Using B^+-Trees in a Two Disk-Single Processor Architecture to Efficiently Process Inclusion Spatial Queries

Mario A. Nascimento *Margaret H. Dunham*

Relatório Técnico IC–97-23

Novembro de 1997

Using B$^+$-Trees in a Two Disk-Single Processor Architecture to Efficiently Process Inclusion Spatial Queries*

Mario A. Nascimento† Margaret H. Dunham‡

Abstract

In this paper we address the problem of indexing spatial data, in particular two dimensional rectangles. We propose an approach which uses two B$^+$-trees, each of them indexing the projected sides of the given rectangles. The approach, which we name 2dMAP21, can also be easily parallelized using two disks – but still a single processor – each holding the trees indexing the projected sides on either axes. We focus on queries of the type "find all rectangles included within another (reference) rectangle". Nevertheless, 2dMAP21 can processe other types of queries as well. We compare our approach to the R*-tree, known as the most efficient R-tree derivative. Our investigation shows that, if the queries have the same spatial distribution of the data, the non-parallel 2dMAP21 may be a competitive alternative to the R*-tree in some cases, whereas the parallelized version of 2dMAP21 outperforms that structure virtually always. 2dMAP21 may consume a little more or less storage space than the R*-tree, depending primarily on the spatial distribution on the indexed MBRs. The use of B$^+$-trees renders our approach to be actually implementable using commercial DBMSs.

1 Introduction

The indexing of multidimensional data is needed in many application domains, e.g., spatial databases and geographical information systems [Sam90]. One widely used way of modeling such type of data is via n-dimensional minimum bounding rectangles (MBR). A two dimensional MBR is the smallest rectangle in which we can fit the two dimensional object being modeled, where such rectangles have sides parallel to the X and Y axes.

Indexing MBRs have received quite some attention in the literature. Among the structures proposed to deal with this problem, the R-tree [Gut84] is certainly the most popular one. The R-tree uses a framework which is similar to the B$^+$-tree, in the sense that it is

*A shorter version of this paper was published and presented at the V ACM Intl. Workshop on Geographical Information Systems, Las Vegas, USA, Nov/97. Research initiated at Southern Methodist Univeristy and further developed as part of GEOTEC's effort, a ProTem-CC project sponsored by the Brazilian National Council of Research (CNPq).

†Invited lecturer at the Institute of Computing, State University of Campinas, Brazil, mario@dcc.unicamp.br. Also a researcher at the Brazilian Agency for Agricultural Research, Campinas, Brazil, mario@cnptia.embrapa.br.

‡Associate Professor at the Southern Methodist University, Dallas, USA, mhd@seas.smu.edu.

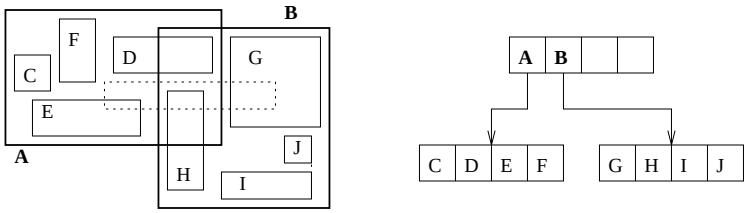


Figure 1: A set of MBRs and the resulting R-tree.

paged, balanced, and has the leaf nodes (at the same level) pointing to the actual data records. Non leaf nodes point to either leaf nodes or represent a super-MBR which contain other super-MBRs or MBRs. Figure 1 (adapted from [SRF87]) shows an example set of MBRs (C, D, ..., J) and super-MBRs (A and B) and the resulting R-tree. (For now ignore the dotted rectangle.)

More efficient variants of the R-tree, such as the R^+-tree [SRF87] and the R^*-tree [BKSS90], have been proposed. The R^+-tree aims to reduce the overlap among the super-MBRs by clipping the MBRs. The trade-off is that some "clipped" MBRs will now appear in more than one super-MBR and this will cause the index size to be greater than the R-tree's. The R^*-tree makes use of a more carefully designed policy for splitting a node, using the concept of deferred split and forced re-insertion. Results in [BKSS90] have shown the R^*-tree to be overall the most efficient R-tree derivative, and as such, we will be using it to compare our approach later in the paper.

Other structures/approaches have been proposed for indexing and querying spatial data. Next we briefly review some of them. The interested reader is referred to Samet's book [Sam90], which surveys several others.

Some work has been done regarding parallel R-trees [KF92]. Even though that research has shown that parallel R-trees may yield much better performance than a non-parallel R-tree, it is likely that a number (more than two) of parallel R-trees would be needed to provide performance comparable to a single (non-parallel) R^*-tree. As we shall see later in the paper our approach requires only two disks to outperform the R^*-tree. To our knowledge no research has been done on parallelizing the R^+-tree or the R^*-tree.

In [FR91] the authors present an approach (named DOT) based on fractal functions which is also capable of transforming MBRs in two or higher dimensions to points in one-dimensional space. Thus DOT (as well as our approach) uses a B^+-tree to index MBRs. Although DOT is shown to outperform the classical R-tree [Gut84] it is doubtful whether it can outperform the R^*-tree (given the results in [FR91]). The issue of parallelizing DOT's approach was not investigated.

The use of B^+-trees to index MBRs has also been suggested in [TP95]. However, in that paper, the authors propose the use of four B^+-trees, each one indexing one of the corner coordinates that determine the MBRs. We propose the use of only two such trees instead. Moreover, the idea of parallelizing access to the the host B^+-trees was not touched upon in [TP95].

Unfortunately, the R-tree, as well as the R^*-tree, is not very "smart" to process "inclusion" type of queries. In an inclusion query, one is interested in finding all MBRs totally

included within a reference MBR. Next we sketch the R-tree's algorithms for this particular type of query[1]. We use R to denote the root of the R-tree indexing all MBRs, Θ to be the reference MBR used in the query. All entries in the nodes are super-MBRs (if internal nodes) or actual MBRs (if leaf nodes). Finally, assume that each entry in a super-MBR points to either a super-MBR or MBR r.

Algorithm 1 *P = Rtree-Inc(R, Θ)*

1. For each entry r in node R
   (a) If R is an internal node and r intersects Θ then
   P = Rtree-Inc(r, Θ)
   (b) If R is a leaf node and r is included in Θ then
   $P = P \cup r$

For an illustration of the algorithm's behavior refer to Figure 1 where the dotted rectangle represents the reference MBR Θ. When processing the query the algorithm notices that both A and B intersect with Θ. Therefore both sub-R-trees rooted at A and B are traversed. When inspecting the leaf entries at A and B it is easy to see that no MBR is actually included in Θ. That is, even though the answer was empty, the entire tree was searched. Although the illustration is quite simple, it is not hard to see that sets of large MBRs will also imply large super-MBRs, thus an increased amount of overlap among them, and consequently among them and the query MBR. This ultimately will lead to a high number of sub-R-trees being traversed, potentially uselessly.

Note that, in addition to such shortcoming, the R-tree (as well as its derivatives) is quite a specialized data structure, not available in most DBMSs.

The contribution of this paper is to address both issues raised above. Firstly, we address the problem of indexing MBRs subject to a more common framework, namely B^+-trees (e.g., [EN94, Chapter 5]). We thus aim at allowing traditional (i.e., existing) DBMSs to index spatial data. Secondly, centering our discussion on the inclusion type of query, we provide a simple yet competitive way (when compared to the R*-tree) of using two B^+-trees (possibly but not necessarily hosted under distinct disks) to process such type of queries.

In our previous work [ND97] we designed an indexing approach, built on top of a B^+-tree, which allows the indexing of temporal ranges. Such an approach, named MAP21, is based on mapping the two end points of a range to a single point and using this as the indexing value for the range. We have shown that under simple assumptions MAP21 can process efficiently several types of queries.

To extend MAP21 to index MBRs we represent the MBR by the projection of its sides onto the X and Y axes. We may then use two B^+-trees (possibly hosted in distinct disks) to index such projections, which are equivalent to temporal ranges. Processing a query is a matter of returning the intersection of the answer provided by the processing of both trees. The main advantage of such an approach, which we call 2dMAP21, is that to index and query MBRs, all one needs is two B^+-trees, available in virtually any commercial DBMS.

[1] Note that the algorithm is exactly the same for an R*-tree.

This paper extends the work presented in [ND96] in several ways. In [ND96] we presented preliminary investigations on the 2dMAP21, which showed it to be superior to Guttman's R-tree. In this paper we compare 2dMAP21 to the R*-tree, recognized by the related literature at large as the most efficient R-tree derivative. In [ND96] we investigated the intersection query. For that type of query however, 2dMAP21 needs to know the upper bound for the length of the indexed ranges in either axis. In fact 2dMAP21's performance did depend on such upper bound, the largest it was, the slower the query processing time. As we shall see shortly, processing inclusion queries (which is the focus of this paper) do not depend on such upper bound. We also derive a simple formula for 2dMAP21's expected query processing time. Finally, in [ND96] only synthetic data sets were used, in this paper we explore 2dMAP21's performance using real data sets.

In the next Section we present a review of the MAP21 approach (for a thorough introduction we refer the reader to [ND97]). In Section 3, we present the 2dMAP21 approach, focusing on the inclusion type of query (notice that others can be processed as well). Section 4 presents a performance analysis, using both synthetic and real data. We conclude the paper in Section 5.

2 Review of the MAP21 Approach

MAP21 maps the two end points of a range $R^k = [R_s^k, R_e^k]$ into a single value and uses this one as an indexing value for the range. We assume the following:

- α is the maximum number of digits needed to represent any range end value and thus
- The starting and ending points of an "indexable" range are: (a) Non-negative integer values and (b) Upper-bounded. I.e., $R^k = [R_s^k, R_e^k] \Rightarrow 0 \leq R_s^k \leq R_e^k \leq 10^\alpha - 1$.

Therefore, any range $R^k = [R_s^k, R_e^k]$ can be indexed using a unique value provided by the mapping function: $\Phi(R^k) = \Phi(R_s^k, R_e^k) = R_s^k 10^\alpha + R_e^k$ Note that, in practical terms, what the function $\Phi(R^k)$ does is to "left-shift" the start of the range. It is straightforward to obtain the original range given its mapping, namely: $V^k = [\Phi_s^{-1}(\Phi(V^k)), \Phi_e^{-1}(\Phi(V^k))]$ where: $\Phi_s^{-1}(\Phi(V^k)) = (\Phi(V^k) - \Phi(V^k)\%10^\alpha)/10^\alpha$, $\Phi_e^{-1}(\Phi(V^k)) = \Phi(V^k)\%10^\alpha$ and % is the traditional remainder operator. In addition, the following holds [ND97]:

Proposition 1 *The above defined function $\Phi(.)$ maps distinct ranges into distinct points. Given $R^k = [R_s^k, R_e^k]$ and $R^l = [R_s^l, R_e^l]$ then $\Phi(R^k) = \Phi(R^l) \Leftrightarrow R_s^k = R_s^l$ and $R_e^k = R_e^l$.*

Proposition 2 *The ordered points in the resulting index represent a lexicographical order of the ranges. Given $R^k = [R_s^k, R_e^k]$, $R^l = [R_s^l, R_e^l]$, and $\Phi(V)$ as defined above, $R_s^k < R_s^l \Rightarrow \Phi(R^k) < \Phi(R^l)$; and if $R_s^k = R_s^l$ then $R_e^k < R_e^l \Rightarrow \Phi(R^k) < \Phi(R^l)$.*

We finally define:

Definition 1 *A MAP21 tree is a B^+-tree indexing point values, which represent ranges, and were created using the $\Phi(.)$ function described above.*

Due to the underlying B$^+$-tree structure, MAP21 uses $O(N_r)$ space and requires $O(\log_B N_r)$ I/Os to process and update (deletions included), where B is the page size and N_r is the number of indexed ranges. Furthermore it inherits all previous research results devoted to B$^+$-trees, such as concurrency and recovery [JS93]. It is noteworthy pointing out that not much research has been done regarding concurrency and recovery of R-trees in general.

2.1 Processing the Inclusion Query

Using the MAP21 tree one can process several types of ranges based queries [ND97]. However, as we are primarily interested in an approach that will allow us to process the rectangle inclusion problem, we review the case of the inclusion query only. Proposition 1 will be important as it guarantees that each range appears once and only once in the indexing tree. Proposition 2 ensures that the search in the tree can be done in one-pass. The algorithms we present show how to collect all pointers needed to actually access the data records, hence we are not describing how to access the data records themselves (that was also the case with the R-tree algorithm presented earlier).

The inclusion query takes as input a range $[Q_s, Q_e]$ and returns the pointers associated to records which have a range $R = [R_s, R_e]$, such that $Q_s \leq R_s \leq R_e \leq Q_e$. In what follows we use the following notation: $\lceil\lceil T\rceil\rceil$ ($\lfloor\lfloor T\rfloor\rfloor$) is the smallest (greatest) indexed value greater (smaller) than or equal to T.

Lemma 1 *Given that the indexed ranges are in lexigraphical order in the leaf nodes of the MAP21 tree, to find all ranges that are contained in $[Q_s, Q_e]$ one needs only to scan the ranges between ranges $[Q_s, Q_s]$ and $[Q_e, Q_e]$.* ([ND97]).

We use the above Lemma to derive the following algorithm to answer an inclusion query (where we denote the MAP21 tree by M):

Algorithm 2 *P = MAP21-Inc(M, Q_s, Q_e)*

1. $P \leftarrow \emptyset$
2. Traverse M to the leaf entry indexing $\lceil\lceil Q_s 10^\alpha + Q_s\rceil\rceil$
3. Scan all the leaf entries forward
   (a) If the entry value $\Phi(R^k)$ is such that $\Phi_e^{-1}(\Phi(R^k)) \leq Q_e$ then
   $P = P \cup$ {pointers associated to this entry}
4. Until the leaf entry indexing $\lfloor\lfloor Q_e 10^\alpha + Q_e\rfloor\rfloor$.
5. Return P

Notice that the algorithm may read ranges that do not belong to the answer, but no further overhead is imposed when the actual data records in the answer are retrieved. "Useless" data records are filtered out of the algorithm output and thus only data records belonging to the actual response will be accessed. Nonetheless, similar shortcoming also occurs in the R-tree and the R*-tree. As we shall see in the performance analysis, the R*-tree's performance degenerates faster with the query MBR's size than 2dMAP21's.

2.2 Expected Performance

In this section we to derive the expected number of I/Os yielded when processing inclusion queries using the MAP21 approach.

Let us first assume a B$^+$-tree with nodes that can fit B values (and thus $B+1$ pointers), a query range $Q = [Q_s, Q_e]$ and that all ranges are located in a "maximal" range $[0, L_{max}]$. At each point in the line $[0, L_{max}]$ there is a number of ranges starting on that very point. Given that we have N such ranges we have that $\psi = N/(L_{max}+1)$ is thus the average number of ranges starting at any point in the maximal range $[0, L_{max}]$.

Due to the linear scan in the leaf nodes, for each and every point in the range $[Q_s, Q_e - 1]$ an average of ψ ranges (which are actually points in the B$^+$-tree leaves) are inspected. Note that the range $[Q_e, Q_e]$ is the last one inspected and by construction the only range starting at Q_e inspected by the algorithm. Therefore, denoting $L_q = Q_e - Q_s$, the algorithm above inspects $(L_q - 1)\psi + 1$ ranges, or mapped ranges, i.e., points linearly ordered in the index.

We must now translate this into number of tree nodes (disk blocks) accessed. In general, a B$^+$-tree node with capacity to hold B values actually holds $B \ln 2$ values [Yao78], due to the way nodes are split and merged. Therefore, in average, all index points to be inspected imply $((L_q - 1)\psi + 1)/(B \ln 2)$ I/Os, i.e., tree nodes or disk blocks accessed.

Finally, if N is the number of indexed values (i.e., mapped ranges) then $\log_B N$ suffices to traverse such a tree from its root downward to a leaf node.

Hence, $\log_B N + ((L_q - 1)\psi + 1)/(B \ln 2)$ is the expected number of I/Os needed to perform an inclusion query of length L_q using MAP21. Later in Section 4, we shall verify that such expected value is accurate. It is interesting to note that, unlike we verified in [ND96], the length of the indexed ranges does not play any role in query processing time.

3 The 2dMAP21 Approach

Considering that a MBR θ, with sides parallel to the X and Y axes and determined by its lower left and upper right corners (denoted by $(\theta_{xl}, \theta_{yl})$ and $(\theta_{xu}, \theta_{yu})$ respectively), may be uniquely determined by the projection of both of its sides, onto the X and Y axes, i.e., $\Gamma_{\theta x} = [\theta_{xl}, \theta_{xu}]$ and $\Gamma_{\theta y} = [\theta_{yl}, \theta_{yu}]$, let us now discuss how the MAP21 approach can be extended to index two dimensional MBRs.

One tree is used to index each axis, we thus set up two MAP21 trees, M_x and M_y, before any MBR is input. Once an MBR θ is input, we compute its X and Y projections. $\Gamma_{\theta x}$ and $\Gamma_{\theta y}$. We then use the mapping function $\Phi(.)$ defined earlier to compute the indexing value of both projections, i.e., $\Phi(\Gamma_{\theta x})$, which is input into M_x, and $\Phi(\Gamma_{\theta y})$, which is input into M_y. Figure 2 shows an example of how the projections are obtained while Figure 3 shows the resulting MAP21 trees (internal nodes are omitted for simplicity). Note that there is no need to duplicate the actual data records.

Extending the algorithm *MAP21-Inc* to search for rectangle inclusions is simple. The heart of the algorithm is that if a reference MBR Θ includes another MBR θ then Θ's projections, on both axes, must also include θ's projections, on both axes, as well. Before presenting the algorithm let us use the following notation: Θ is the reference MBR (i.e., the one with which we want to find all others that are included in) with lower left and upper

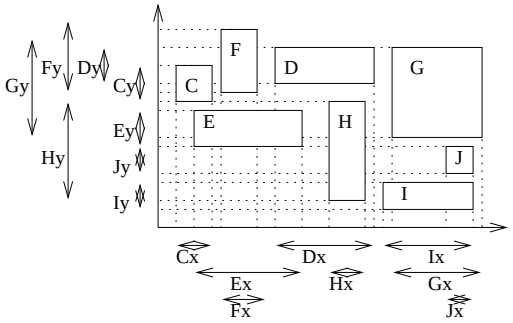


Figure 2: The MBRs of Figure 1 and its projections.

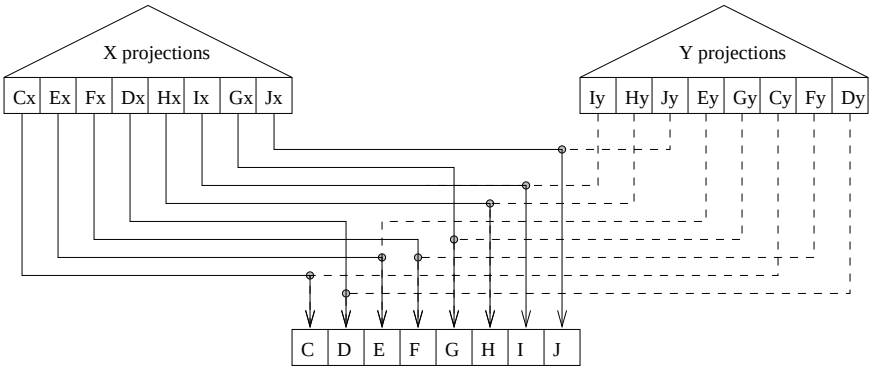


Figure 3: The MAP21 trees indexing the projections in Figure 2.

right corners $(\Theta_{xl}, \Theta_{yl})$ and $(\Theta_{xu}, \Theta_{yu})$ and M_x and M_y are the MAP21 trees indexing the MBR projections on axes X and Y.

Algorithm 3 *P = 2dMAP21-Inc(M_x, M_y, Θ)*

1. $P_x = P_x \cup$ *MAP21-Inc*$(M_x, \Theta_{xl}, \Theta_{xu})$
2. $P_y = P_y \cup$ *MAP21-Inc*$(M_y, \Theta_{yl}, \Theta_{yu})$
3. RETURN $P = P_x \cap P_y$

It is important to note that M_x and M_y are rather independent. This allows us to host them under distinct disks, and thus update and, more importantly, search them in parallel. Notice however, that we do not require more than one processor, i.e., all we need to parallelize 2dMAP21 is to place each index in a different disk, so that they can be searched in parallel. We assume that the last step in the algorithm, i.e., computing $P_x \cap P_y$, can be done in main memory. Note that we are dealing with pointer values only, which are rather small data types. In the worst case, we assume a relatively small number of I/Os should suffice in most cases to compute such intersection.

4 Performance Analysis

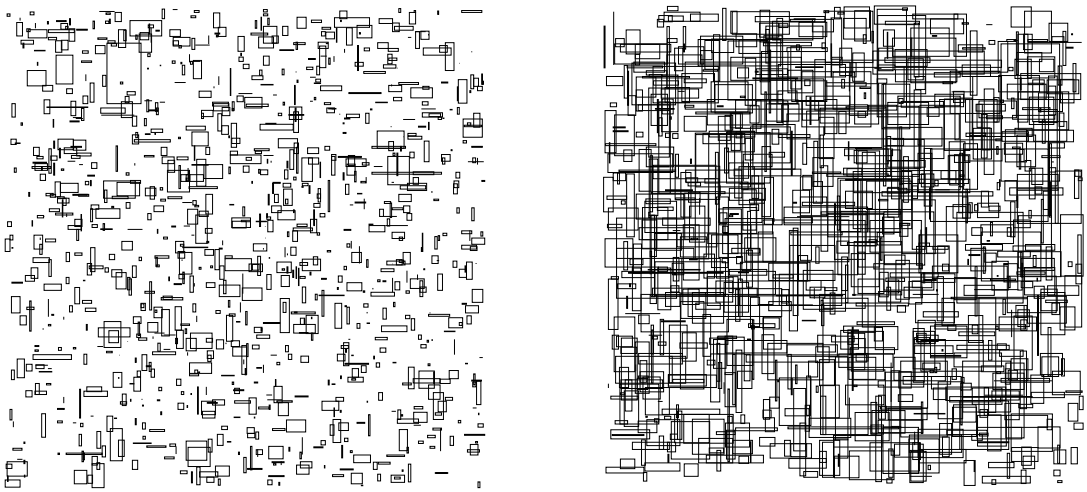
To validate the 2dMAP21 approach we have actually implemented and compared it against an implementation of the R*-tree (as described in [BKSS90]). We first evaluate the structures' performance with respect to synthetic data and then real data sets.

For both cases the disk block size (i.e., the size of a node in the tree) was set to 1,024 bytes. Recall that a B$^+$-tree node must hold B indexing values plus $B + 1$ pointers. The data type used to represent the non-negative indexed values uses 4 bytes, and so do the pointers (to data or other nodes). This yields $4B + 4(B + 1) = 1,024$, hence (a rounded) $B = 127$.

We simulated the parallel implementation by neglecting any CPU overhead and taking the maximum number of I/Os yielded by any single tree (i.e., worst performance) as the overall performance indicator. Recall that we assume that the intersection of the sets returned by each call to *MAP21-Inc* can be processed in main memory (i.e., without any further I/O).

4.1 Indexing Synthetic Data

Similarly to the analysis conducted in [P$^+$95], we use three sets MBRs: small, medium and large ones. The areas of the MBRs in each set average, respectively, 0.02%, 0.1% and 0.2% of the total area. As [TP95, P$^+$95, KSCL95, FR91] have done, we populate each set with 10,000 MBRs. Notice that this implies that the sum of the areas covered by each set averages 2, 10 and 20 times as big as the total area. This allows us to appreciate the performance of the indexing structures with respect to the overlap ratio rather than only



(a) Small MBRs. (b) Large MBRs.

Figure 4: Sample MBRs from the synthetic data sets.

with respect to the size of the indexing set. For a better idea of the data sets Figures 4.1(a) and (b) depict sample of the data sets containing small and large sized MBRs.

We assume that the indexed MBRs are uniformly spread over the map of interest. The data (area size) has an exponential distribution, which we believe to be more realistic than a uniform distribution. The queries, i.e., the reference MBRs, were also generated using the same type of distributions and sizes. The results show the average number of I/Os required to process 250 queries, using each of the described query area sizes.

The formula derived in Section 2.2 applies for each tree, that is, for each set of projected ranges. The expected number of I/Os when both trees are hosted in the same disk, and thus accessed serially, can be obtained by doubling the expected number for one disk.

Figures 5, 6 and 7 show how 2dMAP21 and the R*-tree perform when indexing MBRs with a given average MBR area and varying query area. 2dMAP21-P denotes the results obtained by querying the two underlying MAP21 trees in parallel.

The first observation we can draw is that the R*-tree performance degrades much faster than 2dMAP21's with the increase in the size of the indexed MBRs. This can be explained by the fact that the larger the indexed MBRs the larger the overlap ratio among them and eventually among them and the query MBR. This ultimately leads to a large number of "false hits", i.e., sub-R-trees that must be traversed without collaborating with the final answer.

Careful observation reveals that for a fixed size of MBRs, 2dMAP21's performance suffers a little more than R*-tree's with the increase of the query's area. That is, 2dMAP21's curve is a little more steep than 2dMAP21-P's and the R*-tree's. This suggests that 2dMAP21 is more sensitive, although not much more, to the query area than the R*-tree. For instance, for a set of small MBRs (Figure 5), we note note that when varying the average size of the indexed MBRs from small to large this results in 2dMAP21 (which is non-parallelized) being worst at the beginning and better at the end.

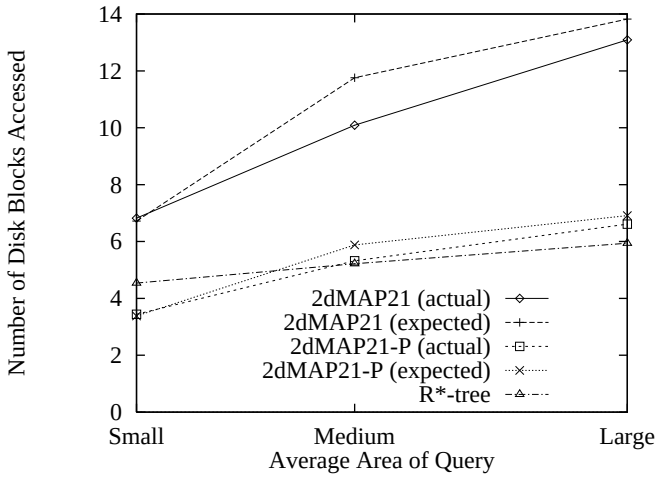


Figure 5: Query processing when indexing Small MBRs.

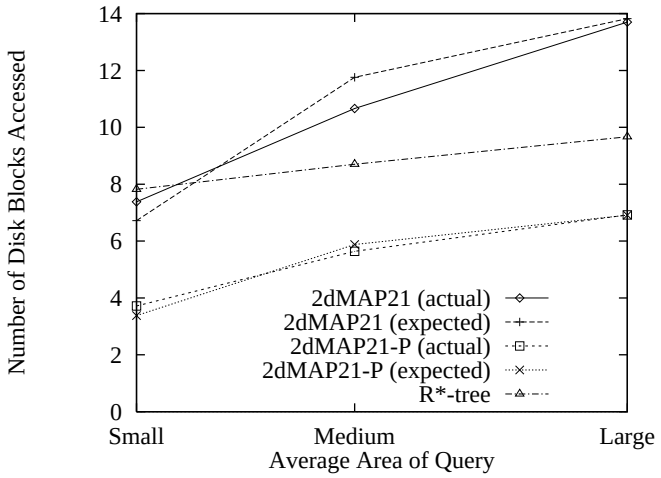


Figure 6: Query processing when indexing Medium MBRs.

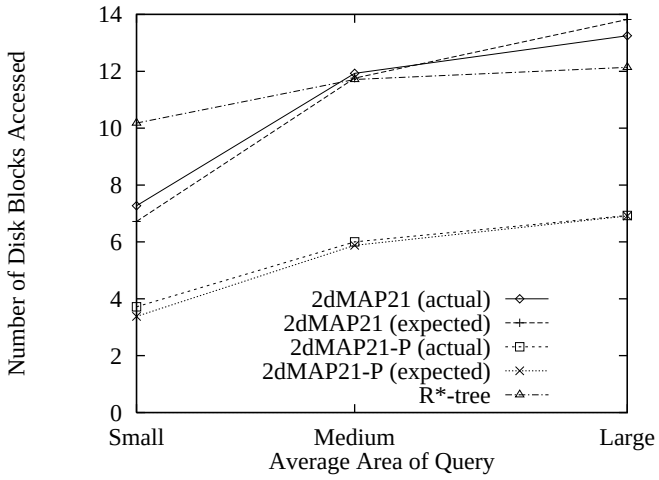


Figure 7: Query processing when indexing Large MBRs.

From Figure 5 we see that the R*-tree is faster than 2dMAP21 (45% in average), whereas it was basically comparable to 2dMAP21-P (it ranged from being 25% slower to 10% faster). When indexing medium MBRs (Figure 6), 2dMAP21-P averaged 50% less processing time than the R*-tree. While for small queries both 2dMAP21 and the R*-tree delivered nearly the same performance, the R*-tree was 30% faster for large queries. Finally, for the case of Figure 7 (indexing large MBRs), 2dMAP21-P was again 50% faster, in average, than the R*-tree. The R*-tree average performance was close to 2dMAP21's (with little advantage to the latter), R*-tree's performance ranged from 30% slower (small queries) to 10% faster (large queries).

Even though not shown we have observed that increasing the number of indexed MBRs the relative performance among the investigated structures has not changed substantially. That is, 2dMAP21-P yields faster query processing time virtually always, whereas 2dMAP21 is quite competitive for indexing a set of large MBRs. We also verified that the expected number of I/Os for 2dMAP21-P, is rather accurate.

Note that as the areas of indexed MBRs increase so does the gap between 2dMAP21-P's performance and the R*-tree. Thus, even if the intersection between P_x and P_y (see Algorithm *2dMAP21-Inc*) cannot be performed in main memory there is some "performance gain" that could be spent with disk I/Os to compute such intersection and still yield 2dMAP21-P faster than the R*-tree. An exception to such observation is the case of indexing small MBRs.

Taking these issues into consideration we believe that 2dMAP21 is a competitive alternative to the R*-tree. We have verified (but not shown in this paper) that 2dMAP21 outperforms, by a rather large margin, the classical R-tree. We also conjecture that 2dMAP21 can outperform the R^+-tree as well.

In addition, and unlike the majority of alternatives to index spatial data, 2dMAP21 needs nothing else than commonly available B^+-trees and some sort of embedded SQL (e.g., [EN94, Chapter 7]) to be functional.

The total size of the MAP21 trees used by the 2dMAP21 approach is about a 25% larger than R*-tree. 2dMAP21's size ranged from 201 to 228 tree (disk) nodes while R*-tree's sizes varied little around 172 tree nodes.

It is important to mention that we did take into account that the data type used to hold MAP21's mapped range is twice as big a data type needed to hold a single end-of-range (i.e., a coordinate) value under the R-tree.. Consider that b bytes are needed to hold any coordinate value. The R-trees, must keep four coordinates per MBR, thus using $4b$ bytes. 2dMAP21 must keep two projections (in both axes), each consuming $2b$ bytes, hence using a total of $4b$ bytes as well[2]. Therefore, using a larger data type is not a shortcoming of MAP21.

4.2 Indexing Real Data

We close this section analysing the performance of the investigated structures when indexing real data sets. For that we used two data sets, named MG and LB. MG and LB contain about 40,000 and 53,000 MBRs respectively, representing actual roads somewhere in the USA. A sample of both data sets are shown respectively in Figures 8(a) and (b). Those MBRs are not uniformly distributed. We again generated 250 small, medium and large queries, with the same relative area sizes we used earlier and still being uniformily spread over the total area. The results are shown in in Table 1.

In this case, 2dMAP21-P, which had been thus far the best performer, was nearly 100% slower than the R*-tree. This can be explained as follows. As the data MBRs are not uniformly spread there may be "holes" in the total area of interest where no MBR exists. Consequently when a query MBR falls into such holes, the R*-tree can quickly determine that no data MBR can be included in that one and return an empty answer without going further down in the tree. 2dMAP21, on the other hand does not have a spatial view of the data, but rather two one-dimensional views, which are not related, as fas as the 2dMAP21 approach goes. Hence, using such views, 2dMAP21 will investigate where each of them could possibly contribute to the query's answer, and only after each view has been completely processed would the actual answer (possibly empty) be found.

We have then experimented to use query MBRs with the same non-uniform spatial distribution as the indexed data set. As the MBRs in the real data sets have a non-trivial spatial distribution we used 500 randomly chosen MBRs from the data sets themselves as the query MBRs. The results are shown in Table 2. In that case we again have 2dMAP21-P yielding smaller query processing time than the R*-tree. We therefore conclude that in order for 2dMAP21 be competitive we should be able to avoid querying holes in the data space. If one assumes that the user queries data he/she knows, querying holes may not be very likely.

It was interesting to note that for the real data sets, the two resulting MAP21 trees are overall smaller than the R*-tree, consuming respectively 517 and 882 disk nodes for the MG data set and 673 and 1054 for the LB data set. This happens because many of the projections of the data MBRs coincide and as such are indexed only once. The R*-tree cannot take advantage of such "coincidences", unless the MBRs are identical.

[2] For our experiments using $b = 2$ sufficed.

(a) MG's MBRs. (b) LB's MBRs.

Figure 8: Sample MBRs from the real data sets.

Table 1: Average query processing time when using real data sets and queries uniformily distributed.

MG data set			
Query Size	R*-tree	2dMAP21-P	2dMAP21
Small	3.44	7.16	13.10
Medium	5.72	12.42	23.20
Large	7.22	15.57	30.00
LB data set			
Query Size	R*-tree	2dMAP21-P	2dMAP21
Small	3.44	7.73	15.56
Medium	5.07	13.76	17.69
Large	8.20	17.88	39.28

Table 2: Average query processing time when using real data sets and queries extracted from the same data set.

Data set	R*-tree	2dMAP21-P	2dMAP21
MG	4.28	3.87	7.60
LB	4.20	4.04	7.85

5 Conclusions

Access structures for spatial data in general and R-trees in particular are not widely available in commercial DBMSs, despite being well known. Our goal in this paper was to address this gap by using a framework widely available on existing DBMSs, namely B^+-trees. Our proposed approach, named 2dMAP21, is based on two standard (and potentially parallel) B^+-trees, each indexing projected sides of MBRs. Even though we have addressed only the inclusion query, 2dMAP21 can process several other types of queries. Its algorithms are straigthforward extensions of those used for 2dMAP21. For some types of queries, however, such as intersection, query performance depends on the length of the longest indexed range. This dependence may been dealt with by partitioning the data set into several disjoint sets [ND97][3]. As we have seen in this paper such dependence does not exist in the case of inclusion queries in particular.

We should stress that presenting a simple indexing approach that outperforms a well known spatial indexing approach, is not the only contribution of this paper. Just as important is the fact that the proposed 2dMAP21 approach can be implemented on top of most existing DBMS facilities in a straigthforward manner.

We have shown that the proposed approach, which we call 2dMAP21, virtually always outperforms the R*-tree in terms of query processing time when the host B^+-trees are accessed in parallel, and in many situations when not. One notable exception happens when the query MBRs have a distribution different than the data MBRs. In such case, 2dMAP21 pays the price of not having a two-dimensional view but rather two one-dimensional views. We therefore conclude that, in order to avoid performance problems due to the lack of 2dMAP21's "global view", it is better to use it when both data and query have the same spatial distribution.

When indexing uniformly spread data 2dMAP21 used about a quarter more storage. As storage is a much less expensive commodity than time in most application domains, we conclude that 2dMAP21 is a very attractive approach for this problem. Nevertheless, when indexing the real data set 2dMAP21 was actually smaller than the R*-tree.

Future efforts should be devoted towards: (1) extending this approach to higher dimensional space – where we believe it can be more advantageous as this would yield a decrease in the R*-tree fan-out and subsequently increase its height; and (2) actually implement the 2dMAP21 approach using embedded SQL, thus allowing a standard DBMS to index and query spatial data.

Acknowledgments

The authors wish to thank: Yannis Theodoridis for the R*-tree source code and comments on an earlier draft of this paper; and Ibrahim Kamel for providing the MG and LB data sets. We acknowledge the use of Jan Janninck's source code for the B^+-tree used as MAP21's framework [Jan95]. We also thank Aalborg University's Dept. of Computer Science (es-

[3] Preliminary studies have shown that as much as 6 parallel disks (3 disks for the partitions on each axis) may be needed to outperform the R*-tree with respect to intersection queries.

pecially Prof. Christian S. Jensen) for the facilities provided during a visit by the first author.

References

[BKSS90] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conferencee on Management of Data*, pages 322–331, Atlantic City, NJ, June 1990.

[EN94] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.

[FR91] C. Faloutsos and Y. Rong. DOT: A spatial access method using fractals. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 152–159, Kobe, Japan, April 1991.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conferencee on Management of Data*, pages 47–57, Jun 1984.

[Jan95] J. Janninck. Implementing deletions in B^+-trees. *ACM SIGMOD Record*, 24(1):6–8, March 1995.

[JS93] T. Jonhson and D. Shasha. The performance of current data structure algorithms. *ACM Transactions on Database Systems*, 18(1):51–101, March 1993.

[KF92] I. Kamel and C. Faloutsos. Parallel R-trees. In *Proceedings of the 1992 ACM SIGMOD International Conferencee on Management of Data*, pages 195–204, San Diego, CA, June 1992.

[KSCL95] M.S. Kim, Y.S. Shin, M.J. Cho, and K.J. Li. A comparative study of spatial access methods. In *Proceedings of the Third ACM International Workshop on Advances in Geographic Information Systems (ACM-GIS'95)*, pages 29–36, Baltimore, MD, December 1995.

[ND96] M.A. Nascimento and M.H. Dunham. Using B^+-trees as a practical alternative to the classical R-tree. In *Proceedings of the 11th Brazilian Symposium on Databases (SBBD'96)*, pages 187–200, São Carlos, Brazil, October 1996. Available at URL http://www.cnptia.embrapa.br/~mario/Papers/tr-96-cse-05.ps.

[ND97] M.A. Nascimento and M. H. Dunham. Indexing valid time databases via B^+-trees – the MAP21 approach. Technical Report CSE-97-08, School of Engineering and Applied Sciences, Southern Methodist University, 1997. Available at URL http://www.cnptia.embrapa.br/~mario/Papers/tr-97-cse-08.ps.

[P+95] D. Papadias et al. Topological relations in the world of minimum bounding rectangles: A study with R-trees. In *Proceedings of the 1995 ACM SIGMOD International Conferencee on Management of Data*, pages 92–103, San Jose, CA, June 1995.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Strucutures.* Addison-Wesley, Reading, MA, 1990.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multidimensional objects. In *Proceedings of the Thirteenth Very Large Databases Conference*, pages 507–518, Brighton, England, September 1987.

[TP95] Y. Theodoridis and D. Papadias. Range queries involving spatial relations: A performance analysis. In *Proceedings of the Second International Conference on Spatial Information Theory (COSIT'95)*, Semmering, Austria, September 1995.

[Yao78] A. Yao. 2-3 trees. *Acta Informatica*, 2(9):159–170, 1978.