Compacting Discriminator Information for Spatial Trees

Inga Sitzmann **Peter J. Stuckey**

Department of Computer Science and Software Engineering
The University of Melbourne
Parkville, Victoria, 3052
Email: {inga,pjs}@cs.mu.oz.au

Abstract

Cache-conscious behaviour of data structures becomes more and more important as memory sizes increase and whole databases fit into main memory. For spatial data, R-trees, originally designed for disk-based data, can be adopted for in-memory applications. In this paper, we will investigate how the small amount of space in an in-memory R-tree node can be used better to make R-trees more cache-conscious. We observe that many entries share sides with their parents, and introduce the partial R-tree which only stores information that is not given by the parent node. Our experiments showed that the partial R-tree shows up to 30 per cent better performance than the traditional R-tree. We also investigated if we could improve the search performance by storing more descriptive information instead of the standard minimum bounding box without decreasing the fanout of the R-tree. The partial static O-tree is based on the O-tree, but stores only the most important part of the information of an O-tree box. Experiments showed that this approach reduces the search time for line data by up to 60 per cent.

Keywords: Index structures, spatial databases, in-memory databases.

1 Introduction

Latest surveys have shown that the availability of cheap memory will lead to computer systems with main memory sizes in the order of terrabytes over the next 10 years [Bernstein et al., 1998]. Many databases will then fit entirely in main memory. For database technology, this means that the traditional bottleneck of memory-disk latency will be replaced by the cpu-memory latency as the crucial factor for database performance. It is therefore increasingly important for database index structures and algorithms to become sensitive to the cache behaviour.

Recent research on index structures includes several papers addressing cache-sensitive index structures. In [Rao and Ross, 2000], the authors propose a pointer elimination technique for B+-trees. Nodes of the CSB+-tree are stored contiguously and therefore only the pointer to the first child node needs to be stored in the parent. This pointer elimination technique was extended to spatial data structures in the paper [Ross et al., 2001] which introduced cost-based unbalanced R-trees (CUR-trees). The cost factors of the cache behaviour of a given architecture can be modeled into a cost function and a CUR-tree is built which is optimized with respect to the cost function and a given query model. Prefetch instructions in combination with multiple-size nodes can further assist to achieve better cache behaviour [Chen et al., 2001]. Other methods improve the space utilization by compressing the entries in R-tree nodes to get wider trees [Kim et al., 2001]. The values describing the discriminator or minimum bounding box (MBB) in R-trees are represented relatively to its parent and the number of bits per value is reduced by mapping the values to a coarser representation. The problem of cache-sensitivity of data structures with complex keys is addressed in the paper [Bohannon et al., 2001] which uses partial keys of fixed size.

In this paper, we will investigate how we can make better use of in-memory spatial tree nodes by eliminating unnecessary information in the discriminator and storing more descriptive information than the minimum bounding boxes traditionally used to approximate the spatial objects.

In-memory R-trees typically have very small node sizes, between 3 and 7 entries, because the natural node size is determined by the size of a cacheline. We observed that for such R-trees a substantial part of the information is stored multiple times at different levels in the tree. Very often, entries in the nodes share at least one of their sides with the enclosing parent bounding box.

We propose the use of partial information in R-tree nodes and show how this can improve space utilization and performance of R-trees. We introduce the concept of *partial R-trees* where information is handed down from parent to children to eliminate multiple storing of values. Only values which add information about the entry which is not already given in the parent are stored in the node.

We also investigate how more descriptive information than the standard minimum bounding box can be stored while not using up additional space compared to the standard R-tree. The *partial static O-tree* is based on the O-tree [Sitzmann and Stuckey, 2000], but stores at most four values only, thus not increasing the fanout of the tree while providing better discriminator information and thus improving the search.

The structure of the paper is as follows. Section 2 briefly describes R-trees and talks about their in-memory use. We present partial R-trees in Section 3, and describe how redundant information can be eliminated and how this affects insertion and search. Section 4 describes how to store more descriptive information about the entries in partial O-trees. We present experimental results in Section 5 before concluding our paper in Section 6 with a summary and an outlook to future work in Section 7.

2 Using R-trees in main memory

2.1 The R-tree structure

The R-tree and its variants [Guttman, 1984, Beckmann et al., 1990] is a data structure for n-dimensional data. It has originally been designed to index disk-based data.

An *R-tree node* t has a number of subtrees $t.n$, and for each $1 \leq i \leq t.n$ a discriminator or minimum bounding box $t[i].d$ (which is an array of four side values), and a pointer $t[i].t$. The pointer points at an object identifier if the node is a leaf node. Otherwise it points at another R-tree node. R-trees are built with a maximum number of entries per node M and a minimum number of entries per node $m \leq M/2$.

Further rules which determine the shape of the R-tree are:

1. All leaf nodes appear on the same level.
2. Every node which is not the root node contains between m and M entries.
3. The root node has at least two entries unless it is a leaf node.

If insertion of an entry in a node results in an overfull node, the node is split. Splitting algorithms with linear and quadratic complexity have been presented in the literature [Guttman, 1984, Beckmann et al., 1990].

2.2 Differences between disk and in-memory R-trees

The concept of the R-tree to organize spatial objects in a balanced search tree by using minimum bounding boxes (MBBs) as discriminators works well in the disk-based context. The I/O cost of R-trees is directly dependent on the number of block accesses for systems with small main-memory, and is very competitive compared to other spatial access methods. This approach can also be transfered to in-memory use of R-trees and works similarly well with respect to the cache-behaviour of R-trees, whose performance depends on the number of cache misses occurring. Thus, the strength of the R-tree concept is the same for disk-based and in-memory use.

Looking at the size of the R-tree nodes, we observe that, as expected, the number of entries per node in the in-memory case is significantly smaller. In the disk-based case with a page size of 4 KBytes, we can fit 170 entries into one node of a regular R-tree and 255 if we use the CSB+-technique to eliminate pointers assuming 4 byte long numbers. For in-memory R-trees using a cacheline of 64 bytes, a normal R-tree node using 4 bytes per number can only store 3 entries. The CSB+-pointer elimination technique does

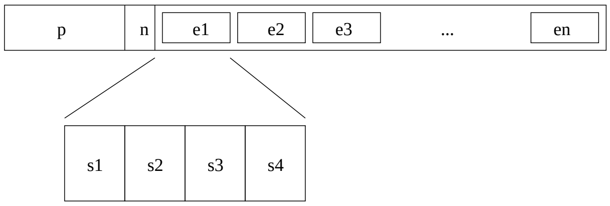


Figure 1: Memory layout of a CSB+-R-tree node

not increase the fanout of the tree in this case. Assuming 2 bytes per number, we can fit 5 entries, and with the CSB+-technique 7 entries per node.

Figure 1 shows the memory layout of a CSB+-R-tree node. A node consists of the pointer p to the contiguous array of child nodes, a counter n for the number of entries and entries e_1 to e_n. Each entry e_i consists of four sides s_1 to s_4. As the child nodes of a CSB+-R-tree node t are stored contiguously, we can refer to the i^{th} child node of t as $t.(p+i)$, using the base pointer $t.p$ and adding the offset i. In the conventional R-tree notation, this corresponds to $t.t[i]$, the i^{th} child node of a conventional R-tree node.

Analyzing the characteristics of small R-tree nodes, we can make some observations which are quite different to the large R-tree nodes used for disk-based data:

- the discriminator MBBs are usually very small and entries share sides with the enclosing MBB
- increasing the fanout of small nodes has an immediate effect on the height of the tree ($\lceil log_3(10^6) \rceil = 13$ and $\lceil log_4(10^6) \rceil = 10$)

We will investigate whether we can improve performance of in-memory R-trees by making use of the two properties stated above.

3 Eliminating multiply stored information

3.1 Analyzing R-trees with small nodes

The idea for partial R-trees was motivated by a small experiment. Using simple datasets of lines and polygons (see Section 5), we counted how many discriminators in an R-tree share one or more sides with their parents when using nodes of small sizes. The results in Table 1 are for an R-tree with 3 entries per node.

File	Sides shared with parent (in %)					Avg.
	1	2	3	4	0	sides
l10000	30.14	49.50	13.38	0.01	6.80	2.30
l50000	31.20	48.03	14.47	0.01	6.15	2.28
l100000	30.93	48.35	14.55	0.01	6.03	2.28
p10000	27.35	29.67	18.72	7.40	16.79	2.27
p50000	27.74	28.80	10.09	7.73	16.61	2.26
p100000	27.79	20.04	19.12	7.75	16.27	2.25

Table 1: MBB sides shared with parent MBB

On average, only 2.25 to 2.3 sides of a discriminator have to be stored per entry as shown in the last column. For all files, at least 83 per cent of discriminators share at least one side with their parent. We will propose a method where the information from the parent discriminator is used to construct the complete box and show that this increases the fanout of the R-tree nodes, resulting in better search performance.

3.2 A more compact R-tree representation

We propose storing only those sides of a discriminator which the entry does not have in common with its parent. Consider the objects depicted in Figure 2.

Figure 2: Objects in an R-tree node

The dashed line represents the minimum bounding box of a node that contains the three objects $O1$, $O2$, and $O3$. As $O1$ shares the left and upper side with the parent, we only need to store the right and bottom side. For $O2$, only the left side needs to be stored as the other three sides are given by the parent discriminator. All four sides for $O3$ must be stored in the node as the entry shares no side with the parent.

The structure of an R-tree node with partial information will therefore be more dynamic than the traditional R-tree node structure. Instead of storing four sides for every discriminator, we store between 0 and 4 sides per entry. The structure is shown in Figure 3.

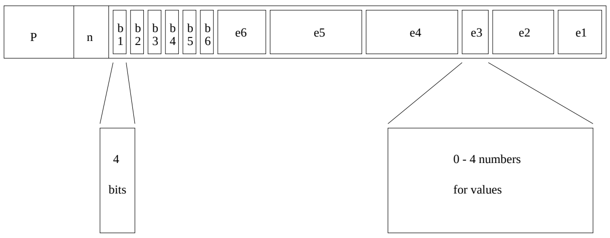


Figure 3: The node structure of a partial R-tree

The partial R-tree node contains a pointer p and a counter n for the number of entries in the node. The first part of the node then contains a sequence of 4-bit fields. The bitfields correspond to the entries in ascending order. Each bitfield indicates which sides are stored in the node and which sides have to be inherited from the parent. The actual sides are stored in reverse order starting from the end of the node. This allows us to make full use of the space of the R-tree node and to minimize copy and comparison operations during insertion and search.

As Table 1 shows, we only store about 2.25 sides per discriminator. Assuming that a side is 4 bytes large, we save $1.75 \times 4 = 7$ bytes, which corresponds to 56 bits. Using 4 bits per entry for extra information, we still save 52 bits per entry.

A *partial R-tree node* s has a base pointer $s.p$, a number of subtrees $s.n$, and for each $1 \leq i \leq s.n$ a 4-bit array $s[i].b$ indicating which sides differ from the parent discriminator. The remainder of the node is an array of side values $s.s$ where $s.s[j]$ is the j^{th} side value in the node (corresponding to the side of the j^{th} true bit occurring in the bit fields). Here we ignore the fact that this array is stored in reverse order. We assume $s.l$ gives the number of sides stored in the partial R-tree node. Note this is not actually stored in the node, since it can be calculated from the number of *true* bits in the 4-bit fields.

3.3 Creating the partial representation

Given an R-tree node t with parent discriminator pd we can create a corresponding partial R-tree node s by mapping each entry in t to its partial information using the algorithm ToPartial. We ignore the pointer information, which is effectively unchanged.

```
ToPartial(t, pd)
   s.p := t.p
   s.n := t.n
   s.l := 0
   for i := 1 to t.n
      for j := 0 to 3
         if (pd[j] = t[i].d[j])
            s[i].b[j] := false
         else
            s[i].b[j] := true
            s.s[s.l] := t[i].d[j]
            s.l := s.l + 1
   return s
```

For each discriminator $t[i].d$ occuring in the R-tree node we check each side j versus the parent discriminator. If it is different to the corresponding side in the parent, we set the corresponding bit $s[i].b[j]$ store the side in the next available space in the partial R-tree node s. The $s.l$ keeps track of how many sides are stored in total.

3.4 Extracting information from a partial R-tree

To read a discriminator in a partial R-tree node, we need to combine the information from the parent discriminator with the information stored in the node. The algorithm ToComplete converts a partial R-tree node s together with its parent discriminator pd to a (total) R-tree node. Again we ignore the pointer information.

```
ToComplete(s, pd)
    t.p := s.p
    t.n := s.n
    l := 0
    for i := 1 to s.n
        t[i].d := pd
        for j := 0 to 3
            if (s[i].b[j])
                t[i].d[j] := s.s[l]
                l := l + 1
    return t
```

After setting the number of subtrees appropriately, the count of sides is initialized, and then each bit field is examined in turn. The discriminator is initialized to the parent discriminator, and then for each different side (makred by a true bit), the next side value is copied in.

The R-tree node t returned from ToComplete for partial R-tree node s is identical to the R-tree node given to ToPartial to construct s. No information is lost. It is important to note that for search operations we do not need to restore the entries completely but can rather use the partial information to perform search as we will describe below.

3.5 Building partial R-trees

The insertion procedure of an object in a partial R-tree requires some extra steps compared to the standard R-tree insertion. The Algorithm Insert describes the insertion of an object o with discriminator e in a partial R-tree s with discriminator pd. The algorithm effectively first converts each partial R-tree node visited to its corresponding (total) R-tree node, performs the insertion and then converts back to a partial R-tree node (or nodes).

```
Insert(s, pd, e, o)
  case s
  external:
    t := ToComplete(s,pd)
    t.n := t.n + 1; t.(p + t.n) := o; t[t.n].d := e
    s := ToPartial(t, pd ∪ e)
    if (Full(s))
      (sl, sr, dl, dr) := Split(s, pd ∪ e)
    else
      (sl, sr, dl, dr) := (s, null, pd ∪ e, {})
    return (sl, sr, dl, dr)
  internal:
    t := ToComplete(s, pd)
    i := ChooseSubtree(t, e)
    (sl, sr, dl, dr) := Insert(s.p + i, s[i].d ∪ e, e, o)
    %% replace t[i] by tl and tr
    t.p + i := sl; t[i].d := dl
    if (sr ≠ null)
      t.n := t.n + 1; t.(p + t.n) := sr; t[t.n].d := dr
    s := ToPartial(t, pd ∪ e)
    if (Full(s))
      (sl, sr, dl, dr) := Split(s, pd ∪ e)
    else
      (sl, sr, dl, dr) := (s, null, pd ∪ e, {})
    return (sl, sr, dl, dr)
```

When inserting an object into an external node, we convert the node to a complete node and simply add the entry naively. The new parent discriminator is $pd \cup e$, that is the minimal bounding box that includes both pd and e. As the representation of the node depends on the parent discriminator, the representation of entries other than the new entry may have changed and might need to be recalculated. For example, an entry previously sharing 2 sides with the parent might now only share 1 side with the new, larger parent discriminator. We convert the expanded (total) R-tree node back to a partial R-tree node. Full then tests whether the new representation is too large to fit into a node. A partial R-tree node s is full if $s.n \times 4 + s.l \times sidebits$ is too great to fit in the node (together with the bits for $s.n$ and the pointer).

If the new representation of the node is too large for the node, either because there was no space for the new entry, or its addition caused other entries to no longer fit, the node is split. We use the linear and quadratic splitting algorithms proposed in the literature [Guttman, 1984, Beckmann et al., 1990]. The Split algorithm split the node s into two nodes containint the same entries that will each fit in the available space. The changes are then passed back to the parent node.

In internal nodes, we first select the best subtree for insertion. We choose the subtree whose discriminator shows the smallest increase in area after insertion of the new entry. Insertion is then continued in the selected subtree. The results of the insertion on the lower level are then used to recalculate the new representation of the node. Note that we need to recalculate the representation of the node not only when the node on the lower has been split but also when the parent discriminator has changed. The new representation might be larger after the parent discriminator has been enlarged. This means, that we might have to split a node on an internal level, even if no new entry was added to it by insertion lower in the tree. We check if a split is necessary after the new representation has been determined and propagate the changes to the parent level.

The algorithms shown convert partial R-tree nodes to R-tree nodes and back again for most processing for ease of explanation, in the implementation most conversion of discriminators is avoided.

3.6 Searching partial R-trees

When searching a partial R-tree, we do not need to reconstruct the complete entry before we can determine whether search should continue in a child tree or not. Instead, we only need to compare the stored sides of an entry with the query as we know that the inherited sides match the query.

This is clarified by Figure 4. Imagine we have determined that query box q intersects with the parent discriminator pd of some entry with discriminator e. In order to determine this we have checked that the box q does not lie completely above pd, to the left of pd, etc. These comparisons are represented by the four dashed arrows. In order to determine that q intersects e we do not need to consider whether it lies completely below e or to the right since these comparisons were already made with the parent box pd. Hence the only two comparisons required are with respect to the sides of e not shared with the parent.

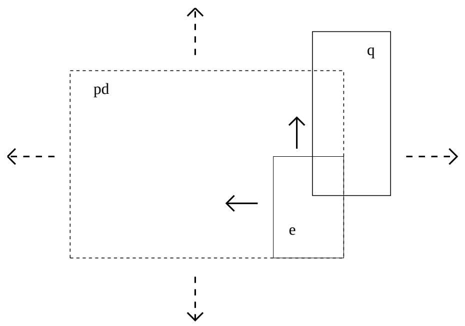


Figure 4: Reducing comparisons with partial R-trees

We search an R-tree node by simultaneously scanning through the bitfield at the start of the node and the entries stored at the back of the node. We store the query sides in such a way that we can detect a mismatch of entry and query with at most four comparisons.

```
Search(s, q)
    l := 0
    for i := 1 to s.n
        match := true
        for j := 0 to 3
            if (s[i].b[j])
                if (s.s[l++] ≤ q.d[i]))
                    match := false; break
        if (match)
            case (s)
            external: Output(s.p + i)
            internal: Search(s.p + i, q)
```

For each entry in an internal node, we check whether the stored sides clash with the query. We use the bit fields to determine which sides stored in $s.s$ refer to which discriminator sides We know that all the sides not stored in the node are the same as in the parent discriminator and therefore match the query. If the entry matches the query, we continue search on the level below. Otherwise, we consider the next entry. For each entry in an external node, we compare all stored sides with the query and output the entry if it matches the query.

The search algorithm shows that the redundancy in a normal R-tree not only occurs when storing sides, but also when doing comparisons. By having partial R-trees we reduce the number of comparisons during search at the same time as we reduce the number of sides stored in an entry.

4 Storing better information

In the partial R-tree, we try to improve the performance of R-trees by eliminating redundant information, thus creating more room for other entries which

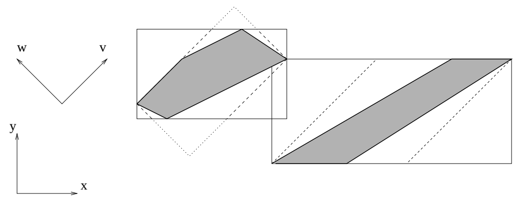


Figure 5: Approximation of polygons in a R-tree and O-tree

results in an increased fan-out of the node. Alternatively, we can try to improve the search behaviour of the tree by using the gained space to store more descriptive information than the standard minimum bounding box. This will help to filter out unsuccessful search paths at a higher level in the tree. In the paper [Sitzmann and Stuckey, 2000], we introduced the O-tree, a constraint-based data structure, which stores an orthogonal box in addition to the standard bounding box to give a better description of the objects in the tree.

4.1 The O-tree approach

The structure of the O-tree is similar to the R-tree. The difference is that we store two minimum bounding boxes per entry: the conventional MBB and an additional MBB along axes v and w which leave the origin at an angle of $\pi/4$ to the x and y axes. The values of v and w can be obtained as $v = (x + y)/\sqrt{2}$ and $w = (y - x)/\sqrt{2}$. Thus, an object in an O-tree is described by eight values, representing the lower and upper bounds on the four axes x, y, v and w. We can store an O-tree discriminator in an array of eight side values" the lower and upper bound of the x axis are stored in sides 0 and 1, sides 2 and 3 refer to the y axis, sides 4 and 5 to the v axis and sides 6 and 7 to the w axis.

As shown in Figure 5, the standard MBB, depicted with solid lines, is a very poor representation for some kinds of data. Although, the two shaded polygons are far from intersecting each other, an intersection test based on the MBBs indicates an overlap. More information about the object is given if we describe the object with an additional orthogonal box (with dashed lines). Using the intersection of both boxes, the lack of overlap of the two polygons is clear.

The O-tree representation is particularly useful for line data. Figure 6 compares the area of the discriminator in an R-tree and O-tree for line data.

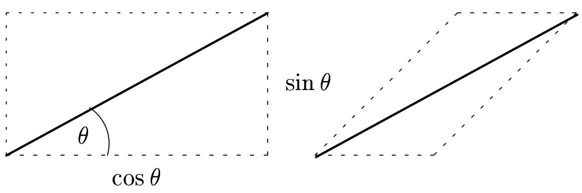


Figure 6: Size of the O-tree bounding box and the R-tree bounding box for line data

When storing a 2-d unit length line at an angle $0 \leq$

$\theta \leq \pi/8$ to the horizontal, the area of the bounding box is $\cos(\theta)\sin(\theta)$. In an O-tree, on the other hand, the area of the intersection of the bounding boxes is $\cos(\theta)\sin(\theta) - \sin^2(\theta)$. This means that the O-tree region bounding a line is on average $2 - \frac{\pi}{2} \approx 0.43$ times the area of the R-tree minimum bounding box.

In our experiments, we found that O-trees indeed improve the accuracy of the search significantly. But the disadvantage of O-trees became also apparent: as we store eight numbers per entry instead of four, the fanout of the tree is significantly reduced. Therefore, the overall search performance could only be slightly improved for line data intersection queries.

The O-tree as first presented is impractical for in-memory use. For a typical 64 bytes cacheline, we can only fit 2 entries in a node.

In this paper, we transfer the O-tree approach to in-memory data structures and try to overcome the weakness of the O-tree by storing only the four most descriptive sides of the O-tree and combine this information with information obtained from the parent discriminator.

4.2 A compact O-tree representation

Although eliminating shared sides also reduces the number of sides per discriminator in an O-tree, on average, we still need to store more than four sides per discriminator. The fanout of the O-tree is therefore still smaller than in the standard R-tree. We therefore apply another technique and eliminate those side of the O-tree discriminators that add no or little information about the objects they describe. The partial O-tree is based on the complete O-tree representation of a discriminator, but only the four most descriptive sides are selected for storage in the entry. Less than four sides can be selected if the discriminator shares more than four sides with its parent. The node structure is similar to the partial R-tree representation.

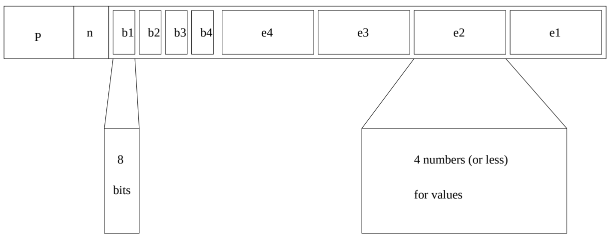


Figure 7: The node structure of a partial O-tree

The node contains a pointer p to the first of the child nodes and a counter n. The bitfields form the first part of the remaining node, but in the partial O-tree case contain 8 bits each, indicating which sides of the entry are stored in the node. The sides are stored again in descending order at the back of the node. In most cases, we will store 4 sides per entry, but for some nodes, more information will be shared with the parent and the number of sides can be further reduced. Compared to the R-tree, we have slightly more space overhead with one additional byte per entry used as the bit vector, but we have almost halved the space requirements of the complete O-tree.

4.3 Selecting the most descriptive data

A partial O-tree node is created by starting off with the complete O-tree node and for each enrty repeatedly discarding the least useful information until the representation contains at most 4 sides. The algorithm OToPartial is the equivalent for O-trees of ToPartial.

```
OToPartial(t, pd)
  s.p := t.p
  s.n := t.n
  s.l := 0
  for i := 1 to t.n
    for j := 0 to 7
      s[i].b[j] := true
      if (pd[j] = t[i].d[j])
        s[i].b[j] := false
    while (|s[i].b| > 4)
      s[i].b := EliminateSide(pd, t[i].d, s[i].b)
    for j := 0 to 7
      if (s[i].b[j])
        s.s[s.l] := t[i].d[j]
        s.l := s.l + 1
  return s
```

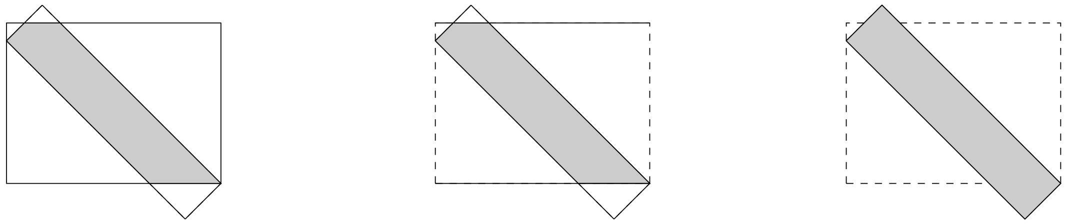
For each O-tree discriminator d We first eliminate the sides of d which are shared with its parent by setting their bits in $s[i].b$ to $false$. As long as this bit field (considered as a set) contains more than 4 sides, we repeatedly call EliminateSide to delete the side with the least importance, i.e., information, from the set of sides. Once we have reduced the set to at most four sides, we store the sides in the node and set the bitfield.

EliminateSide chooses the side to eliminate which causes the least increase in area of the discriminator. The area of the discriminator is the intersection of the MBB and the orthogonal MBB. We discard the side for which the discriminator shows the least increase in area if the side is eliminated.

Figure **??** shows an example for the elimination of sides. The depicted minimum bounding box and orthogonal minimum bounding box describe an object or a group of objects. In part (a) of the figure, all sides of both boxes are stored and their information describes the area shaded grey. (Stored sides are shown as solid lines). We can see that the lower and upper bounds of x do not add any extra information, as these bounds are given by the mobb already. If we eliminate them, as shown in part (b), we still describe the same shaded area. Furthermore, we can observe that the lower and upper bounds of y add only little information about the object. Eliminating these from the discriminator results in an area which is slightly larger. but still a tight approximation of the object.

4.4 Reconstructing the O-tree representation

For search or dynamic insertion, we might want to reconstruct the complete O-tree representation. Opposed to the partial R-tree, during the conversion from the complete O-tree representation to the partial O-tree representation, information can get lost. We



(a) (b) (c)

Figure 8: Eliminating sides of an O-tree discriminator

only store an approximation of the original O-tree discriminator, therefore the partial O-tree representation will be less accurate than the complete O-tree representation, while still more descriptive than the R-tree information.

Algorithm OToComplete converts a partial O-tree node back into a complete O-tree node, i.e., an entry with eight sides. Although the entry is complete, it is only an approximation of the original complete O-tree representation. As for the partial R-tree, we start with copying the parent discriminator. We then replace the values of the sides stored with their actual values. The function tighten propagates the tightens the O-tree discriminator. The resulting discriminator is an approximation of the original O-tree entry.

```
OToComplete(s, pd)
    t.p := s.p
    t.n := s.n
    l := 0
    for i := 1 to s.n
        d := pd
        for j := 0 to 7
            if (s[i].b[j])
                d[j] := s.s[l]
                l := l + 1
        t[i].d := Tighten(d)
    return t
```

Although we do not store all information about the O-tree discriminator, we may be able to reconstruct some more information from the sides we have stored. The MBB on axes x and y also gives bounds on axes v and w and vice versa. We can thus tighten these bounds. Figure 9 illustrates tightening on a MBB for x and y and MBB for v and w (MOBB). The MBB for x and y determines a tighter lower bound for v and w, while the MBB for v and w determines a tighter upper bound for y. The tighter bounds are illustrated by dashed lines.

The function tighten takes the discriminator with the sides that have been read from the node and the parent.

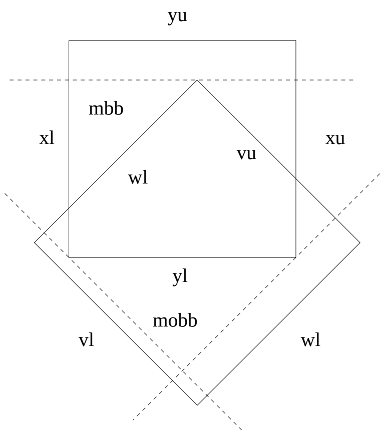


Figure 9: Tightening the representation of an O-tree discriminator

```
Tighten(d)
    d[0] := max(d[0], (d[4] − d[7])/√2)
    d[1] := min(d[1], (d[5] − d[6])/√2)
    d[2] := max(d[2], (d[4] + d[6])/√2)
    d[3] := min(d[3], (d[5] + d[7])/√2)
    d[4] := max(d[4], (d[0] + d[2])/√2)
    d[5] := min(d[5], (d[1] + d[3])/√2)
    d[6] := max(d[6], (d[2] − d[1])/√2)
    d[7] := min(d[7], (d[3] − d[0])/√2)
return d
```

Boundaries of x and y based on the values of v and w of the orthogonal box are computed. The original values are replaced if the computed bounds are tighter. The same process is then performed for the orthogonal MBB.

4.5 Insertion in dynamic partial O-trees

The insertion of entries in a partial O-tree is identical to the insertion for partial R-trees except for the function GetRepresentation. For both trees, the new representation of a node is based on the previous representation stored in the tree. In the partial R-tree, this representation is a complete and accurate representation of the entry. In the partial O-tree, this representation is already an approximation of an entry. Creating a new partial O-tree representation will therefore produce another approximation of an already approximated entry. The quality of the representation of an entry therefore deteriorates every time a representation has to be changed and is recalculated. During dynamic insertion this happens very frequently. The accuracy of object description in the partial dynamic O-tree is therefore very poor and leads to inaccurate search involving high cost. Partial dynamic O-trees therefore seem not useful in practice, although the very first approximation is a very accurate description of entries which provides more information than the R-tree bounding box.

We therefore suggest to use the partial O-tree approach in a static environment. The partial static O-tree is generated from an existing O-tree and every discriminator in the tree is converted only once into the partial O-tree representation. The result is a partial O-tree which takes up only about half the space of the original O-tree but represents the entries in a more descriptive way than the R-tree.

4.6 Building a static partial O-tree

A static partial O-tree is created from a complete O-tree. We can determine the fanout of a partial O-tree for a given node size. A complete O-tree with that fanout is created. The nodes in the complete O-trees are about twice as large as the ones in the partial O-tree. We then read the complete O-tree and convert every discriminator into the partial representation by applying the StoreSides algorithm. The result is a partial O-tree with nodes that fit in the given node size and with the fanout of the complete (larger) O-tree. As we only had to apply the approximation step once, the discriminators in the tree are very accurate.

4.7 Searching partial O-trees

We can search partial O-trees in two different ways which are different in cost and accuracy. The AccurateSearch shown below reads every entry in the tree and tries to reconstruct its complete O-tree representation using ReadEntry.

```
AccurateSearch(s, pd, q)
    t := OToComplete(s,pd)
    for i := 1 to s.n
        if intersect(t[i].d, q))
            case (s)
            external: Output(s.p + i)
            internal: AccurateSearch(s.p + i, t[i].d, q)
```

The search algorithms traverses the tree, reconstructs the complete representation of the entries and outputs the leafs whose approximations intersect with the approximation of the query object.

The alternative is a faster, but less accurate search algorithm similar to the partial R-tree search. The entries are not completely reconstructed, but only the sides stored are compared with the query. This reduces the search time as no copying of the parent node, copying of entry sides and tightening have to be performed. FastSearch is exactly analogous to Search.

```
FastSearch(s, e)
    l := 0
    for i := 1 to s.n
        match := true
        for j := 0 to 7
            if (s[i].b[j])
                if (s.s[l++] ≤ q.d[i]))
                    match := false; break
        if (match)
            case (s)
            external: Output(s.p + i)
            internal: FastSearch(s.p + i, q)
```

Both search algorithms only implement the filter step which creates a set of candidate objects which might actually intersect the query. The refinement step takes each candidate object in a subsequent step and checks its intersection with the query object.

5 Experiments

5.1 Description of Experiments

Our experiments were conducted on a Sun Ultra-5 with 270 MHz and 256 MByte RAM under Solaris 2.6. A level-2 cache line on our architecture is 64 bytes. We implemented the partial R-tree and partial O-tree using the CSB+-technique as described in Sections 3 and 4. We compared the partial R-trees to a normal R-tree which also uses the pointer elimination technique. For a 64 byte cacheline, traditional O-trees cannot be used as the node can only contain 2 entries. We therefore do not include results of the normal O-tree in our graphs and discussion. For all trees, We used the quadratic splitting algorithm for all trees.

Our test data consists of a set of randomly constructed line and polygon data relations. Each line data set contains a number of lines each with approximate length 20 in a square of area 5000. The polygon data sets contain convex polygons with up to 10 nodes and edges of length approximately 40 in a square area of 10000. The polygons are constructed by randomly creating the 10 points and using the Graham scan algorithm to calculate their convex hull.

Our experiments measure the performance for partial R-trees and partial static O-trees and compare them with the search performance of the R-tree. For each test case, we queried each tree with 10,000 random queries and measured the number of node accesses, search time, number of results and search times including the refinement step.

5.2 Experimental Results

Figures 10 and 11 show the average number of node accesses per query for line data and polygon data, respectively. Node accesses can be used as a rough measure for cache misses, i.e., it is the worst case number of cache misses. For the line data, the partial R-trees show a reduction of 30 per cent in the number of node accesses compared to the R-tree. The accurate partial O-tree reduces the number of node accesses by 60 per

lines	R-tree	Partial R-tree	Partial O-tree Accurate	Partial O-tree Fast
100	26	21	25	26
500	124	88	80	88
1000	221	159	139	152
5000	1010	724	474	513
10000	1954	1403	869	933
50000	9194	6444	4010	4295
100000	18193	12701	7613	8105

Figure 10: Average node accesses for a line data query

polys	R-tree	Partial R-tree	Partial O-tree Accurate	Partial O-tree Fast
100	56	48	55	56
500	279	209	264	266
1000	559	407	549	554
5000	2711	1982	2708	2732
10000	5305	3923	5303	5354
50000	25182	18640	25610	25863
100000	51492	38728	51021	51516

Figure 11: Average node accesses for a polygon data query

cent. The fast search on the partial O-tree still has 55 per cent less node accesses than the R-tree. For polygon data, the partial R-tree shows a similar improvement of 25 per cent. The partial O-trees, on the other hand, only reduce the number of node accesses by a marginal 1 per cent.

Measuring the search time per query for line and polygon data, we obtained the results shown in Figures 12 and 13. For line data, the partial R-tree improves on the normal R-tree by about 25 per cent. The partial O-tree with accurate search shows how expensive the accurate search is: although it could decrease the number of node accesses significantly, the gained time is used to perform expensive operations. The search time of the partial accurate O-tree is 60 per cent higher than for the R-tree. The less accurate fast O-tree search algorithm, on the other hand, reduces the search time by up to 35 per cent. This shows clearly that, although the fast search is slightly less accurate and will search more subtrees, its faster execution time results in the best performance. For polygon data, the partial R-tree reduces search time by up to 25 per cent compared to the

lines	R-tree	Partial R-tree	Partial O-tree Accurate	Partial O-tree Fast
100	0.08	0.06	0.22	0.09
500	0.17	0.17	0.64	0.26
1000	0.33	0.29	1.12	0.42
5000	1.92	1.51	3.84	1.49
10000	4.11	3.20	7.03	2.79
50000	19.74	14.48	32.58	13.06
100000	38.72	28.37	61.64	24.59

Figure 12: Average search time for line data

polys	R-tree	Partial R-tree	Partial O-tree Accurate	Partial O-tree Fast
100	0.11	0.10	0.43	0.15
500	0.31	0.37	2.09	0.68
1000	0.72	0.75	4.22	1.44
5000	5.21	4.07	21.36	7.51
10000	10.89	8.50	42.37	15.19
50000	54.10	40.30	203.94	74.29
100000	110.58	84.10	405.51	147.84

Figure 13: Average search time for polygon data

lines	(Partial) R-tree	Partial O-tree Accurate	Partial O-tree Fast	O-tree
100	232	85	110	85
500	1175	418	516	416
1000	2283	805	953	805
5000	11751	4155	4498	4155
10000	23417	8286	8809	8282
50000	116440	41177	43223	41131
100000	233360	82657	85969	82581

Figure 14: Total number of results for line data (in thousands)

normal R-tree. The partial O-trees do not improve on the R-tree. Again, the accurate O-tree search suffers from the high computation cost and shows an increase of 250 per cent. Even the fast O-tree search shows a slight increase in search time. While showing a great improvement for line data, the additional orthogonal bounding box of O-trees does not seem to help to make search on polygon data more efficient.

Nevertheless, as shown in Figures 14 and 15, it still reduces the number of hits in the filter step significantly. We have included the number of results for a complete O-tree with the same fanout to have a measure of the best possible reduction in results.

Compared to the R-tree and partial R-tree, which both have the same number of hits, the complete O-tree reduces the number of results by 65 per cent for line data and 25 per cent for polygon data. The accurate search partial O-tree still achieves a 64 per cent reduction for line data and 23 per cent for polygon data. The fast O-tree search is slightly less accurate, but can still show a reduction of 64 per cent for line data and 20 per cent for polygon data.

polys	(Partial) R-tree	Partial O-tree Accurate	Partial O-tree Fast	O-tree
100	595	467	486	456
500	2833	2192	2287	2155
1000	5745	4422	4583	4371
5000	28474	22057	22875	21725
10000	56938	44033	45608	43419
49999	283970	218826	225952	216042
100000	567204	436057	449977	430840

Figure 15: Total number of results for polygon data (in thousands)

We can now compare the search time of the search trees if we take the refinement step into account. The refinement step takes the set of results obtained by searching the tree in the filter step and tests for actual intersection of the query object and the object in the candidate result set. The search times for this experiment are shown in Figures 16 and 17.

	R-tree	Partial	Partial O-tree	
lines		R-tree	Accurate	Fast
100	0.10	0.09	0.25	0.14
500	0.31	0.31	0.79	0.42
1000	0.66	0.66	1.36	0.72
5000	4.22	3.47	4.85	2.63
10000	8.67	7.06	9.11	5.10
50000	43.17	34.46	43.30	24.02
100000	87.13	68.72	81.69	45.80

Figure 16: Average search time for line data including refinement time

	R-tree	Partial	Partial O-tree	
polys		R-tree	Accurate	Fast
100	0.27	0.26	0.62	0.35
500	1.12	1.17	2.86	1.53
1000	2.48	2.40	5.86	3.20
5000	14.91	13.15	30.08	16.85
10000	30.38	26.78	59.47	33.50
50000	153.34	132.38	290.43	163.11
100000	311.89	267.86	574.49	324.38

Figure 17: Average search time for polygon data including refinement time

As the number of results does not change, the improvement of the partial R-tree compared to the normal R-tree is similar for search including the refinement step. For the O-trees, on the other hand, the relative performance to the R-tree has changed. The accurate search O-tree now shows a slight improvement over the R-tree for the large line data file and similar performance for the other line data sets. The fast search O-tree now shows a reduction in search time of up to 60 per cent, again for the larger files. For polygon data, even the reduced number of candidates for the refinement step does make the partial O-tree competitive. For the accurate search, the search time now is 60 higher than for the R-tree and still no improvement is shown for the fast O-tree search on polygon data.

6 Summary

We investigated how to make better use of the space in a small R-tree node for in-memory applications. As many entries share sides with their parents, we introduced the partial R-tree which only stores information that is not given by the parent node. Our experiments showed that the partial R-tree shows better performance than the R-tree for random line queries on line and polygon data. The improvements range from 10 to 30 per cent. This is due to a higher fanout of the node which showed to be 4 compared to 3 in the normal R-tree. We also investigated if we could make better use of the space by storing different information that promises to yield a better approximation of the entry. The partial O-tree is based on the O-tree but stores only the most important part of the information of an O-tree box. We implemented a static version of the partial O-tree and investigated two search algorithms. The fast search algorithm still shows enough accuracy and showed improvements of 35 per cent for line data without refinement step and 60 per cent improvement for line data with refinement step. For polygon data, search could not be improved, but the static partial O-tree still showed stable performance similar to the R-tree.

7 Future Work

We will investigate if we can further improve the fanout of the tree by storing the bitfields encoded using the Huffman-encoding. Furthermore, we will also experiment with static partial O-trees in a disk-based database environment.

References

[Beckmann et al., 1990] Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. (1990). The R∗-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331.

[Bernstein et al., 1998] Bernstein, P. et al. (1998). The Asilomar Report in Database Research. *SIGMOD Record*, 27(4):74–80.

[Bohannon et al., 2001] Bohannon, P., McIlroy, P., and Rastogi, R. (2001). Main-Memory Index Structures with Fixed-Size Partial Keys. In *Procs. of ACM SIGMOD Conference.*

[Chen et al., 2001] Chen, S., Gibbons, P. B., and Mowry, T. C. (2001). Improving Index Performance Through Prefetching. In *Procs. of ACM SIGMOD Conference.*

[Guttman, 1984] Guttman, A. (1984). R-trees: a Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57.

[Kim et al., 2001] Kim, K., Cha, S. K., and Kwon, K. (2001). Optimizing Multidimensional Index Trees for Main Memory Access. In *Procs. of ACM SIGMOD Conference.*

[Rao and Ross, 2000] Rao, J. and Ross, K. A. (2000). Making B^+-Trees Cache Conscious in Main Memory. In *Procs. of ACM SIGMOD Conference*, pages 475–486.

[Ross et al., 2001] Ross, K. A., Sitzmann, I., and Stuckey, P. J. (2001). Cost-based Unbalanced R-trees. In *Procs. of SSDBM Conference*.

[Sitzmann and Stuckey, 2000] Sitzmann, I. and Stuckey, P. J. (2000). O-trees: a Constraint-based Index Structure. In *Proc. ADC 2000*, volume 22:2 of *Australian Computer Science Communications*, pages 127–134.