure 6.2: Implementation framework schematic.

```
public class NodeEntry implements Comparable<NodeEntry> {

  private int level, child;
  private Rectangle predicate;
  ...
}
```

Figure 6.3: `NodeEntry` field definitions.

6.2 BV-tree implementations

The BV-tree was introduced by Freeston as an 'abstract' access method, in that it supports any region representation in which the containment property is guaranteed, *i.e.* given two region boundaries A and B, either A contains B, B contains A, or A and B are disjoint. This requirement is, however, rather special. Provision of the containment property means that, when an overflowing node is to split, a boundary must be decided that not only meets the local requirements of the split, but that also preserves the containment property. The only information available to support this choice is that present *locally* (in the splitting node), but the chosen split boundary must still be guaranteed to contain, be contained by, or be disjoint from every other boundary in this level of decomposition throughout the entire tree, *i.e. globally.* This clearly requires split boundaries to be absolutely predictable.

The BANG file's approach to splitting nodes is the only one of which we are aware that provides the containment property. Node splits are binary and symmetric, in that each binary split takes place at the midpoint of an interval in the splitting dimension, which is chosen in strict rotation. An effect of this is that the space must be bounded to enable selection of appropriate midpoints, providing the global predictability of split boundary positions. The approach was described briefly in section 3.3.5 and we provide an algorithm below in the presentation of our BV-tree splitting policy.

Recall also that the BANG file does not represent holey regions explicitly on disk, but that these must be interpreted from the collection of rectangular regions associated with a set of entries — instances of the `NodeEntry` class described in section 6.1 are therefore suitable for representing the BV-tree entries decoded from disk, but not the interpreted entries used in VFS-tree operations. We describe the implementation of BV-tree interpreted predicates in sections 6.3 and 6.4. This means that, although rectangles are encoded in [22] as bitstrings, we encode them as floating-

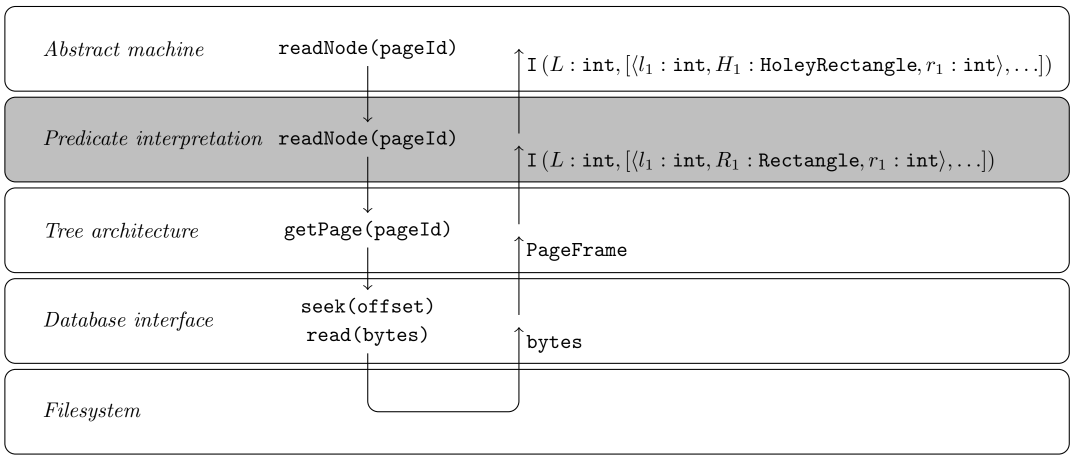


Figure 6.4: Implementation framework schematic with predicate interpretation integrated into the store.

point numbers. This does not affect the characteristics of the structure other than to require more physical space to represent a BV-region on a disk page, and allows us to take advantage of the shared framework described in section 6.1. We consider the effect of the increased disk-space requirement for this representation in section 8.1.1.

We described the potential of our implementation framework to permit development of structures using different algorithmic approaches, and have been able to demonstrate this with two BV-tree implementations, one using a hand-coded translation of our abstract machine rules, and another in a 'traditional' recursive style, based on our understanding of the structure gained using the abstract machine approach. We also demonstrated experimentally the implementations' equivalence — a series of performance results such as those presented in chapter 7 were obtained from both implementations and found to be identical.

6.3 BV-tree: Abstract machine implementation

Our abstract machine implementation is a hand-coded translation of the VFS-tree algorithms provided in chapter 5. These use an explicitly interpreted node structure, abstracting away from predicate interpretation by integrating the `interpret` function into the store. Provision of this in our abstract machine implementation can be thought of as another layer of abstraction, immediately above the tree architecture layer, and shown shaded in figure 6.4. The node structure returned to the predicate interpretation layer uses rectangle-based `NodeEntry` objects as described in section 6.1; the predicate interpretation layer then takes this direct decoding of the on-disk representation and uses it to interpret the BV-tree's holey regions.

6.3.1 Predicate interpretation

Predicate interpretation takes place from a list of `NodeEntry` instances and returns a list of `HoleyEntry` objects with an analogous structure; the `HoleyEntry` class's field definitions are shown in figure 6.5. The predicate component of a `HoleyEntry` is a `HoleyRectangle`; these are implemented as an outer `Rectangle` and a list of holes, as shown in figure 6.6.

Interpretation of a `HoleyEntry` from a list of `NodeEntry` objects has two parts: first, the list

```
public class HoleyEntry {

  private int level, child;
  private HoleyRectangle predicate;
  ...
}
```

Figure 6.5: `HoleyEntry` field definitions.

```
public class HoleyRectangle {

  private Rectangle outer;
  private ArrayList<Rectangle> holes;
  ...
}
```

Figure 6.6: `HoleyRectangle` field definitions.

```
public ArrayList<HoleyEntry> interpret(ArrayList<NodeEntry> entries) {
  ArrayList<HoleyEntry> output = new ArrayList<HoleyEntry>();
  for(NodeEntry e:entries) {
    HoleyRectangle hr = new HoleyRectangle(e.rect());
    for(NodeEntry h:entries)
      if (h!=e && h.level() == e.level() && e.rect().contains(h.rect()))
        hr.addHole(h.rect())
    hr.consolidateHoles();
    output.add(new HoleyEntry(e.level(), hr, e.child());
  }
  return output;
}
```

Figure 6.7: `interpret` method for `HoleyRectangle` interpretation. This forms part of the predicate interpretation layer.

of rectangles from entries of the same level and contained by the outer rectangle is assembled as a list of potential holes, then the list of holes is *consolidated* — holes within holes are removed. The code for the `interpret` and `consolidateHoles` methods is given in figures 6.7 and 6.8.

Figure 6.9a shows a collection of regions, which we assume to be drawn from entries of the same level number. Taking these entries as input to the `interpret` method of figure 6.7, the selection of outer region W, the addition of holes in `interpret` and their subsequent removal by `consolidateHoles` is shown in figures 6.9b, 6.9c and 6.9d respectively.

6.3.2 Containment and Intersection

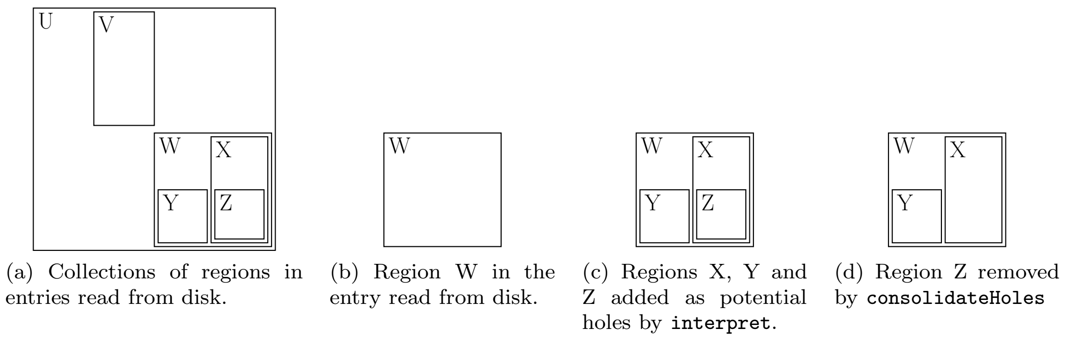
The BV-tree insertion algorithm requires us to be able to test for:

- satisfaction of a predicate by a point, *i.e.* containment of a point in a region, to support exact-match searching and insertion;
- implication of the satisfaction of one predicate by that of another, *i.e.* containment of one region in another, to support demotion.

The code for the point containment test is given in figure 6.10. Simply, if a point is contained in the outer region, and in none of the interpreted region's holes, then it is contained in the interpreted region.

```
public void consolidateHoles() {
  for(int i = holes.size()-1; i >=0; i--) {
    Rectangle r = holes.get(i);
    for(Rectangle h:holes) {
      if (r!=h && h.contains(r)) {
        holes.remove(i);
        break;
      }
    }
  }
}
```

Figure 6.8: The `HoleyRectangle` class's `consolidateHoles` method.



(a) Collections of regions in entries read from disk.

(b) Region W in the entry read from disk.

(c) Regions X, Y and Z added as potential holes by `interpret`.

(d) Region Z removed by `consolidateHoles`

Figure 6.9: Interpretation of holey region W \ (X ∪ Y).

```
public boolean contains(Point p) {
  if (!(outer.contains(p))
    return false;
  for(Rectangle h:holes)
    if (h.contains(p))
      return false;
  return true;
}
```

Figure 6.10: The `HoleyRectangle` class's `Point` containment test.

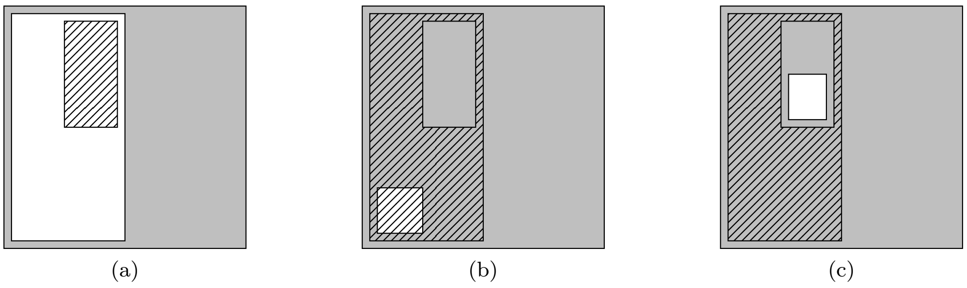


Figure 6.11: Testing for containment between instances of `HoleyRectangle`.

```
public boolean contains(HoleyRectangle r) {
  if (!(outer.contains(r.outer))
    return false;
  for(Rectangle h:holes) {
    boolean holeContained = false;
    for(Rectangle rh:r.holes)
      if (rh.contains(h))
        holeContained = true;
    if (!holeContained)
      return false;
  }
  return true;
}
```

Figure 6.12: The `HoleyRectangle` class's `HoleyRectangle` containment test.

Testing region containment for demotion is more complicated, because it must consider not only the relationship between the outer boundaries of the two regions under consideration, but also that between their respective holes. For `HoleyRectangle A` to contain `HoleyRectangle B`, we require that:

- `A.outer` contains `B.outer`;
- `B.outer` does not fall into any of `A`'s holes;
- if any of `A`'s holes fall inside `B.outer`, that they also fall inside one of `B`'s holes.

We illustrate some possible cases in figure 6.11. In each, the space contained in `HoleyRectangle A` is shaded and that in `B` is hatched. In figure 6.11a, `B.outer` falls into a hole in `A`, so `A` does not contain `B`. In figure 6.11b, `B.outer` is not in any hole of `A`'s, but contains a hole in `A`. This hole does not fall inside any of `B`'s holes; once again, `B` is not contained in `A`. In figure 6.11c, although `B.outer` contains a hole in `A`, that hole falls within one of `B`'s own holes — in this case, `A` contains `B`. The code for region containment testing in the `HoleyRectangle` class is given in figure 6.12.

Testing for intersection between predicates is required for region searches. In the case of predicates respecting the containment property, no 'partial' intersection is possible — given a predicate, P, and any other predicate in the tree, Q, P either contains Q, is contained by Q, or is disjoint from Q. Arbitrary query predicates, however, may intersect a tree region's outer boundary without either containing it or being contained by it. To test for intersection between an arbitrary query predicate region, `Q` and a `HoleyRectangle`, `A`, we instead compose the intersection between `Q` and `A.outer`, then test the composed rectangle for containment in `A`; if the region of intersection is contained in one of `A`'s holes, the test for intersection will fail. Hyperspherical queries are not, however, tested in this way; instead, as in other access methods like the R*-tree, we use the minimum distance between the query region and a region's outer boundary to decide whether or not the region should be explored.

6.3.3 Splitting policy

The term 'splitting policy' is used to describe an heuristic approach, at runtime, to partitioning a splitting node's entries. The R*-tree, as described in section 3.2.3, generates a number of possible partitionings of the node's entries, before deciding on the 'best' split based on a measure of fitness. This is common, but not unique, to SPP structures, as we shall see in section 6.5.1. The BANG file's splitting policy is unusual in that, instead of generating a set of different partitionings and evaluating each, it generates partitionings in a strict sequence, stopping when a 'good' partitioning — in this case, that with best occupancy balance — is found. Informally, this proceeds as follows:

1. Generate the first partitioning by dividing the region into two along the first splitting dimension, selecting, as a hole, the most heavily occupied half of the region. Choice of splitting dimension is ordered; the first splitting dimension is calculated by examining each side of the splitting region, in order, to find the first side that is longer than the rest. If the region is found to be hypercubic, the splitting order cycles round to the first dimension.

   The hole formed is a new region in its own right, and also implies a holey region formed by the remainder. Calculate the split ratio for the partitioning by dividing the number of primary entries contained inside the hole by the number contained in the region outside the hole.

2. Generate the next partitioning by dividing the hole into two along the next dimension in the splitting order, and select, as a hole, the most heavily occupied half. Calculate the split ratio in the same way.

3. Generate successive partitionings as in step 2, until the partitioning's split ratio falls below 1. At this point, the most recent partitioning has more entries outside the hole than in, while the reverse is true of the previous partitioning. One of these two has best balance — this is the partitioning returned by the split policy. If the partitioning with best balance is the first partitioning generated, this can be described by a buddy (*i.e.* a non-holey) region split.

4. Allocate all remaining interpreted entries in the node as follows:

   - If the entry is contained in the hole, add it to the set of primary entries inside the hole;
   - if the entry is contained in the outer, holey region, add it to the set of primary entries in the holey region;
   - if the entry is contained in neither region, it must be elevated.

   The set of entries that require elevation contains at most one entry from each level, including the level of the splitting node; *i.e.* a primary entry.

Figure 6.13 presents the portion of the `splitI` (internal node split) method responsible for generating and selecting partitions from a list of entries in an overflowing node. The dimensionality of the space is specified by the constant `DIM`, and the first splitting dimension identified by the method `getFirstSplitDir()` (the implementation of which is not shown). The `while` loop generates successive partitions until `bal < 1`, after which the best partitioning of the last two generated is selected. Finally, if this is to be a buddy split, the `outer` region is adjusted to be the buddy of the `hole`.

Figure 6.14 shows stages in execution of the algorithm; only primary entries are shown. Notice that in figure 6.14b the split balance is calculated from the partitioning of seven entries, while in the other stages it is calculated from six. This is because figure 6.14b is the only partitioning in which no entry would require elevation.

6.3.4 Implementing the abstract machine

The presentation of algorithms as a collection of transitions between abstract machine states implies a certain amount of machinery, required to execute those transitions and to select the appropriate sequence of transitions for the execution of the overall operation. In order to provide a direct implementation of the abstract machine descriptions, we need to provide that machinery. In our Java implementation, operations are specified as methods that take as arguments the operation's input command parameters, and return the termination command parameters in a single object.

The general structure of an operation as a Java method is given in figure 6.15. Each transition is enclosed in a `while(true)` loop, allowing us to test whether the transition is triggered and `break` out of the loop if the test fails. If the test is passed, actions required to execute the transition are taken and the output configuration prepared. The loop then exits with a `continue` statement that returns control to the top of another `while(true)` loop, enclosing all transitions — at this point the output configuration parameters, formed during the previous iteration of the outer loop, become the input configuration parameters for the next iteration.

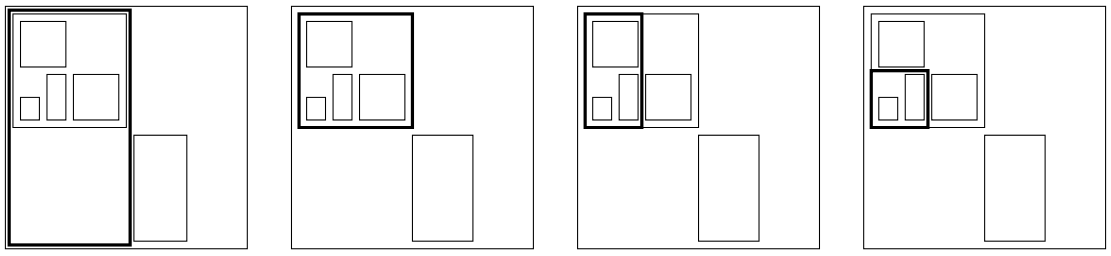
As an example, figure 6.16 gives a code fragment from the start of the Java abstract machine implementation of `vfsInsert`. This includes the initialisation of operational parameters and the implementation of transition 7.2 from figure 5.21 (repeated here in figure 6.17).

This is a very direct style of translation from the rules; notice that, at the start of transition 7.2 in figure 6.16, the required command type (`Ins`) is tested and a node read from the store (`Node node = store.get(r)`). This transition, however, may not be selected for execution — if the node is not a leaf, this loop will exit and testing for the next transition will begin. In practice, an implementation of every rule in this fashion would mean that execution of a single transition might result in several calls to `store.get(r)`. Although we sought an explicit stepwise description in the abstract machine for algorithmic clarity, a direct translation into real program code is clearly not efficient.

In section 2.5.2, we described the ordering of transitions to avoid having to negate conditions explicitly in subsequent rules. Note, however, that the input configuration also includes at least one test — a pattern match of the input command — and may include others such as the requirement for a non-empty stack of reduction tuples. To optimise the abstract machine implementation, therefore, rules should be ordered to allow transitions of the same input command to be grouped together, and, within a group, to make a single read from the store where required. We demonstrated a functional approach to this in the OCaml implementation of the B+ tree in figure 2.10; in the Java implementation it can be handled with an additional `while(true)` grouping around each rule subset. Figure 6.18 illustrates a possible grouping for the `vfsInsert` transitions with a `D` input configuration; note that transitions 7.12 and 7.14 appear first, as these are promoting out of the

```
public NodeSplit splitI(ArrayList<HoleyEntry> entries,
      Rectangle exterior, int nodeLevel) {
    int firstSplitDir = getFirstSplitDir(exterior);
    int splitDir = firstSplitDir;

    Rectangle hole = exterior;
    Rectangle prevHole = null;
    float bal = entries.size(); // worst possible!
    float prevBal = 2; // worse than if we get perfect first time round

    while (bal >= 1) {

      prevBal = bal;
      prevHole = hole;

      // find most heavily occupied half
      Rectangle[] halves = hole.bisect(splitDir);
      int ct0 = 0;
      int ct1 = 0;
      for (HoleyEntry e :  entries) {
        if (e.level() != nodeLevel)
          continue;
        if (halves[0].contains(e.predicate().outer()))
          ct0++;
        else if (halves[1].contains(e.predicate().outer()))
          ct1++;
      }

      // set up the new regions
      if (ct0 > ct1)
        hole = halves[0];
      else
        hole = halves[1];
      HoleyRectangle outer = new HoleyRectangle(exterior);
      outer.addHole(hole);

      // calculate balance
      int in = 0;
      int out = 0;
      for (HoleyEntry e :  entries) {
        if (e.level() != nodeLevel)
          continue;
        if (hole.contains(e.predicate().outer()))
          in++;
        else if (outer.contains(e.predicate().outer()))
          out++;
      }
      bal = (float) in / out;
      splitDir = (splitDir + 1) % DIM;
    }

    // decide best
    if (1 / bal > prevBal)
      hole = prevHole;
    HoleyRectangle outer = new HoleyRectangle(exterior);
    outer.addHole(hole);

    // buddy split?
    Rectangle[] halves = exterior.bisect(firstSplitDir);
    if (hole.equals(halves[0]))
      outer = new HoleyRectangle(halves[1]);
    else if (hole.equals(halves[1]))
      outer = new HoleyRectangle(halves[0]);

  ...
```

Figure 6.13: Implementation of `splitI` for a BANG file style region decomposition using explicitly interpreted predicates.



(a) $5/1 = 5.0$ (b) $5/2 = 2.5$ (c) $3/3 = 1.0$ (d) $4/2 = 0.5$

Figure 6.14: Split balances at stages in the execution of `splitI`. Partitioning (c) is selected.

```
public ResultObj operation(⟨InputCmdParams⟩) {
  OpParam1Type nextOpParam1 = ... ;
  OpParam2Type nextOpParam2 = ... ;
  OpParam3Type nextOpParam3 = ... ;
  ...

  outerloop:
  while(true) {
    OpParam1Type opParam1 = nextOpParam1 ;
    OpParam2Type opParam2 = nextOpParam2 ;
    OpParam3Type opParam3 = nextOpParam3 ;
    ...

    // transition 1
    while(true) {
      if(!(⟨transitionCondition⟩))
        break;
      nextOpParam1 = ⟨outputConfigParam1⟩ ;
      nextOpParam2 = ⟨outputConfigParam2⟩ ;
      nextOpParam3 = ⟨outputConfigParam3⟩ ;
      ...
      continue outerloop ;
    }

    // transition 2
    while(true) {
      if(!(⟨transitionCondition⟩))
        break;
      nextOpParam1 = ⟨outputConfigParam1⟩ ;
      nextOpParam2 = ⟨outputConfigParam2⟩ ;
      nextOpParam3 = ⟨outputConfigParam3⟩ ;
      ...
      continue outerloop ;
    }

    ...

    // terminating transition
    while(true) {
      if(!(⟨transitionCondition⟩))
        break;
      return new ResultObj(⟨resultParams⟩);
    }
  }
}
```

Figure 6.15: Structure of an operation in the Java implementation of the abstract machine.

```java
public int vfsInsert(LeafEntry le, int root, Store sigma) {
  // transition 7.1 - initialise
  Command nextCmd = new Ins(le);
  ReductionTuple nextFt = new ReductionTuple(root,
    new ArrayList<HoleyEntry>(), HoleyRectangle.wholeSpace());
  Stack nextStk = new Stack();
  DemoteQueue nextDq = new DemoteQueue();
  Store nextStore = sigma;

  outerloop:  while (true) {

    // rotate configuration parameters
    Command cmd = nextCmd;
    ReductionTuple ft = nextFt;
    Stack stk = nextStk;
    DemoteQueue dq = nextDq;
    Store store = nextStore;

    // transition 7.2 - insert into next level
    while (true) {
      if (!(cmd instanceof Ins))
        break;
      LeafEntry kv = ((Ins) cmd).entry();

      int r = ft.pid();
      Node node = store.get(r);
      if (!node.isInternalNode())
        break;
      INode iNode = (INode) node;
      int bigL = iNode.level();

      ArrayList<HoleyEntry> bigV = iNode.entries();
      ArrayList<HoleyEntry> bigG = ft.pending();
      bigG = fs.reinterpret(bigG, bigV);
      bigV = fs.reinterpret(bigV, bigG);

      Key k = kv.key();
      HoleyEntry _ps = null;
      HoleyRectangle Q = ft.predicate();

      for (HoleyEntry Lqt :  bigG.concat(bigV)) {
        if (Lqt.level() == bigL && Lqt.predicate().contains(k)) {
          _ps = Lqt;
          break;
        }
      }
      HoleyRectangle p = _ps.predicate();
      int s = _ps.child();

      HoleyRectangle qAndP = Q.compose(p);
      ArrayList<HoleyEntry> bigGPrime = new new ArrayList<HoleyEntry>(), ();
      for (HoleyEntry lqt :  bigG.concat(bigV)) {
        if (lqt.level() < bigL && qAndP.intersects(lqt.predicate())) {
          bigGPrime.add(lqt);
        }
      }

      nextCmd = cmd;
      nextFt = new ReductionTuple(s, bigGPrime, qAndP);
      nextStk = stk.push(ft);
      nextDq = dq;
      nextStore = store;

      continue outerloop;
    }
    ...
```

Figure 6.16: Fragment of the Java abstract machine implementation of `vfsInsert`.

$$\left\langle \texttt{ins}\,(\langle k,v\rangle)\,,\,\langle r,\mathbf{G},Q\rangle\,,\,\pi\,,\,\mathbf{D}\,,\,\sigma\right\rangle \quad\rightsquigarrow\quad \left\langle \texttt{ins}\,(\langle k,v\rangle)\,,\,\langle s,\mathbf{G}',Q\wedge p\rangle\,,\,\langle r,\mathbf{G},Q\rangle::\pi\,,\,\mathbf{D}\,,\,\sigma\right\rangle \tag{7.2}$$
$$\text{if } \sigma(r) = \texttt{I}\,(L,\mathbf{V})$$
$$\text{where } \mathbf{E} = \texttt{reinterpret}\,(\mathbf{V},\mathbf{G}) \oplus \texttt{reinterpret}\,(\mathbf{G},\mathbf{V})$$
$$\text{and } \langle \_,p,s\rangle = \texttt{uniq}\,(\texttt{lev}_{=L}\,(\mathbf{E}),P) \text{ and } P = \lambda x.\text{match } x \text{ as } \langle l,q,t\rangle \text{ in } q(k)$$
$$\text{and } \mathbf{G}' = \texttt{filter}\,(Q\wedge p,\texttt{lev}_{<L}\,(\mathbf{E}))$$

Figure 6.17: Transition 7.2 of the `vfsInsert` algorithm, given in full in figure 5.21.

splitting VFS-root and do not require an existing page to be read from the store.

6.4 BV-tree: Recursive implementation

Our recursive BV-tree implementation is a more 'traditional' index structure implementation, in the sense that it takes more familiar approaches to algorithm implementation and tree entry handling. Tree operations are implemented using recursive Java methods, and `NodeEntry` objects are handled directly, as returned from the tree architecture layer, effectively interpreting predicates on-the-fly. This has consequences for our approaches to predicate interpretation and node splitting, which we discuss here.

6.4.1 Predicate interpretation

Our approach to predicate interpretation in the abstract machine implementation is to interpret explicitly the predicate associated with each subtree in a node, and then to examine each predicate with respect to the property we seek, *e.g.* containment of a point, or intersection with a region. When handling decoded, uninterpreted `Rectangle` regions, associated with `NodeEntry` objects, it becomes necessary to consider them collectively, in a way that permits testing of the required property without explicitly constructing the interpreted regions.

Consider the case of point containment. Given two rectangles, A and B, both may contain a point p, but the containment property then implies that either A contains B, or B contains A. In either case, the inner rectangle represents a hole in the outer, so the point is contained solely in the inner, interpreted region. In general, given a set of rectangles possessing the containment property, the rectangle bounding the interpreted region that contains a point p is the rectangle, drawn from the set, that 'most closely encloses p'. This is the rectangle that contains p, and contains no other rectangle that contains p. To find this rectangle, without explicitly interpreting predicates, we impose a partial ordering on the set of rectangles — that implied by containment. Given two rectangles, A and B:

$$\mathrm{A} \subseteq \mathrm{B} \leftrightarrow \mathrm{A} \leqslant \mathrm{B}$$

Given a list of rectangles ordered in this way, the first rectangle in the list that contains p is also the rectangle that most closely encloses it.

In practice, we do not handle lists of rectangles, but lists of `NodeEntry` objects. Suppose that, during the insertion of a point p, we sort a list of primary entries into the containment order of their component rectangles. If we then search the list, and identify the rectangle that most closely encloses p, we have also identified the entry into which p is to be inserted. In the abstract machine implementation, we separated predicate interpretation from entry selection, but in the recursive

```
...
outerloop:  while (true) {

  // rotate configuration parameters
  Command cmd = nextCmd;
  ReductionTuple ft = nextFt;
  Stack stk = nextStk;
  DemoteQueue dq = nextDq;
  Store store = nextStore;

  ...

  dLoop:
  while(true) {
    if (!(cmd instanceof D))
      break;

    while(true) {
      if(!stk.isEmpty())
        break;
      if(!(dq.isEmpty())) {
        ... // execute transition 7.12
        continue outerloop;
      } else {
        ... // execute transition 7.14
        return newRoot;
      }
    }

    int pid = stk().peek().pid();
    INode node = (INode)store.get(pid);

    // transition 7.6
    while(true) {
      ...
      continue outerloop;
    }

    // transition 7.7
    while(true) {
      ...
      continue outerloop;
    }
    ...
```

Figure 6.18: Grouping of `D` input configurations in `vfsInsert` to optimise store reads.

implementation the two tasks are merged into one. We must therefore discuss the interpretation of predicates in the context of the operation in which they are interpreted.

In the insertion of the point, p, having identified, as above, the subtree into which p is to be inserted, we must now identify the pending set required to accompany the descent of that subtree. Recall from our discussion in section 5.3.1 that this must include every entry whose interpreted predicate intersects that subtree's predicate. It is therefore not possible to construct the pending set during our first pass of the list, because at that point the subtree itself is unknown. Having identified the subtree, however, the only information immediately available is its region's outer boundary — the rectangle in the selected entry. This leaves us with two options:

- explicitly interpret the region associated with the selected subtree, and identify other entries whose interpreted regions intersect it, as in the abstract machine approach, or;
- search the list in order, as before, for every entry of any level whose rectangle is contained by the outer boundary of the selected subtree, and the first entry of each level whose rectangle contains that boundary.

We take the second approach here, because it is consistent with the spirit of the approach taken to identify the subtree for descent. Neither approach is incorrect, but note that the second can result in the inclusion of entries that are not actually required — if the descent subtree is actually holey, we may inadvertently include entries that fall into one of its holes.

Figure 6.19 gives an extract of the recursive BV-tree insertion code. Note that, before selecting entries, we merge in the incoming pending set, but that there is no requirement for explicit reinterpretation, because no interpretation is performed at all until we begin to search the sorted list. The merged list of entries is sorted with a call to `Collections.sort(allEntries)`; it is for this reason that the `NodeEntry` class implements the `Comparable` interface. Its implementation is given in figure 6.20, and enables sorting of the list into containment order within entry level number, allowing the code in figure 6.19 to search the list one level at a time. The variable `level` indicates the level number of entries being sought during each iteration. If an entry of `level` or below is found, it may be required — `level` is updated to match that of the entry under consideration, after which the entry's rectangle is examined. If contained in the descent subtree's region, it is added to the pending set. If it contains the subtree's region, this is the last entry required of this level number, and `level` is decremented.

Selection of entries for descent and pending sets for the execution of hyperspherical queries is performed, as in the abstract machine case, using the minimum distance between the query region and an entry region's outer boundary. Note, however, that if we wish to take the same approach to testing intersection with holey regions as in the abstract machine, we have no choice but to construct an explicit interpreted predicate. Figure 6.21 illustrates the intersection of a query region with two entry boundaries, but observe that the inner entry boundary forms a hole in the outer, so testing for intersection with outer boundaries alone suggests, erroneously, that both regions must be explored.

6.4.2 Splitting policy

A requirement of implicit predicate interpretation is that every region involved in the interpretation of a node's entries' predicates must remain available throughout any operation within the node. This includes the partitioning of a node's entries when the node is to split. When partitioning

```
...
public ArrayList<NodeEntry> insert(Point pt, NodeEntry parent,
    ArrayList<NodeEntry> pndIn, ArrayList<NodeEntry> dq)
  ...
  ArrayList<NodeEntry> local = readNode(pid);
  ArrayList<NodeEntry> all = new ArrayList<NodeEntry>();
  for(NodeEntry e:local)
    all.add(e);
  for(NodeEntry e:pndIn)
    all.add(e);
  Collections.sort(all);

  NodeEntry subtree;
  for(NodeEntry e:all)
    if(e.predicate().contains(pt)) {
      subtree = e;
      break;
    }
  int level = e.level() - 1;
  for(NodeEntry e:all)
    if(e.level() <= level) {
      level = e.level();
      if(subtree.predicate().contains(e.predicate())
        pndOut.add(e);
      else if(e.predicate().contains(subtree.predicate()) {
        pndOut.add(e);
        level--;
      }
    }
  ...
```

Figure 6.19: Interleaved predicate interpretation and subtree/pending set selection in the recursive BV-tree implementation.

```
public class NodeEntry implements Comparable<NodeEntry> {
...
  public int compareTo(NodeEntry e) {
    if(e.level != this.level)
      return e.level - this.level ;
    if(e.predicate.contains(this.predicate))
      return -1 ;
    if(this.predicate.contains(e.predicate))
      return 1 ;
    return 0 ;
  }
  ...
```

Figure 6.20: The `NodeEntry` class's implementation of the `Comparable` interface.

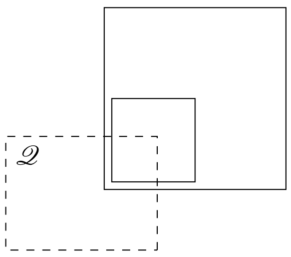


Figure 6.21: Intersection between holey BV-tree regions and arbitrary query rectangles. Both BV-region outer boundaries (solid) intersect the query rectangle (dashed), but the inner represents a hole in the outer.

explicitly interpreted entries, this happens as a result of reinterpreting local entries with respect to an incoming pending set, but this is not the case here, where predicates are interpreted on-the-fly.

This is the chief source of differences between the splitting policy for the recursive BV-tree implementation and that of the abstract machine. The major differences are as follows:

- All entries, local and pending, are passed to the split policy when partitioning a node's entries;
- pending entries are partitioned normally when calculating the node split, but excluded from considerations of balance;
- pending entries are excluded from the splitting policy's output.

This approach creates the possibility that, if one or more of the entries identified for elevation is a pending entry, those entries need not be elevated (in fact, they already are). The critical point here is that, in the absence of pending entries, it may be impossible to interpret a local entry's predicate fully, so it may appear to require elevation when, in fact, it does not. In such cases, addition of the pending entry illustrates that the local entry does not require elevation, because the pending entry is already elevated.

Figure 6.22a shows a BV-tree mid-execution of an insertion. Insertion into a leaf has caused that leaf to split, promoting entry L_0 into node N, which has a primary capacity of six entries and is now overflowing. Pending entry C_0 is also present. The geometry of these regions is given in figure 6.22b. Region B is $[0, 0.5)^2$; A, not shown in full, is $[0, 1)^2$. The boundary of region X (the region component of entry X_1) is coincident with A and that of Y is shown dashed in figure 6.22b. The other dashed boundary, Z, in 6.22b, indicates the boundary on which the overflowing node must split. This example illustrates precisely why pending entries must be considered at node split:

- If all eight entries (including C_0) are considered, the resulting partitioning is $[\mathrm{A}_0, \mathrm{B}_0, \mathrm{F}_0, \mathrm{H}_0]$ and $[\mathrm{J}_0, \mathrm{K}_0, \mathrm{L}_0]$, with entry C_0 selected for elevation. C_0 is not local to node N — it is an incoming pending entry — so no actual elevations are required.
- If only the seven local entries entries (excluding C_0) are considered, the resulting partitioning will be $[\mathrm{A}_0, \mathrm{F}_0, \mathrm{H}_0]$ and $[\mathrm{J}_0, \mathrm{K}_0, \mathrm{L}_0]$, with entry B_0 selected for elevation. B_0 is local to node N and will be physically elevated, but in fact this is not necessary.

Note that an elevation of the kind described above, while unnecessary, does not break the BV-tree guarantee. However, it demonstrates that we can improve on the lower performance limit guarantee, and where possible should do so.

6.5 KDB-VFS tree implementation

Our KDB-VFS tree implementation, like Robinson's original K-D-B tree, describes its spatial decomposition using explicit rectangles rather than an intranode kd-tree. This means that no further predicate interpretation is required. When a node overflows, its entries are partitioned using a split plane that bisects the node's region, and any entries that do not lie wholly to one side or the other of that plane are elevated into the level above. Unlike the approach to splitting a node described for the BV-tree using a BANG-style decomposition, the splitting policy employed here generates a number of partitionings, then selects the 'best' according to a metric described below.

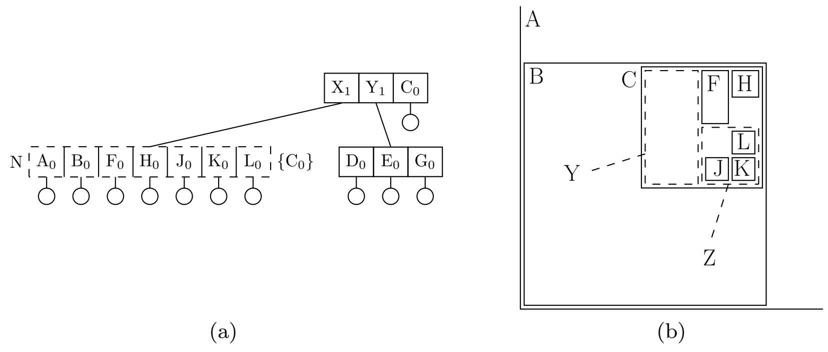


Figure 6.22: Implicit predicate interpretation requires consideration of pending entries at node split. Failure to consider pending entry C_0 when splitting node N leads to the unnecessary elevation of entry B_0.

6.5.1 Splitting policy

The KDB-VFS splitting policy generates possible split planes for an overflowing node based on the node's contents. In a d-dimensional space, a split 'plane' for a region consists of a projection of the region into $(d-1)$-dimensions, and a location in the dth dimension, such that the plane bisects the region in the dth dimension at the given location. When splitting a leaf node, each point in the leaf is used to generate the d, $(d-1)$-dimensional planes in which it lies, while in an internal node, a region is used to generate the $2.d$, $(d-1)$-dimensional planes in which each of its faces lie. For each plane generated, the node's contents are partitioned into those which lie to one side of the plane and those to the other. For convenience we will refer to these as 'left' and 'right'. In the case of internal nodes, a region may lie to neither side but be bisected by the plane; entries with these regions must be elevated.

When we introduced the KDB-VFS tree in section 4.5.2, we did so for the reason that its region descriptions made it simpler to handle when discussing concepts around virtual forced splitting; it is not intended to be a practical access method implementation. Our principal reason for this expectation is that there is no guaranteed upper bound to the number of entries that might require elevation at node split. This expectation led us to use the following heuristic for evaluating the quality of entry partitionings at node split: we choose the partitioning that requires the fewest number of entries to be elevated while remaining within permitted split balance.

When splitting an overflowing KDB-VFS node, our splitting policy acts as follows:

1. Identify potential split planes as described above;

2. For each split plane, partition entries and evaluate the quality of the partitioning as follows:

    (a) Select the partitioning with fewest elevated entries and in which both left and right partitions are at least 25% full;

    (b) Resolve ties by selecting the partitioning with best left/right balance (*i.e.* smallest difference between the number of entries on either side of the split plane, excluding any elevated entries);

    (c) Resolve ties by selecting the partitioning with 'preferred' split orientation. Our preference is to cycle through splitting dimensions, as in the BV-BANG case, so the preferred

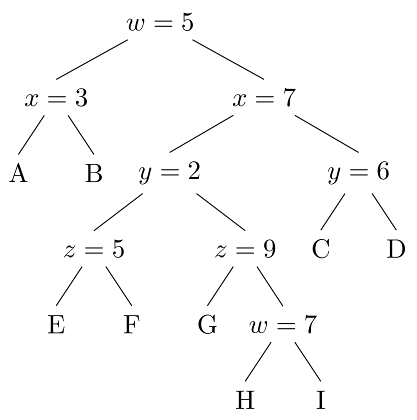


Figure 6.23: kd-tree partitioning of a four-dimensional space into nine regions.

Region	**Extent in dimension**				**Partition when split on**							
	w	x	y	z	w=5	x=3	x=7	y=2	y=6	z=5	z=9	w=7
A	[0,5)	[0,3)	[0,10)	[0,10)	L	L	L	U	U	U	U	L
B	[0,5)	[3,10)	[0,10)	[0,10)	L	R	U	U	U	U	U	L
C	[5,10)	[7,10)	[0,6)	[0,10)	R	R	R	U	U	U	U	U
D	[5,10)	[7,10)	[6,10)	[0,10)	R	R	R	R	R	U	U	U
E	[5,10)	[0,7)	[0,2)	[2,5)	R	U	L	L	L	L	L	U
F	[5,10)	[0,7)	[0,2)	[5,10)	R	U	L	L	L	R	U	U
G	[5,10)	[0,7)	[2,10)	[0,9)	R	U	L	R	U	U	L	U
H	[5,7)	[0,7)	[2,10)	[9,10)	R	U	L	R	U	R	R	L
I	[7,10)	[0,7)	[2,10)	[9,10)	R	U	L	R	U	R	R	R

Figure 6.24: Partitioning of figure 6.23's nine regions using each of the kd-tree's split planes.

orientation is the one after that of the last executed split (not necessarily the last preferred split direction);

(d) If more than one candidate remains, just pick one.

In the case of leaf nodes, the same criteria are used, starting with criterion 2b (since points cannot be elevated).

6.5.2 Occupancy guarantees

Experimentally, we observed that, in more than three dimensions, it was often not possible to partition a node's entries and require that the resulting nodes be more than 25% full, and so, in all experiments, the minimum permitted node occupancy was set at 25%. We make no assertions about this observation; in particular, it may be that KDB-VFS trees used to index spaces of higher numbers of dimensions than those tested ($\leqslant 16$) would require an even lower minimum occupancy bound.

Figure 6.23 gives a kd-tree description of the subdivision, into nine regions, of a four-dimensional, hypercubic space with sides $[0, 10)$. This is not shown geometrically for obvious reasons. The axes of the space are labelled w, x, y and z. Figure 6.24 summarises the possible partitionings of the eight entries using each of the split planes in the kd-tree; L indicates that an entry lies to the left of the split plane, R that it lies to the other, and U that it lies across the split plane and would require elevation.

We make two observations concerning the partitionings in figure 6.24:

- The less well occupied partition on either side of the split plane never contains more than 2 entries;
- one partitioning (on $w = 5$) in which two entries lie on one side of the split plane requires no elevations.

If we take the nine entries to belong to an overflowing node with capacity for eight entries, this example illustrates that there certainly exist cases in which no higher occupancy than 25% can be found, but also illustrates that lowering the minimum occupancy limit so far might remove the need to elevate entries altogether. As we describe in chapter 7, this turned out to be the case experimentally; in almost no situations did KDB-VFS trees under evaluation undergo a node split that required entries to be elevated. This alone makes it a rather unlikely candidate as a practical VFS-structure — it almost never uses virtual splitting.

6.6 Summary

In this chapter we described salient features of our implementation; specifically the details necessary to transform the collection of VFS operations described in chapter 5 into a working implementation, of either a BV-tree or a KDB-VFS tree. The KDB-VFS tree implementation, while never intended to be a practical VFS-tree implementation, proved to behave rather unexpectedly, in that our choice of node splitting policy all but prevented it from using virtual forced splitting at all.

We implemented the BV-tree in two different ways, and were able to demonstrate that the two were equivalent. Abstracting the detail of predicate interpretation out of the core tree code and into a lower layer permits a more intuitive description of the BV-tree's operation, by separating issues of selecting entries or subtrees from the predicate interpretation necessary to make that selection. Furthermore, we found that some aspects of the recursive implementation could only be handled using explicitly interpreted predicates. We believe that this bears out our suggestion, made in section 2.4.3, that complex structures are more effectively described in terms of the higher-level predicates that they represent than as collections of lower-level data.

Chapter 7

Experimental work

7.1 Introduction

The primary purpose of experimental work reported in this chapter is to provide some performance data for the BV-tree, which we believe to have been unimplemented until now. For the purposes of comparison we include data for our 'theoretical' VFS access method, the KDB-VFS tree. Trees were constructed using three datasets of eight different dimensionalities in the range 2 to 16. All trees were implemented in a common test framework, as described in chapter 6, enabling us to ensure a consistent approach to implementation of shared functions and instrumentation. Our principal benchmark is our R*-tree implementation, although for interest (and for a degree of sanity-checking) we include an implementation of Guttman's original R-tree, using the quadratic split policy.

7.1.1 Tree setup

Trees were implemented using the common `NodeEntry` entry format described in chapter 6. The `level` component of this structure is of use only in VFS-trees, and is set to `-1` in other structures. As described in chapter 6, an n-dimensional *Predicate* rectangle is implemented as a list of n intervals bounded by floating-point numbers. Intervals (and therefore rectangles) are closed in R-tree family structures, *i.e.* a pair of floating-point numbers a and b is interpreted to mean the interval $[a, b]$. Region disjointness in VFS-structures, however, requires that intervals be half-open — we choose the form $[a, b)$ — to ensure that a point lying on a boundary between two or more regions is contained in only one region. To ensure that all test datasets are indexable by all structures, we therefore use datasets confined to a half-open, n-dimensional unit cube, $[0, 1)^n$.

Page size was set to accommodate a maximum number of 24 entries per page in 16 dimensions; the maximum number of entries per page in two dimensions is 110. Page occupancy constraints were set as recommended in structures' original presentations: the R-tree has a minimum page occupancy of 1/3 [25] and the R*-tree a minimum page occupancy of 40% [3]. The KDB-VFS tree minimum occupancy was set at 25% as described in chapter 6. The BV-tree's minimum occupancy is never measured explicitly, because its split policy generates partitionings sequentially until the inner/outer split ratio falls below 1, as described in section 6.3.3. This approach is guaranteed to produce partitionings with at least 1/3 occupancy, as observed by Freeston in [19], by appealing to the proof of minimum occupancy guarantees for kd-tree based decompositions provided in [39].

Disk IO cost is the sole metric considered here for performance evaluation, measured as a count of disk page reads and writes. VFS nodes are written atomically; those consisting of $n > 1$ pages require n disk IO operations for each read or write. We assume that the system has sufficient buffers available to retain in the buffer pool all pages required during a single operation. A page must be brought into memory from disk when first accessed, but if accessed again during the same operation, for example when posting entries after a node split, when reinserting entries into the R*-tree, when demoting BV-entries or when visiting a BV-node more than once (for each RVFS-node represented therein), then the page will be found in the buffer pool and not incur a further read cost.

Our justification for this approach is that we wish to avoid any one operation encountering a page fault in the buffer pool, for the reason that this would introduce another variable, not directly related to tree behaviour — that of the buffer pool's page replacement policy. Furthermore, in section 7.2.4, we argue that the IO cost of using an index structure should always be considered alongside that of answering a query without using an index. Query execution cost should never be allowed to exceed that of answering the query by sequential scan, and, in our experiments, the largest sequential file was under 7.5MB in size. This should be well within the capacity of any real buffer pool.

7.1.2 Datasets

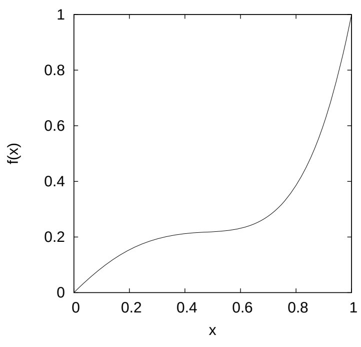
We took the decision to generate artifical datasets for use in our assessment, for the principal reason that it is difficult to obtain sufficient quantities of real-world data with enough flexibility to support a wide range of investigations. We considered real datasets drawn from the machine-learning community[1], but, as we wish to evaluate structures' performance in datasets of varying numbers of dimensions, we require a collection of datasets with similar data distribution characteristics in different dimensionalities.

We describe below the production of three 'primary' datasets of 50000 unique points in 16 dimensions (the Java code for dataset generation is given in appendix A). For each primary dataset, related 'secondary' datasets in 2–14 dimensions were produced by projecting the appropriate number of attributes. Each secondary dataset was subsequently tested for point uniqueness before use. In the case of the 2-dimensional CL dataset (see below), this required the exclusion of four points; in all other cases every point in the secondary dataset was found already to be unique. As described above, all three of our datasets are confined to the unit space with an open upper bound, $[0, 1)^n$.

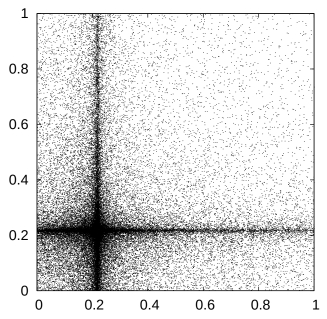
UN: Uniform

A point consists of an 16-dimensional feature vector $[p_1, \ldots, p_{16}]$, produced using 16 pseudorandomly generated numbers in the interval $[0, 1)$. 50000 such points were generated to produce a dataset uniformly distributed across the space.

[1]A large repository is available at `http://archive.ics.uci.edu/ml/datasets.html`.



(a) $f(x) = x^5 + x^4 - x^3 - x^2 + x$



(b) Two-dimensional projection

Figure 7.1: Point density function and two-dimensional PN dataset.

PN: Polynomial

50000 16-dimensional feature vectors $[p_1, \ldots, p_{16}]$ were generated for this dataset by calculating each p_i $(1 \leqslant i \leqslant 16)$ from a pseudorandomly generated x_i in the interval $[0, 1)$ as follows:

$$p_i = x_i^5 + x_i^4 - x_i^3 - x_i^2 + x_i$$

A graph of $f(x) = x^5 + x^4 - x^3 - x^2 + x$ is given in figure 7.1a, from which it can be seen that the range of $f(x)$ is $[0, 1)$ for $0 \leqslant x < 1$. When used to generate 16-dimensional points, the flat region of the function yields a skewed distribution, heavily concentrated around the value 0.2 in each dimension. A two-dimensional projection is given in figure 7.1b, from which it can be seen that using the same function in more than one dimension has the effect of producing a single cluster centred close to 0.2^n. The coaxial smearing of density is a consequence of our approach of distributing feature values along axes independently of one another.

CL: Clustered

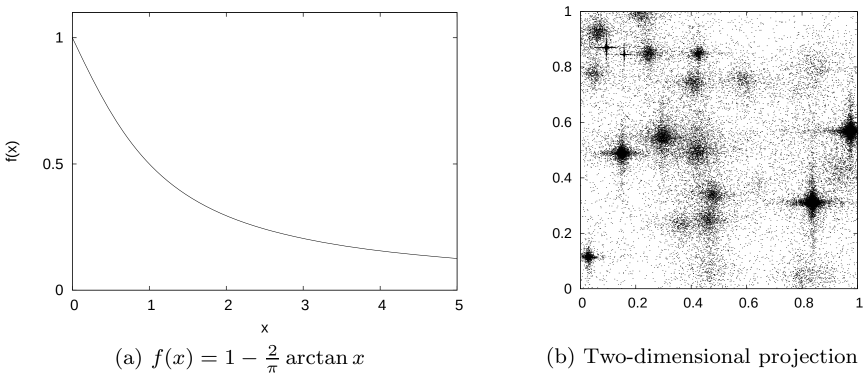
The CL dataset consists of 50000 points distributed in clusters. A cluster is described by a triple, $\langle c, r, s \rangle$, consisting of:

- its centre, c, a 16-dimensional feature vector produced from 16, independently generated, pseudorandom numbers in the interval $[0, 1)$;
- a pseudorandomly generated radius, r, in the interval $[0, 1/\sqrt{5})$;
- a pseudorandomly generated cluster size s of up to 10000 points

(the scaling factors for cluster radius and size were chosen heuristically for the desired character of distribution). We define a cluster density function:

$$f(x) = 1 - \frac{2}{\pi} \arctan x$$

and distribute cluster point coordinates in each dimension across the cluster radius by calculating $f^{-1}(x).r$ for pseudorandom, independently generated values of $f(x)$. Location either above or below the centre point in each dimension is decided by coin-toss. Points that, once generated, fall outside the unit cube $[0, 1)^{16}$ are discarded.



(a) $f(x) = 1 - \frac{2}{\pi} \arctan x$ (b) Two-dimensional projection

Figure 7.2: Cluster density function and two-dimensional CL dataset.

The cluster density function is shown in figure 7.2a; notice that the cluster 'radius' is actually the distance at which cluster density falls to half that at the cluster centre, and that density actually decreases asymptotically with distance from the cluster centre.

7.2 Results

7.2.1 Index construction

The cost of building each tree is expressed in average disk page IOs per insertion; this is the total page read and write cost for each insertion, divided by the number of points in the tree. The cost of an individual insertion varies as the tree grows in height, or, in the case of VFS-trees, as elevated entries cause nodes to exceed the capacity of a single disk page.

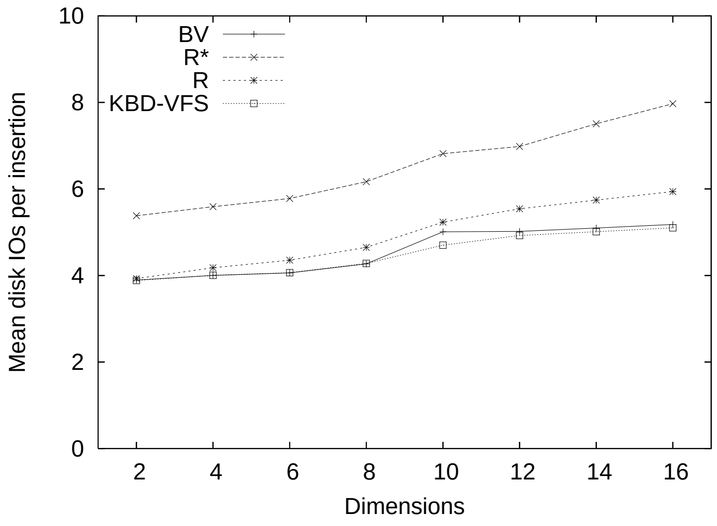
Figure 7.3 shows the cost per insertion of constructing trees using the PN datasets; the UN and CL datasets show very similar characteristics. In general, the KDB-VFS tree, the BV-tree and the R-tree show comparable insertion costs, but become slightly more expensive with increasing dimensionality. The R*-tree is rather more costly than the others, as might be expected from its forced-reinsertion behaviour at node split. This is not entirely consistent with results reported in [3], although Beckmann *et al.* remark that their result — that R*-tree insertion is cheaper than quadratic R-tree insertion — is surprising.

Overall, higher insertion cost tends to suggest a higher degree of reorganisation during insertion, because in any given number of dimensions, most trees were found to be the same height. This is reflected in the size of the resulting structure (see section 7.2.2), which in more regularly reorganised structures is somewhat smaller than is otherwise the case.

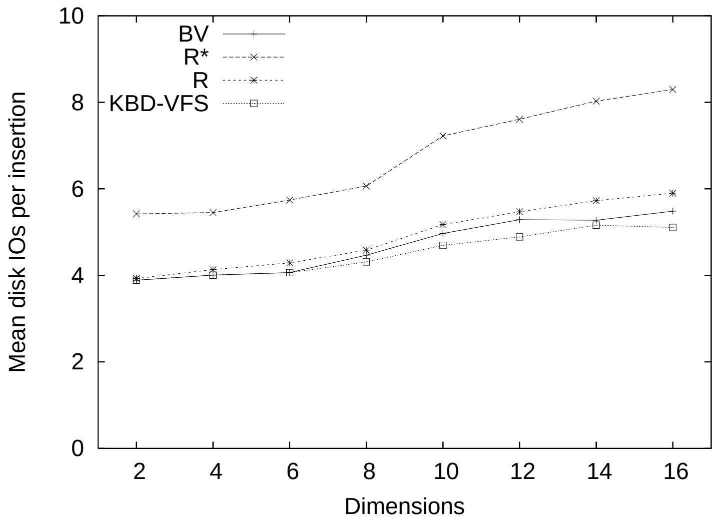
7.2.2 File size

Figure 7.4 shows the variation with dimensionality of file size, in disk pages, for datasets PN and UN; dataset CL is not shown but is broadly similar to the PN case. File size scales linearly with dimensionality, as might be expected from the linear increase in entry size, although the number of pages required to store an R or R*-tree increases at a lower rate than for the VFS structures when indexing the PN dataset. In the case of the R*-tree, we believe this to reflect the greater degree of reorganisation performed during tree construction.

Recall from chapter 6 that our implementations share a common entry structure. The effect

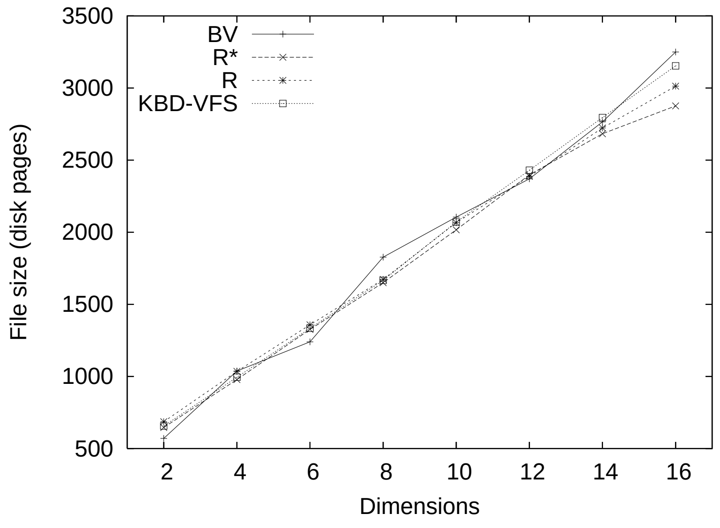


(a) Dataset UN

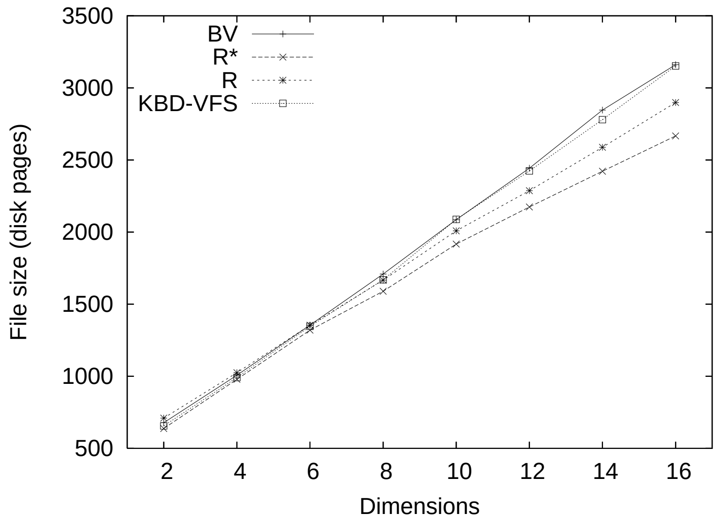


(b) Dataset PN

Figure 7.3: Mean insertion cost of tree construction.



(a) Dataset UN



(b) Dataset PN

Figure 7.4: Variation of file size with dimensionality.

of this is to limit every structure to a common maximum number of entries per page for a given dimensionality. In an implementation made outside this framework, however, we would choose an entry format and representation suited to each specific structure. In the case of the KDB-VFS tree, this would remain the same, but entries in the R- and R*-trees do not require a level number component. Such entries would be concomitantly smaller, resulting in slightly higher fanout for a given page size. If, however, the page size in such an implementation were to be reduced in proportion, we would achieve the same effect as here — the same maximum number of entries per page, and the same results as reported here in terms of page IOs.

We described, in section 3.3.5, the BANG file's mode of describing binary cyclic partitioning using bitstrings. BV-tree entries represented in this format are likely to achieve a significant saving in the size of an entry on disk; we consider this separately in section 8.1.1.

7.2.3 Exact match queries

Figure 7.5 shows the average cost of answering an exact match query in each structure for the CL and UN datasets, calculated over the individual cost of retrieving, in separate operations, every point known to be in each tree.

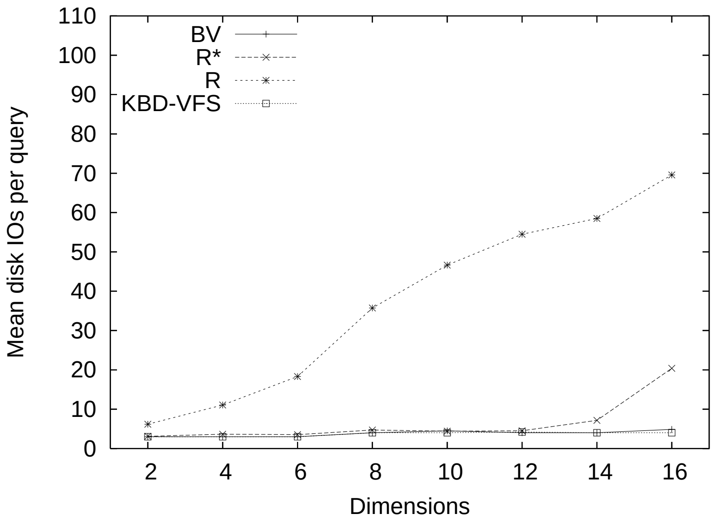
Unsurprisingly, the query cost for VFS structures is largely constant, irrespective of dimensionality. This is to be expected of both the BV and KDB-VFS trees as both provide the single path property. More surprising, however, is the R*-tree's performance in up to 12 dimensions. Using the visualisation software described in chapter 6 we were able to see a stark contrast between the R and R*-trees' ability to partition space in two dimensions, the latter doing so almost disjointly. These results suggest that, over uniform data, the R*-tree's ability to form disjoint partitions is not limited to spaces of very low dimensionality alone.

Notice that, in the R-tree case, query performance flattens off in more than 12 dimensions. Our interpretation of this result is based on the fact that as dimensionality increases, the space becomes more sparsely occupied; we contend that this is an artefact of the size of the dataset under investigation rather than a feature of the R-tree's performance *per se*. In an experiment designed to investigate solely the effect of increasing dimensionality, it might be more appropriate to scale the size of the dataset with dimensionality, to maintain a comparable data density across all dimensions (leading inevitably to very large datasets in high numbers of dimensions). Our decision to use datasets of constant size is pragmatic: we attempt to model the situation in a likely DBMS implementation.

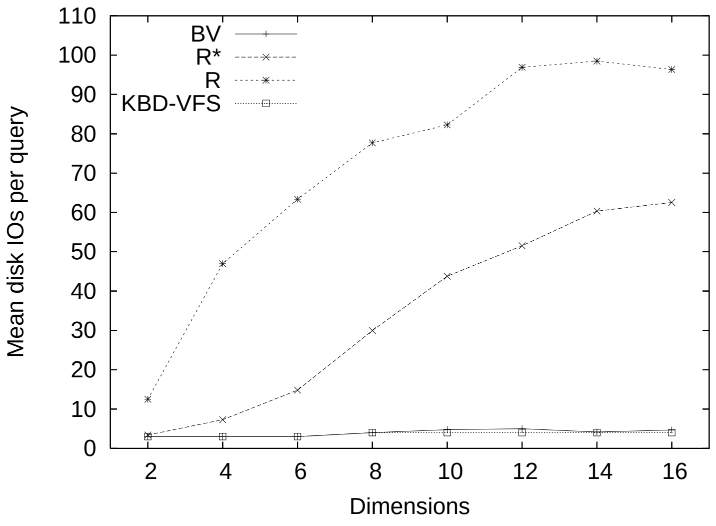
7.2.4 Nearest neighbour queries

K-Nearest Neighbour queries were executed in the R- and R*-trees using Hjaltason and Samet's optimal algorithm [6, 29]. This is optimal in the sense that it visits only those pages that it would be necessary to visit if the distance between the query point and its Kth nearest neighbour were known in advance, and the query executed as a range query. As described in section 5.2.2, this also forms the basis of our VFS-tree K-NN algorithm. For this reason, we do not describe separately range query performance at all: it is the same as K-NN performance.

Before examining the comparative performances of the various structures over different datasets, we make some remarks on the utility of index searching under conditions of deteriorating search efficiency.

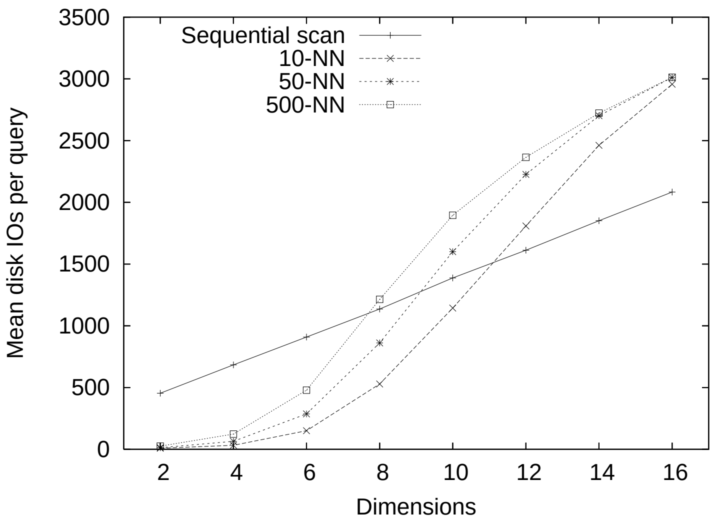


(a) Dataset UN

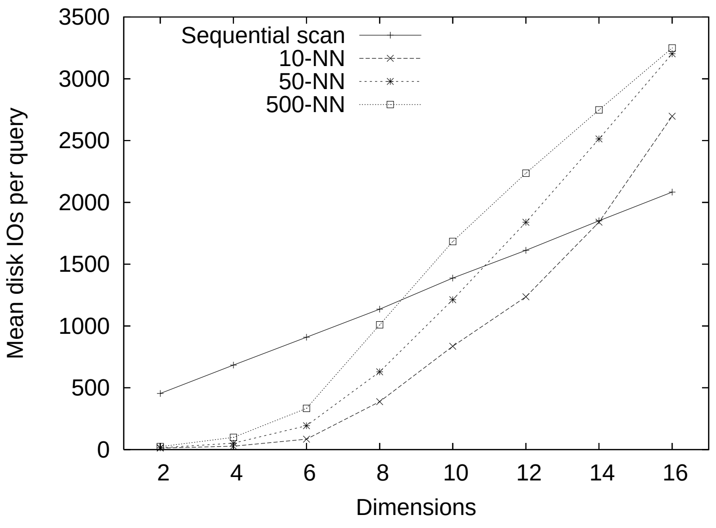


(b) Dataset CL

Figure 7.5: Exact match query costs.



(a) R-tree



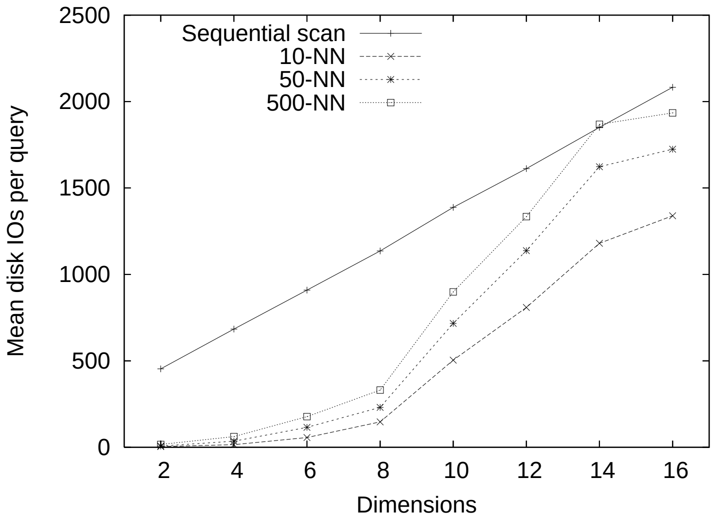
(b) BV-tree

Figure 7.6: Cost of K-NN queries in the R-tree and BV-tree, indexing dataset UN.

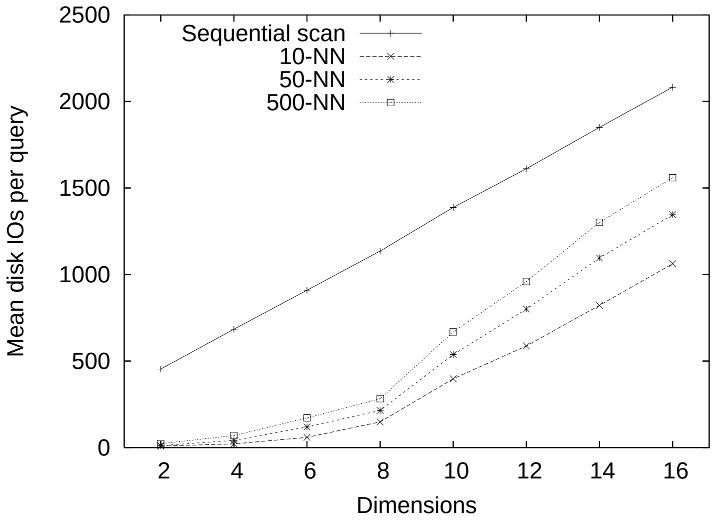
Index utility

Tree structures can use significantly more disk space than would be required to represent the indexed data in an unstructured, 'flat' file, firstly because of the disk space required to store the directory, and secondly because leaf pages are rarely fully occupied. This overhead is usually acceptable because the tree structure permits more rapid searching of the data set. In cases where a search accesses most of a tree's pages, however, it becomes more efficient to answer a query by sequential scan. The entire flat file must be read to answer queries with extent, but exact-match queries, over data sets without duplicates, can be answered by reading, on average, half of the file's pages.

We calculated the space required to store each data set in a flat file, and, in the case of exact-match queries, found it to be cheaper to use any of the index structures than to answer queries by sequential scan (reading half the flat file's pages). However, the cost of answering a query with



(a) R*-tree



(b) BV-tree

Figure 7.7: Cost of K-NN queries in the R*-tree and BV-tree, indexing dataset CL.

extent using a tree increases with dimensionality, until it becomes more expensive to search using the tree than to perform a sequential scan of the entire dataset in a flat file. Figure 7.6a shows the cost of answering a 10, 50 and 500-NN query using an R-tree, against the cost of reading the entire sequential file. The graph shows that, in more than 10 dimensions, even a 10-NN query is more expensive to answer using the R-tree than using a sequential scan. Note also that, by $d = 16$, query cost for 10, 100 or 500-NN converges to a single value — the cost of reading the entire tree. This exceeds the cost of a sequential scan because the tree occupies more space on disk than the flat file. This pattern is repeated, to a greater or lesser extent, in all trees and datasets, and in all but a few cases it is cheaper in 16 dimensions to answer 10-NN queries by sequential scan; figure 7.6b shows this to be the case for BV-trees indexing uniformly distributed data. Particular exceptions include the R*- and BV-trees indexing dataset CL; query costs for these structures are given in figure 7.7.

Query performance

Figure 7.8 gives the average cost of retrieving 10, 100 and 500 nearest neighbours of a query point using structures built over the CL datasets. Sequential scan performance is included as a benchmark — tree structures answering a query in more IOs than a sequential scan are of little interest. The comparative performances between the various structures are repeated very clearly for the retrieval of different numbers of nearest neighbours. The BV-tree, the R*-tree, and, perhaps surprisingly, the KDB-VFS tree show similar performance in up to eight dimensions, and above $d = 8$ begin to diverge; between $d = 10$ and $d = 16$ the most efficient access method is the BV-tree, followed by the R*-tree, and then the KDB-VFS tree. The quadratic R-tree is less efficient in every case.

In fewer than 16 dimensions, the BV-tree and R*-tree outperform all other access methods, although from figure 7.8c it seems likely that even these are likely to be displaced by the sequential scan at dimensionalities not much greater than 16.

Figure 7.9 provides the same set of information as figure 7.8, but for the UN dataset. We observe that, as in the CL case, the performance of access methods relative to one another is consistent across varying values of K. In the uniform data distribution, all tree structures struggle to search as efficiently as in the CL case; curves rise more steeply, show more similar performance between structures, and demonstrate that no structure is more efficient than a sequential scan at $d > 12$. In this case, the R*-tree outperforms the BV-tree consistently; this is not surprising given the results shown in section 7.2.3, in which exact-match query performance suggests that the structure has a low degree of inter-node overlap.

Results from trees built over the PN dataset are presented in figure 7.10. In this case, the BV-tree outperforms the R*-tree by a small margin. Ranking the datasets in the order UN, PN, CL, on a scale of decreasing uniformity, the progression of the BV-tree's behaviour along that scale would suggest that it is naturally more well-suited to the indexing of non-uniform data.

7.2.5 Window queries

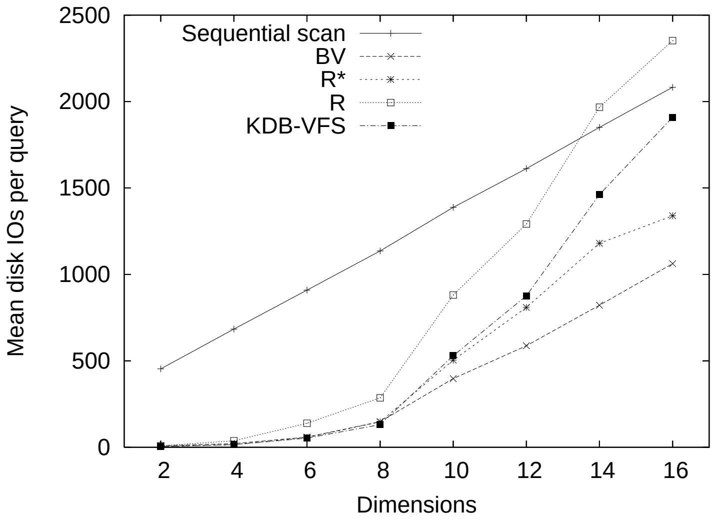
Region queries of hyper-rectangular shape are known as 'window queries'. In one sense these are no different from range queries, in that they specify a region of fixed extent, any point lying in which must be retrieved in the query result set. Recall that we did not present range query results explicitly, as the K-NN approach is optimal with respect to such queries, but we observe some interesting effects in query performance in the BV-tree due to its regular spatial decomposition, and present those results here.

We evaluate window query execution, in each case, by executing a number of queries over a given window size and finding the average disk IO cost to answer a query. The set of windows used are hypercubic with side length 0.625, placed with the hypercube's centre at various points along the diagonal of the space. We used two collections of windows:

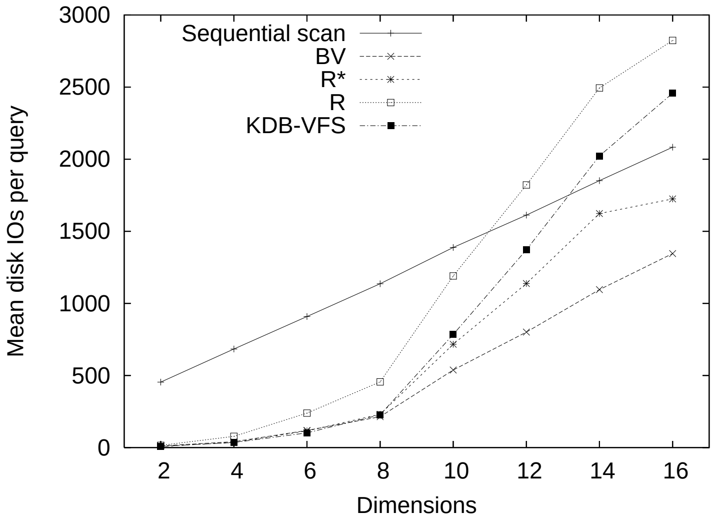
- **Window A**: 19 windows with centre points at

$$[p_1 : 0.05c, \ldots, p_n : 0.05c]$$

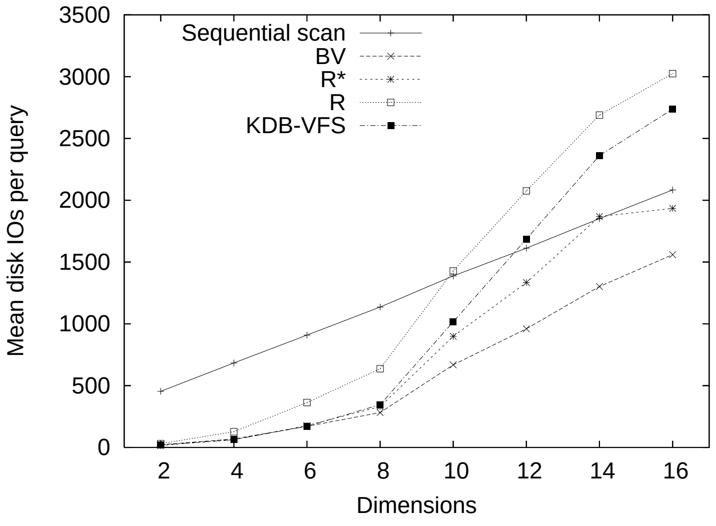
where n is the dimensionality of the space and $1 \leqslant c \leqslant 19$;



(a) 10-NN

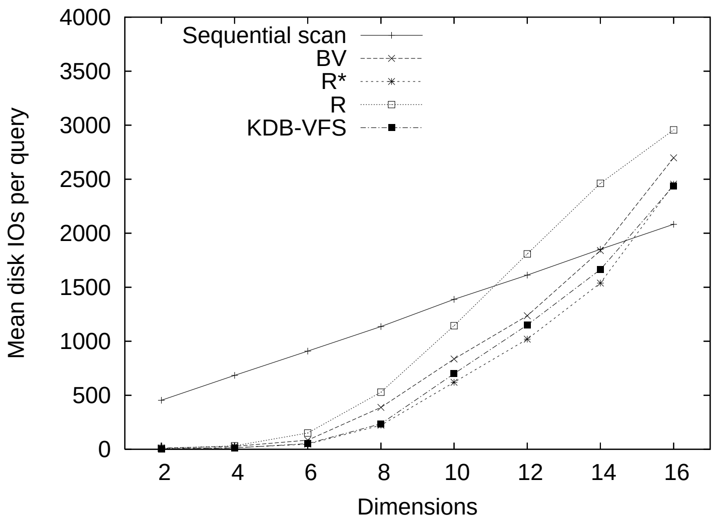


(b) 100-NN

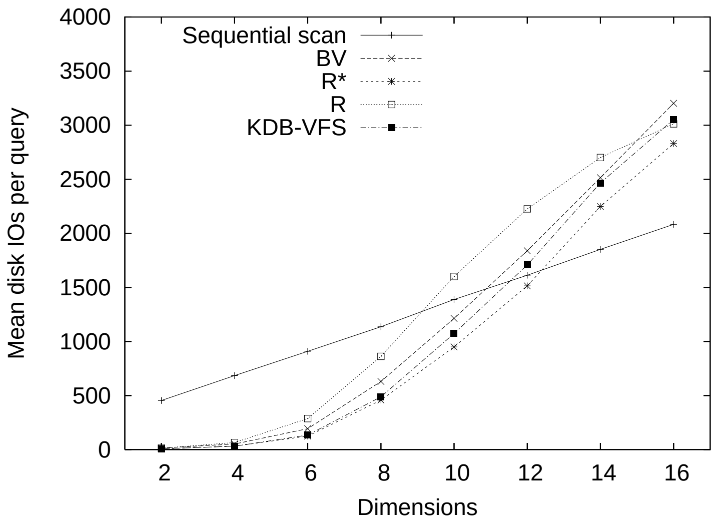


(c) 500-NN

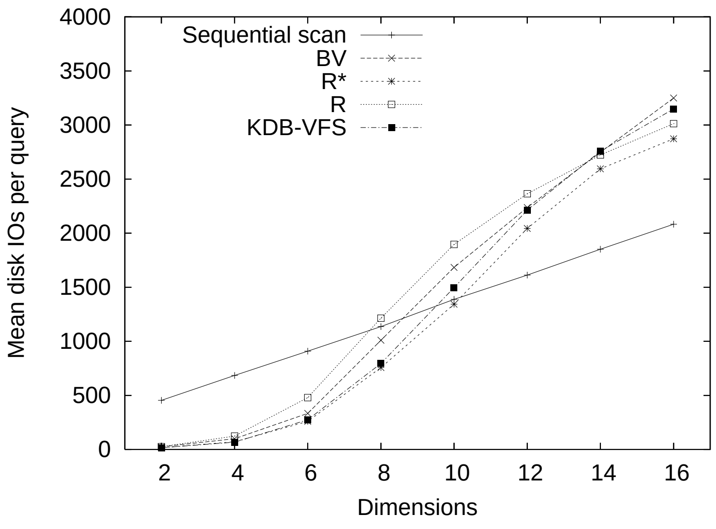
Figure 7.8: Comparative cost of K-NN queries in structures indexing dataset CL.



(a) 10-NN

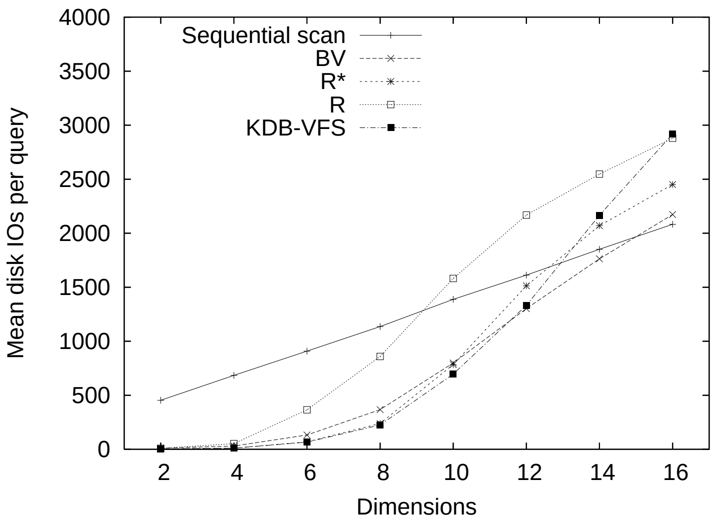


(b) 100-NN

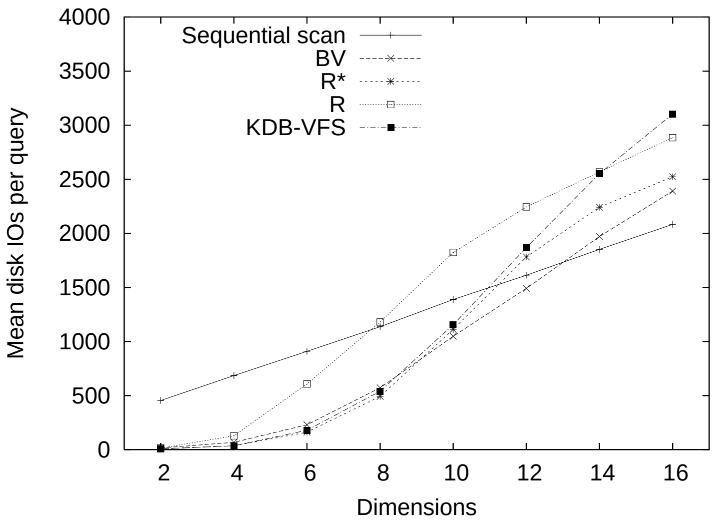


(c) 500-NN

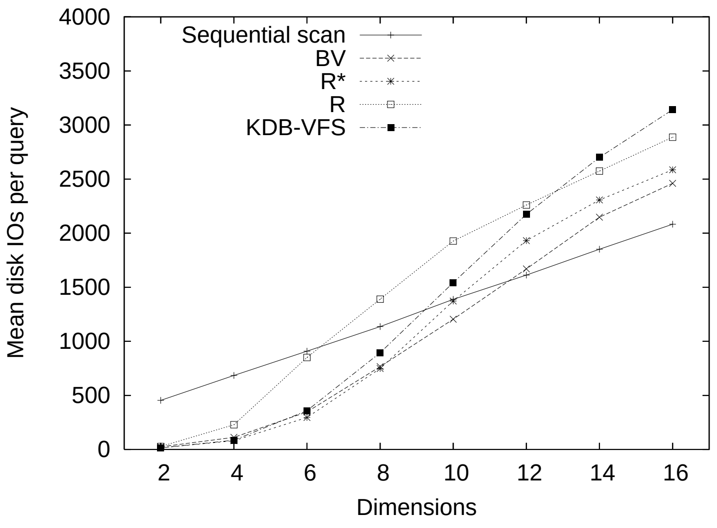
Figure 7.9: Comparative cost of K-NN queries in structures indexing dataset UN.



(a) 10-NN

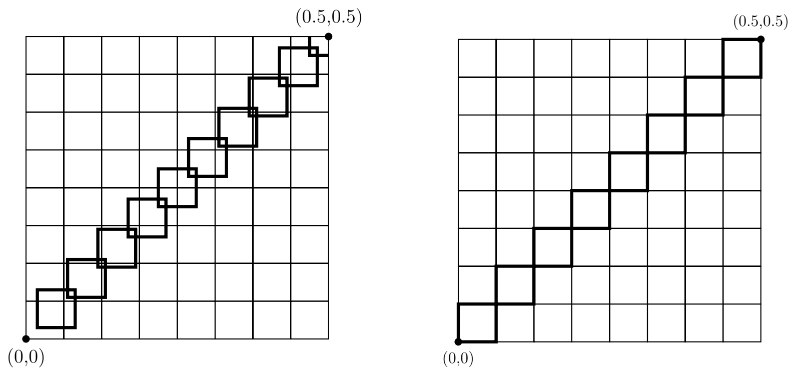


(b) 100-NN



(c) 500-NN

Figure 7.10: Comparative cost of K-NN queries in structures indexing dataset PN.



(a) Window A (b) Window B

Figure 7.11: Window placement for two sets of query results.

- **Window B**: 16 windows with centre points at

$$[p_1 : 0.03125 + 0.0625c, \ldots, p_n : 0.03125 + 0.0625c]$$

 where n is the dimensionality of the space and $0 \leqslant c \leqslant 15$.

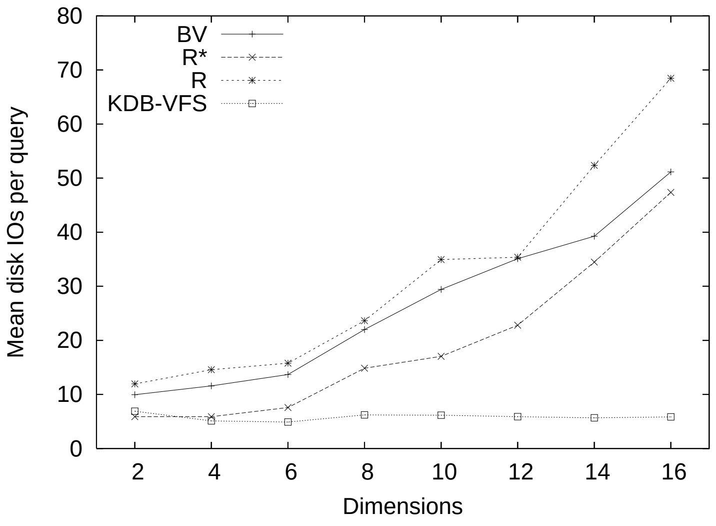
Figure 7.11 shows the $[0, 0.5)^2$ quadrant of the two-dimensional unit space under a BANG file decomposition, with these window schemes overlaid; notice that windows of set B coincide directly with binary partitions in the grid.

The results of running these two sets of window queries over the CL dataset are given in figure 7.12. In both the window A set of queries and the window B set, the KDB-VFS tree vastly outperforms the R*-tree, although the BV-tree only does so in the window B case; indeed in the window A case the BV-tree's performance deteriorates almost as far as that of the quadratic R-tree. With the exception of the BV-tree, structures' performance in answering queries over the two sets of windows is largely the same — the BV-tree's performance suffers badly when query regions lie across its prescribed boundary positions, as this vastly increases the number of BV-regions that the query region intersects.

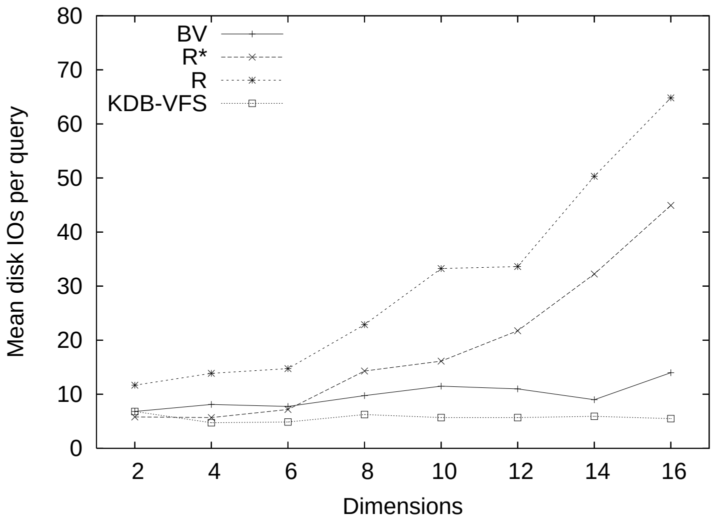
Observe that the sequential scan curve is omitted from the plots in figure 7.12, as it far exceeds the range of query costs presented here. This is not, however, an indication of some efficiency inherent in window query evaluation, but rather demonstrates that the volume of space enclosed by the query becomes proportionally less of the total volume of the space in higher numbers of dimensions. This means that, rather than returning a comparable number of results in each case (as K-NN algorithms do), instead the number of points returned by each window query falls rapidly with increasing dimensionality. On this basis, the KDB-VFS tree's apparently excellent performance can be explained by observing that these window queries have very little extent, and that there are few split positions that lie consistently across this extent.

7.3 Summary

In general, our results suggest that the BV-tree can index multidimensional datasets more efficiently than the R*-tree. Performance gains over the R*-tree are marked for exact-match searching, but expected, because the BV-tree provides the single path property. Importantly, the BV-tree also improves upon R*-tree performance for evaluation of queries with extent in non-uniform data



(a) Window A



(b) Window B

Figure 7.12: Cost of window query execution in CL datasets.

distributions. Interestingly, the KDB-VFS tree, in this implementation, is competitive with respect to the quadratic R-tree, but this is of little practical benefit given that the R*-tree is the *de facto* standard R-tree family implementation.

All structures evaluated here deteriorate in higher numbers of dimensions, often making the use of an index impractical in more than 10 or 12 dimensions. This is consistent with results reported in [8] for the R-tree family, indicating that the degree of internode overlap rises sharply above 2 dimensions, reaching a plateau approaching 100% at above 10 dimensions. Our presentation is unusual in that it provides an explicit sequential scan comparison, but we believe that other structures would fare similarly if compared likewise.

Chapter 8

Conclusions and Further work

In the thesis we explored in detail the concept of virtual forced splitting, in particular, capturing the notion that a VFS-tree is a compact representation of an underlying structure, an RVFS-tree. This notion enabled us to implement a number of tree operations, by describing them as a composition of mapping the VFS-tree to its underlying RVFS-tree, execution of the required operation thereon, and the translation of structural modifications back into the static VFS-tree representation.

Our approach has been driven throughout by consideration of subtrees using the regions they represent logically, as predicates, rather than as the (sometimes only partial) region descriptions with which they are associated physically. This enabled us, in the case of VFS-trees, to describe the difference between a VFS-subtree and the RVFS-subtrees that it represents. In the case of the BV-tree, which, even without consideration of its VFS-tree character, uses a complex region representation that must be interpreted from a list of physical entries, we were also able to describe clearly when and why a subtree must be elevated as a consequence of virtual splitting, and when it can be demoted.

The occupancy requirements of virtual splitting caused us to develop an approach that limits a node's primary occupancy, while allowing unlimited capacity for elevated entries. This allowed us to guarantee that a BV-tree implementation will not fail as a result of falling primary occupancy at higher levels, without allowing primary occupancy to exceed a single page and further reduce the tree's logarithmic exact-match search behaviour. Furthermore, we used this approach to propose another VFS-tree — the KDB-VFS tree — without an obvious upper bound on the number of elevated entries it might contain.

Elevation means that VFS-trees are not statically height-balanced (on disk) — but is it not the VFS-tree on which search operations are performed. The underlying RVFS-tree is height-balanced; every path from the root to the leaf level of the RVFS-tree contains the same number of nodes. The possible extension of a VFS-node to more than one disk page means that different paths through the RVFS-tree, while containing the same number of nodes, may require different numbers of pages to be read. VFS-trees may therefore not be IO-balanced, and practical VFS-tree implementations must provide limits on path IO-length variability.

The BV-tree's limits on elevation confine the variability of path IO-length to the range $[h, \Sigma_{i=1}^{h-1} i]$, making it a theoretically practical VFS-tree, and our implementation of the BV-tree demonstrated this experimentally, with performance — even using a rather inefficient mode of region representation — comparative to or better than the R*-tree.

In the following sections we consider some further research directions suggested by the work presented here.

8.1 BV-tree optimisations

8.1.1 Bitstring region representation

The DYOP file, mentioned briefly in section 2.4.5, identifies regions using a region number, indicating simultaneously a region's extent and position. BANG file regions can be numbered similarly, and in [22], Freeston describes the mapping of key values to the appropriate region number using the physical bitstring representation of each vector coordinate of a key value. We do not consider applying this approach directly, but note that the strict cyclic binary decomposition employed by the BANG file, and used in our BV-tree implementation, can be described by a string of bits.

Rectangular regions, in our implementation, are described by a pair of n-dimensional coordinates indicating the 'bottom left' and 'top right' corners of the region. Figure 8.1 shows the two dimensional unit space A, described in this representation as $\langle(0,0),(1,1)\rangle$, and two further regions: B, $\langle(0.25,0),(0.5.0.5)\rangle$; and C, $\langle(0.5,0.875),(0.625,1)\rangle$. Referring conventionally to the two dimensions as x and y, and using a partitioning order that cycles through the dimensions, beginning with the vertical, region B is formed by as follows:

- Split $\langle(0,0),(1,1)\rangle$ at $x = 0.5$, yielding $\langle(0,0),(0.5,1)\rangle$ and $\langle(0.5,0),(1,1)\rangle$;
- Split $\langle(0,0),(0.5,1)\rangle$ at $y = 0.5$, yielding $\langle(0,0),(0.5,0.5)\rangle$ and $\langle(0,0.5),(0.5,1)\rangle$;
- Split $\langle(0,0),(0.5,0.5)\rangle$ at $x = 0.25$, yielding $\langle(0,0),(0.25,0.5)\rangle$ and $\langle(0.25,0),(0.5,0.5)\rangle$. The latter is B.

Because the split is binary, however, we can describe each subsequent split using a single bit to indicate the lower or upper portion of the space. In the case of B, again:

- Split $\langle(0,0),(1,1)\rangle$ at $x = 0.5$, and choose the lower partition, **0**;
- Split $\langle(0,0),(0.5,1)\rangle$ at $y = 0.5$, and choose the lower partition, **0**;
- Split $\langle(0,0),(0.5,0.5)\rangle$ at $x = 0.25$, and choose the upper partition, **1**.

By concatenating the bits into a string, we form a different, smaller representation for B: "001". C can be described similarly as the bitstring "110101". If we were to use these bitstrings to represent the regions on disk, their representation would clearly by significantly cheaper, consisting of the bitstring and and an additional field for its length. In our Java implementation, B and C could be represented in as little as 3 bytes each — a single byte to contain the bitstring and a 2-byte `short` for its length, instead of 16 bytes to represent each region's four, floating-point, coordinates.

In our experiments, a node entry in 16 dimensions requires 136 bytes (including level and child information; `level` is currently implemented as a 4-byte `int` which is unnecessarily large). Using the scheme outlined above, the smallest regions form the longest bitstrings; the smallest region formed by the BV-tree built using the CL dataset in 16 dimensions would require 168 bits. By using a single byte for an entry's level number, and the bitstring region representation, even this, the tree's largest entry, would require only 28 bytes. Our implemented representation is nearly five times as large as this, and perhaps many times larger than that required for other entries described by shorter bitstrings. This suggests that the BV-tree's outperformance of the R*-tree reported here is likely to be improved upon significantly, and is a promising avenue of exploration.

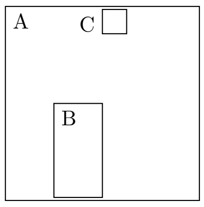


Figure 8.1: Bitstring representation of BV-BANG regions. Region B, $\langle(0.25, 0), (0.5, 0.5)\rangle$ can be represented as 001, and region C, $\langle(0.5, 0.875), (0.625, 1)\rangle$, as 110101.

8.1.2 Single demotes

Recall from section 5.3.2 that much of the complexity of demotion results from the fact that we must be able to execute safely the second demotion of a pair of entries that become demotable simultaneously. Note, however, as we described in section 5.3.4, demoting such a pair of BV-tree entries in the 'wrong' order may mean that only one undergoes a net demotion — if a hole is demoted first, its container may appear to remain virtually split. This situation could be regarded as tolerable, and offers some algorithmic simplifications.

While we have sought to demote every entry as far as is possible, the fact that a demotee can appear to remain virtually split indicates that the guaranteed elevation limit, of one elevated entry per level per primary entry, remains unbroken. Given that this is the case, if the split of an elevated entry is coincident with the boundary of the primary entry across which it is virtually split, we might instead decide to demote only the hole. Algorithmically, it would then be possible to execute the demote from within the original insertion operation, rather than having to schedule it for subsequent execution. This would allow us to do away with the demote queue entirely, and to place a smaller upper bound on the number of cascading demotions that might be caused by a single insertion, reducing it from 2^{h-2} to $h - 1$.

Restricting the BV-tree to single demotions offers a practical simplification of the insertion algorithm, and reduces its cost, without exceeding the BV-tree's guaranteed elevation limit. This makes it worthy of further investigation.

8.2 Other VFS-tree optimisations

8.2.1 Consideration of elevated entries *in situ*

In the VFS-tree, we introduced the use of pending sets as a means of extracting one or more RVFS-subtrees from an elevated VFS-subtree. In general, this is necessary because an elevated entry, as in the case of the BV-tree, may describe more space than its subtree actually indexes. In cases like the KDB-VFS tree, however, in which the space indexed by a subtree is exactly the union of the spaces indexed by the RVFS-subtrees that it represents, it may be possible to avoid using pending sets.

If, while searching a KDB-VFS tree for a point q, we find an elevated entry whose region contains it, we can guarantee that answering the query will require exploration of that entry's subtree. This is because the KDB-VFS tree's region descriptions have no holes. Acknowledging this, we might then decide to explore the elevated subtree *in situ*, saving the cost of reading at least one node. For example, if a level 0 entry is found in, and explored from, a level 1 node, we can skip the reading of a level 0 node and proceed directly to the leaf.

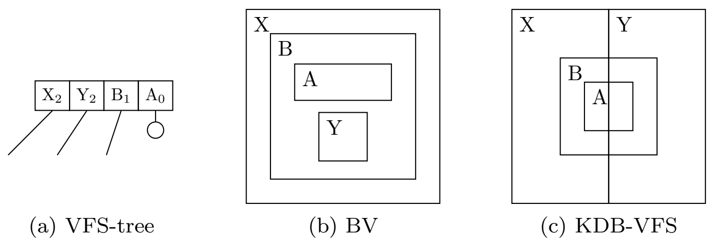


Figure 8.2: Demotability of entries in VFS-trees is heavily influenced by their mode of region representation.

In addition to allowing the execution of certain instances of operations to terminate 'early', this would also simplify the implementation of tree operations on such structures. Furthermore, in structures like the KDB-VFS tree, insistence on pending sets can actually prevent demotions that, intuitively, it seems could be executed correctly. Figure 8.2a shows a VFS-tree node of level 2, containing one elevated entry from each of levels 1 and 0. In the BV-tree case (figure 8.2b gives a possible spatial decomposition), it is unsafe to demote A_0 directly into B_1, because it may be virtually split, at level 1, across region B and an unelevated hole in B. Demotion must instead take place via X_2, even though A_0 may ultimately end up in the subtree of B_1.

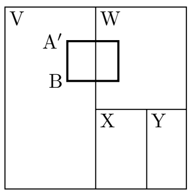
Figure 8.2c gives a possible KDB-VFS case. Here, A_0 and B_1 are both virtually split at level 2, but region A is guaranteed to be a subregion of region B, because B is fully described. This suggests that it should be possible to demote A_0 into B_1, although, unlike in the BV-tree case, it is not possible to do so via either X_2 or Y_2. The semantics of such a demotion in terms of RVFS-subtrees are not clear, and, while intuitively it seems correct to execute a demotion *in situ*, this needs careful consideration.

8.2.2 Hints for demotion

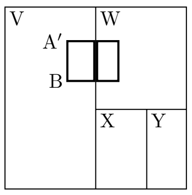
Demotion is key to limiting elevation in VFS-structures, and, particularly in structures to which no obvious formal limit applies, we seek to increase the potential for demotions as far as possible. A possibility for 'inducing' demotions is to provide demotion 'hints' in the form of information about the boundary across which an elevated entry is virtually split.

The idea is that, when caching an elevated entry in a pending set, we would cache with it the boundary or boundaries across which it is virtually split. If, on reaching its primary level, the pending entry were to be selected for descent, the virtual split position information would then be passed into the node. Should the node subsequently overflow and require splitting, partitioning on the virtual split boundary can be tested, and, assuming that the resulting split is within occupancy limits, can be selected in preference to other splits.

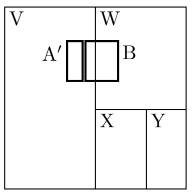
In a region representation like that of the KDB-VFS tree, this can be taken slightly further. Figure 8.3a shows region A, virtually split across regions V and W. By predisposing a split on the virtual split boundary, two demotions are induced (figure 8.3b), but even if a split is selected that is merely parallel to that boundary, at least one demotion can be induced (figure 8.3c). In the BV-tree case, however, this approach would only be of use if a split were to be induced that coincides exactly with the boundary of the region across which an elevated entry is virtually split, because any split of an elevated entry results in at least one demotion. This, is turn, makes this an optimisation that could not be applied to the single-demote tree that we postulate in section 8.1.2.



(a) Region A, virtually split across regions V and W.

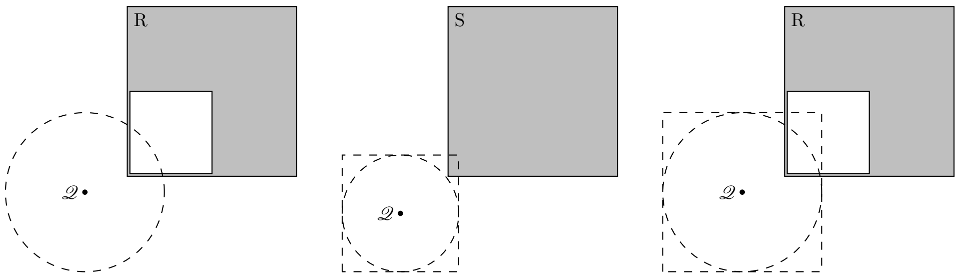


(b) VFS-coincident split of region A, permitting two demotions.



(c) VFS-parallel split of region A, permitting one demotion.

Figure 8.3: Inducing demotions by providing hints for a preferable split position or orientation.



(a) (b) (c)

Figure 8.4: Both direct and bounding box representations of a hyperspherical query region can reduce the potential for pruning subtrees. Using a combined representation may offer an improvement.

8.2.3 Improving pruning in hyperspherical queries

In section 6.3.2, we outlined our approach to deciding whether an arbitrary rectangular query region intersects a holey BV-tree region, by composing the intersection of the query and the BV-tree region's outer boundary, and testing for containment, in the holey region, of the result. We did not take a similar approach to handling intersection of hyperspherical regions, as in the case of K-NN queries, because of the difficulty in composing their intersection with a rectangle. This risks exploring some regions unnecessarily, because the region of intersection may be wholly contained in one of the BV-tree region's holes, as in figure 8.4a.

An alternative approach would be to test for intersection of such a query's bounding box with that of the BV-region, however, as in figure 8.4b, this carries the risk of failing to prune a non-holey region, and therefore exploring it unnecessarily.

A sensible optimisation might be to employ both approaches; to test for intersection between the query and a BV-region's outer boundary, and also to test for containment, by the BV-region, of its boundary's intersection with the query bounding box. In this approach, as in figure 8.4c, the minimum distance of $\mathscr{Q}$ from the bounding box might not allow R to be pruned, but the fact that holey region R does not intersect $\mathscr{Q}$'s bounding box will do so.

8.3 The abstract machine

8.3.1 Automatic code generation

Throughout the thesis, we have described tree operations as sets of transitions between states in an abstract machine, and in chapter 6 we described the implementation of a BV-tree by translating into Java, by hand, the associated set of transition rules. The abstract machine description is rather stricter than alternative pseudo-code descriptions, and indeed the act of translating the rules into Java highlighted a number of errors in the draft version from which we were working, because type errors in the rules were identified by the Java compiler. Furthermore, use of the stack to control tree traversal facilitates other aspects of debugging, because an operation's state is made explicit throughout its execution; progress of an operation can be monitored relatively cleanly with the use of traces such as those presented in chapter 5.

The fact that such a direct translation is possible, and that it forces type errors into the open, suggests that automatic generation of code from an abstract machine description might be possible. For efficient code this would require slightly more than a direct translation; for example, as we described in section 6.3.4, some reorganisation of the rules is necessary to ensure that the store is not read more than once for each transition executed. Performing this reorganisation automatically would require ordering of the rules by command, while preserving the conditional ordering used in their specification. In addition to avoiding multiple store reads, further dependency analysis would be required to avoid re-evaluating definitions shared between rules with the same input command.

While potentially complex, this is not infeasible; in essence we treat the abstract machine specification of access methods as a higher level programming language for their implementation. Translation of the rules is then analogous to program compilation, and could proceed either via a lower-level language (like our Java implementation) or directly into an executable format.

8.3.2 Reasoning about access methods

Reasoning formally about data structures is typically performed using purely functional data structures. Destructive, in-place updates such as those performed by external access methods that read from, and write to, the same disk page, makes formal analysis of these structures rather difficult. The store component of our abstract machine, which we have described as performing the functions of a buffer pool, has been used in a parallel area of research to reason about external access methods using separation logic. Specifically, the separation of the disk page undergoing modification from the rest of the store has allowed arguments about structure-wide invariants to be put forward. Some work by Sexton *et al.* [56] provides some results concerning correctness of operations in the B+ tree, and a similar approach could be taken to establish the correctness of the VFS-tree operations presented here.

8.4 Obstacles to DBMS-integration of the BV-tree

A number of obstacles remain between the implementation of the BV-tree and its integration into a working DBMS.

A significant issue, outside the scope of the thesis, is the requirement for concurrent access to the structure. Some aspects present obvious issues, for example the fact that entries undergoing elevation are removed from a node before being posted into another — such entries are effectively absent from the tree during elevation in our implementation. A likely approach to investigating issues of concurrency via the abstract machine framework is to consider first the application of a well-understood approach; for example that of Lehman and Yao [36] to the B+ tree.

In section 6.2 we described some constraints on the description of a space indexable with a symmetric binary decomposition. A practical implementation of a BV-tree would require, for any data type to be partitioned, a means of locating a splitting position at the midpoint of an interval. This is likely to require a mapping function of some kind, potentially with associated problems of locality preservation in the mapped space, and requires further investigation.

Appendix A

Dataset generation

The generation of the three datasets used for experimental evaluation, denoted PN, UN and CL, is described in chapter 7. The Java code used to generate the datasets is given in figures A.1, A.2 and A.3 respectively.

In each case, a point is produced as an array of floating-point numbers of length `MAX_DIM`. In our implementation, `MAX_DIM` was selected to be 16. The final step in the production of each point is a call to `writeOut`, a method whose implementation we do not give here, but whose purpose is to record the point in secondary storage; in our case this was in a PostgreSQL database.

Notice, in figure A.3, that points are accumulated by building one cluster at a time, using `buildCluster`. Cluster density is controlled by the cluster radius and its number of points (its `intendedSize`). If a point generated for a cluster is found to lie outside the unit cube, it is discarded, but no other point is generated in its place; this is to ensure that the cluster density remains as intended. However, to keep an accurate record of the growing dataset size, this approach requires us to return, from `buildCluster`, the actual number of points added to the dataset in the production of each cluster.

```
public void buildPN(int points) {
  float[] point = new float[MAX_DIM];
  for (int i = 0; i < points; i++) {
    for (int j = 0; j < point.length; j++) {
      float x = (float) Math.random();
      float x2 = x * x;
      float x3 = x2 * x;
      float x4 = x3 * x;
      float x5 = x4 * x;
      point[j] = x5 + x4 - x3 - x2 + x;
    }
    writeOut(point);
  }
}
```

Figure A.1: Java code for generation of the PN dataset.

```
public void buildUN(int points) {
  float[] point = new float[MAX_DIM];
  for (int i = 0; i < points; i++) {
    for (int j = 0; j < point.length; j++)
      point[j] = (float) Math.random();
    writeOut(point);
  }
}
```

Figure A.2: Java code for generation of the UN dataset.

```
public void buildCL(int target) {
  double scale = 1d / Math.sqrt(5);
  int points = 0;

  while (points < target) {
    int size = (int) (10000 * Math.random());
    double radius = Math.random() * scale;
    points += buildCluster(size, radius);
  }

}

public int buildCluster(int intendedSize, double radius) {
  // cluster centre
  float[] ctr = new float[MAX_DIM];
  for (int i = 0; i < ctr.length; i++)
    ctr[i] = (float) Math.random();

  int actualSize = 0;
  float[] point = new float[MAX_DIM];

  nextPoint:  for (int j = 0; j < intendedSize; j++) {
    for (int i = 0; i < point.length; i++) {
      // deflection from centre point in this dimension
      double y = Math.random();
      double xr = Math.tan((1 - y) * (Math.PI / 2)) * radius;

      // distribute around centre
      if (Math.random() < 0.5) // coin toss
        xr *= -1;
      point[i] = (float) xr + ctr[i];

      // if outside the unit cube, discard
      if (point[i] < 0 || point[i] >= 1)
        continue nextPoint;
    }
    writeOut(point);
    actualSize++;
  }
  return actualSize;
}
```

Figure A.3: Java code for generation of the CL dataset.

Bibliography

[1] Walid G. Aref and Ihab F. Ilyas. SP-GiST: An extensible database index for supporting space partitioning trees. *J. Intell. Inf. Syst.*, 17(2-3):215–240, 2001.

[2] Rudolf Bayer. The Universal B-tree for multidimensional indexing. Technical Report TUM-I9637, Technische Universität München, Germany, 1996.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[4] Richard Bellman. *Adaptive Control Processes: A Guided Tour.* Oxford University Press, 1961.

[5] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Comm. ACM*, 18(9):509–517, Sep. 1985.

[6] Stefan Berchtold, Christian Böhm, Daniel A. Keim, and Hans-Peter Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 78–86, 1997.

[7] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel. The Pyramid-Technique: Towards breaking the curse of dimensionality. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 142–153, 1998.

[8] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, San Francisco, U.S.A., 1996.

[9] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *Proceedings of the 7th International Conference on Database Theory*, pages 217–235, Jerusalem, Israel, 1999.

[10] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.

[11] Kaushik Chakrabarti and Sharad Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. In *Proceedings of the 15th International Conference on Data Engineering*, pages 440–447, 1999.

[12] Isaac Cheng, Gary Grossman, and Joe Hellerstein. Guided tour of `libgist v0.9b1`; source code for the GiST, release 0.9 beta 1, 1997. Available at URL `http://gist.cs.berkeley.edu/libgist/libtour.html`.

[13] P. Ciaccia and M. Patella. Bulk loading the M-tree. In *Proceedings of the 9th Australasian Database Conference*, pages 15–26, Perth, Australia, 1998.

[14] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 426–435, Athens, Greece, 1997.

[15] D. Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[16] G. Evangelidis, D. Lomet, and B. Salzberg. The hB$^{\Pi}$-tree: a multi-attribute index supporting concurrency, recovery and node consolidation. *VLDB J.*, 6:1–25, 1997.

[17] Christos Faloutsos and Shari Roseman. Fractals for secondary key retrieval. In *Proceedings of the 8th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 247–252, 1989.

[18] Michael Freeston. Begriffsverzeichnis: A concept index. In *Proceedings of the 11th British National Conference on Databases*, pages 1–22, Keele, UK, 1993.

[19] Michael Freeston. A general solution of the n-dimensional B-tree problem. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 80–91, San Jose, California, May 1995.

[20] Michael Freeston. On the complexity of BV-tree updates. In *Proceedings of the Second International Workshop on Constraint Database Systems*, pages 282–293, Delphi, Greece, 1997.

[21] Michael Freeston. Advances in the design of the BANG file. In *Proceedings of the 3rd International Conference on Foundations of Data Organization and Algorithms*, pages 322–338, Paris, France, June 1989.

[22] Michael Freeston. The BANG file: A new kind of grid file. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 260–269, San Francisco, May 1987.

[23] Michael Freeston. The application of multi-dimensional indexing methods to constraints. In *Proceedings of Constraint Databases and Applications, ESPRIT WG CONTESSA Workshop*, pages 102–119, Friedrichshafen, Germany, September 1995.

[24] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[25] A. Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Record*, 14(2):47–57, June 1984.

[26] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 562–573, Zurich, Switzerland, 1995.

[27] A. Henrich, H. W. Six, and P. Widmayer. The LSD tree: spatial access to multidimensional and non-point objects. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 45–53, 1989.

[28] Andreas Henrich. The LSD^{h}-tree: An access structure for feature vectors. In *Proceedings of the 14th International Conference on Data Engineering*, pages 362–369, 1998.

[29] Gisli R. Hjaltason and Hanan Samet. Ranking in spatial databases. In *Proceedings of the 4th International Symposium on Large Spatial Databases*, pages 83–95, 1995.

[30] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. iDistance: An adaptive B+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

[31] Norio Katayama and Shin'ichi Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 369–380, Tucson, Arizona, United States, 1997.

[32] James T. Klosowski, Martin Held, Joseph S.B. Mitchell, Henry Sowizral, and Karel Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. Vis. Comput. Graph.*, 4(1):21–36, 1998.

[33] Marcel Kornacker. `libgist v1.0`; source code for the GiST, December 1997. Available at URL `http://gist.cs.berkeley.edu/libgistv1/`.

[34] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964.

[35] Dong-Ho Lee and Hyoung-Joo Kim. SPY-TEC: An efficient method for similarity search in high-dimensional data spaces. *Data Knowl. Eng.*, 34(1):77–97, 2000.

[36] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[37] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB J.*, 3(4):517–542, 1994.

[38] Jakub Lokoč and Tomáš Skopal. On reinsertions in M-tree. In *Proceedings of the 1st International Workshop on Similarity Search and Applications*, pages 121–128, April 2008.

[39] David B. Lomet and Betty Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Syst.*, 15(4):625–658, Dec. 1990.

[40] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Yannis Theodoridis. *R-Trees: Theory and Applications*. Springer, 2006.

[41] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.

[42] Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Stephane Bressan. Indexing the edges: A simple and yet efficient approach to high-dimensional indexing. In *Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database systems*, pages 166–174, 2000.

[43] Jack A. Orenstein. Spatial query processing in an object-oriented database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 326–336, 1986.

[44] Jack A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 181–190, 1984.

[45] E. A. Ozkarahan and M. Ouksel. Dynamic and order preserving data partitioning for database machines. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 358–368, Stockholm, Sweden, 1985.

[46] Marco Patella. *Similarity search in multimedia databases*. PhD thesis, University of Bologna, 1999.

[47] Vladimir Pestov. On the geometry of similarity search: dimensionality curse and concentration of measure. *Inf. Process. Lett.*, 73(1-2):47–51, 2000.

[48] PostgreSQL Global Development Group. GiST implementation code, December 2007. Available at URL `www.postgresql.org/download/`; further information available at URL `http://www.sai.msu.su/~megera/postgres/gist/`.

[49] J. T. Robinson. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 10–18, Ann Arbor, Michigan, 1981.

[50] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):188–260, June 1984.

[51] Hanan Samet. Decoupling partitioning and grouping: Overcoming shortcomings of spatial indexing with bucketing. *ACM Trans. Database Syst.*, 29(4):789–830, 2004.

[52] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[53] Bernhard Seeger and Hans-Peter Kriegel. The buddy tree: an efficient and robust access method for spatial data base systems. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 590–601, 1990.

[54] Alan P. Sexton and Richard Swinbank. Bulk loading the M-tree to enhance query performance. In *Proceedings of the 21st British National Conference on Databases*, pages 190–202, Edinburgh, UK, July 2004.

[55] Alan P. Sexton and Richard Swinbank. Virtual forced splitting, demotion and the BV-tree. In *Proceedings of the 25th British National Conference on Databases*, pages 139–152, Cardiff, UK, July 2008.

[56] Alan P. Sexton and Hayo Thielecke. Reasoning about B+ trees with operational semantics and separation logic. In *Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics*, Philadelphia, PA, USA, May 2008.

[57] Markku Tamminen and Reijo Sulonen. The EXCELL method for efficient geometric access to data. In *Proceedings of the 19th ACM IEEE Design Automation Conference*, pages 345–351, Las Vegas, Nevada, June 1982.

[58] C. Traina, Jr., A. Traina, C. Faloutsos, and B. Seeger. Fast indexing and visualization of metric data sets using Slim-trees. *IEEE Trans. Knowl. Data Eng.*, 14(2):244–260, 2002.

[59] Wei Wang, Jiong Yang, and Richard R. Muntz. PK-tree: A spatial index structure for high dimensional point data. In *Proceedings of the 5th International Conference on Foundations of Data Organization*, pages 27–36, Kobe, Japan, 1998.

[60] David A. White and Ramesh Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th International Conference on Data Engineering*, pages 516–523, Washington DC, USA, 1996.

[61] Rui Zhang, Panos Kalnis, Beng Chin Ooi, and Kian-Lee Tan. Generalized multidimensional data mapping and query processing. *ACM Trans. Database Syst.*, 30(3):661–697, 2005.