External Memory Data Structures for 3 and 4-Sided Queries

Michal Lemczyk
Glenn Hickey
Michael Lawrence

Technical Report CS-2005-17

October 10, 2005

Faculty of Computer Science
6050 University Ave., Halifax, Nova Scotia, B3H 1W5, Canada

External Memory Data Structures for 3 and 4-Sided Queries

Michal Lemczyk Glenn Hickey Michael Lawrence
{lemczyk,hickey,michaell}@cs.dal.ca

Abstract

In this paper, we experimentally evaluate the relative performance of 3 external memory query structures. The R and Priority R (PR) Trees are compared using 4-sided rectangle queries in R^d. For 3-sided point queries in R^2, we compare the two structures mentioned above to the External Priority Search (EPS) Tree. While the PR and EPS Trees possess optimal I/O bounds, they are largely theoretical works. The R-tree, on the other hand, is a simpler heuristic data structure that is commonly used in practise but is inefficient in the worst case. After comparing wall times and I/O efficiency of queries, updates and bulk loads of the three structures on a variety of simulated and real data, we find the R-Tree variants to generally outperform the EPS and PR trees in R^2. In higher dimensions, however, the PR-Tree is most efficient.

1 Introduction

4-sided rectangle query structures are particularly useful for bounding box searches on spatial databases but can also be applied to for 4-sided queries on arbitrary data. Likewise, 3-sided queries on 2D points are fundamental to efficiently answering many kinds of databases searches. Structures to perform these tasks efficiently on large datasets are well-studied in the theoretical literature. The Priority R-Tree (PR-Tree), for instance, is asymptotically optimal in terms of I/O efficiency for 4-sided rectangle queries while the External Priority Search (EPS) Tree is a dynamic optimal data structure for 3-sided queries on 2D points. In practise, however, neither of these structures sees much actual use. Instead, the R-tree and its simple variants are extremely popular despite relying on heuristics and being far from optimal in the worst case. We implemented these three data structures and, in the experiments that follow, investigated whether this trend is justified. Subjects of interest include the identification of "hidden constants" in the theoretical algorithms and the practicality and portability of analysis in terms of I/Os rather than running times.

The following three sections provide an overview of the the R, PR, and EPS-Trees respectively and discuss some practical concerns observed during implementation. The experimental results of the R and PR-Trees on 4-sided rectangle queries on simulated and real data are then discussed in Section 5. All three structures are then compared, along with the B+-Tree, with respect to 3-sided point queries in Section 6.

2 The R-Tree

The R-tree [1] is a fundamental external data structure for spatial access. It is largely based on the B-tree yet differs in that it can support rectangular queries and data in arbitrary dimension whereas the B-tree is limited to 1D points. This distinction does come at a price, however. B-trees are optimal in part because the intervals covered by sibling nodes are *disjoint*, allowing a target leaf to be found by a single walk down the tree. Rectangular data can overlap and therefore the area covered by two children of a given internal node may not be disjoint. This phenomenon is exacerbated when the region of coverage by a node is only minimally occupied – ie, there is a large amount of empty space. A query can, in the worst case where all nodes overlap, traverse the entire tree without returning a single rectangle, yielding a linear worst-case bound. In practise, the outlook is much brighter as heuristics in the insertion, node-splitting and, if applicable, bulk-loading algorithms allow a tree to be created and maintained such that overhead due to traversing nodes that contain no data in the query is minimized.

The structural properties of an R-tree are nearly identical to those of a B-tree. It is height balanced and all data is stored in leaves which lie at the same level. The root node must have at least two children whereas internal nodes and leafs must contain between B/a and B children and data elements respectively. Each internal node contains a bounding box associated with each of its children that is the minimum bounding box for the subtree rooted at that child. Each leaf contains a bounding box for each data element it contains along with a pointer (if necessary) to the element's full representation in the database. Note that although an element can be covered by the bounding rectangles of two different leafs, it is only ever stored under one. The R-tree, similarly to the B-tree, thus requires linear storage.

2.1 R*-Tree Updates

The original R-tree, when confronted with the choice of how to split a node or choose a leaf for insertion, attempts to minimize the area of the affected nodes in the resulting tree. The basis for this heuristic is that having more tightly packed nodes reduces the probability of a query visiting a child without actually intersecting any of its contents, thus wasting an I/O. While this idea seems intuitive, no real proof was given to show that minimizing the area is practically more efficient than other intuitive heuristics such as minimizing node margins (perimeters) or minimizing overlap between nodes. This was the impetus cited by Beckmann et al. [2] behind their proposed R*-tree. This variant of the R-tree retains the structural simplicity of the original, differing only in the update algorithms, but was shown to be practically superior by extensive experiments and quickly became the standard. For these reasons, the R*-tree variant was selected to represent R-tree updates in this paper. The relevant algorithms are highlighted briefly as follows:

Choose Subtree
This routine is invoked at each level of the tree, starting from the root, in order to search for a leaf to update during insertion. In the R-tree, the child is chosen whose bounding box's area would be enlarged the least by the insertion of the new node. The R* algorithm uses this heuristic for every level except for parents of leaf nodes which are instead chosen according to the minimum increase of the overlap between their children and the new node. This enhancement performed terribly in our tests as minimizing overlap in this way results in scattering neighbouring rectangles into different nodes, possibly at opposite ends of the tree. Indeed, it only makes sense when inserting into an already mature R-tree whose leaf nodes are sufficiently deep and already well-packed by area-enlargement. Since insertions were used both for updates and tree creation, this heuristic was disabled in the implementation and *Choose Subtree* operates on area alone. The cost remains one I/O for each level of the tree or $O(\log_B N)$.

Node Split
A node split is triggered by reinsertion (explained later) into a full node. For each axis, the rectangles are sorted by both their lower and upper coordinate values. A linear number of partitions are computed for each sorted list by splitting it in two such that each side of the split has at least B/a elements. The larger a is, the more possible partitions there are. The sum of the margins of all partitions for each axis are computed and the axis with the minimum sum is selected as the split axis. The partition across the split axis with the minimum margin is used to split the node. A single node split costs $O(1)$ I/Os as only the sibling and parent need to be modified but, as usual, it can propagate up the $O(\log_B N)$ levels of the tree.

Deletion
Unlike B-trees, R-trees do not merge underfull nodes. Instead, the node is simply deleted and its children reinserted into the tree. Note that internal nodes deleted in this way must be reinserted into the same level in which they existed previously. Despite its increased cost, this procedure helps account for the fact that the tree is built greedily. As it changes over time, the decisions made in the past by *Choose Subtree* may turn out to be clearly suboptimal. Reinserting nodes from the top provides a chance to better place the nodes to reflect changes in other parts of the tree. This procedure is left unchanged from the R-tree.

Forced Reinsertion
This is the core R* algorithm and basically extends the principle described above to insertion. If a rectangle is inserted into an overfull leaf, or an internal node is created from a split and inserted into an overfull parent, then instead of calling *Node Split*, room is made for the new element by removing a selected number of children from the targeted node. These children are selected as the nodes whose bounding rectangles' centroids are furthest from the centroid of the parent node. The removed nodes are reinserted back to their old level starting from the root. If an overflow is caused by a *re*insertion, *Node Split* is called. The additional cost of reinsertions are justified by the improved query performance of the resulting tree. We have seen that each insertion and deletion can trigger up to $O(B)$ reinsertions. There are no guarantees given that these amounts can be amortized over $O(B)$ updates but the expected behaviour is that they will be.

2.2 Hilbert Bulk Loading

If the input is known in advance, a static bulk loading algorithm that operates on all the data at once should be preferable to greedily inserting each item as described above both in performance and the quality of the resulting tree. Indeed, this is often the case as clustering the input data using a space filling curve has been employed to a high degree of success [4]. A space filling curve visits all points on a grid (whose granularity defines the order of the curve) exactly once and never crosses itself. Such a curve can be used to impose a linear ordering on a set of d-dimensional data. A Hilbert curve [5] is a space filling curve with particularly good clustering properties in that points which are near to each other in the Hilbert ordering are most likely to be neighbours in space. This clustering property leads directly to a bulk loading algorithm: Each rectangle is mapped to a point whose position on the Hilbert curve is computed. The rectangles are then sorted by their Hilbert numbers and a scan is performed to assign $O(B)$ consecutive rectangles in the sorted order into each leaf. The remaining levels of the tree are constructed in the same fashion leading to a $O(SORT(N))$ I/O algorithm overall. The Hilbert number of a 2D rectangle is typically computed as the position of its midpoint on the 2D Hilbert curve. All other information about the rectangle is lost which can be a detriment to packing rectangles of vastly varying size and shape. An alternative, referred to as Hilbert H4 [14] obtains the position of the point $(x_{min}, y_{min}, x_{max}, y_{max})$ on a 4D Hilbert curve and is expected to perform better when there is a high variance in the shapes and sizes of the input.

2.3 Curse of Dimensionality

The R-Tree structure can support rectangles of arbitrary dimension without any modification, the only effect being a reduction of fanout due to the increased size of the input. The efficiency lost with increasing dimensions extends well beyond the reduction of fanout, however, as most of the update and bulk loading algorithms discussed above are affected. For the R* algorithms, the concepts of area and overlap area are the most troubling. Since the area occupied by N uniformly distributed points increases exponentially with d, the resulting trees will tend to be accordingly dominated by empty space. Experiments performed in [8] also show that the percentage of overlapping nodes within the R* tree increases from under 5% to over 90% when d is raised from 2 to 5. Furthermore, there is no reason to expect that splitting nodes across a single split axis will be particularly useful on higher dimensional data where the optimal clustering may lie across several dimensions.

The Hilbert bulk loaders are also affected, since with a larger number of dimensions there comes a larger number of pairs of points which are distant from each other on the curve but next to each other in space [16]. Additionally, and perhaps even more relevant to these experiments, is that the Hilbert numbers generated by the Hilbert library [6] are limited to 64 bits. This directly limits the order of the grid chosen to $64/d$. For example, only 8 bits could be used to represent each axis of a coordinate in 8D. This is a large loss of resolution on our 64 bit input. An attempt was made to reduce the dimensionality of the input using principle components analysis so that the Hilbert number could be computed at a higher resolution but the resulting trees were inferior in every case. A possible explanation is that the uniformly generated data simply does not contain any redundant dimensions. There are better alternatives to the R-tree for high dimensional data in the literature. One example is the X-tree [8] which is an efficient extension of the R-tree but beyond the scope of this report

3 The PR-Tree

The Priority R-Tree, introduced in [14], is proved to be capable of answering a rectangular window query on N rectangles of dimension d, in $O((\frac{N}{B})^{1-\frac{1}{d}} + \frac{T}{B})$ I/Os, where B is the block size, and T is the output size. Furthermore, it is a linear space structure, with the capability to be bulk-loaded in $O(sort(N))$ I/Os. Updates to the PR-Tree may be performed as regular R-Tree updates (requiring $O(\log_B N)$ I/Os), however the query bound may be lost due to the degradation of the structure. In regards to the distinction between the PR-Tree and R-tree, intuitively the PR-Tree utilizes the extremes of a collection of rectangles along each dimension to "describe" the collection for purposes of data division, as opposed to traditional methods which solely utilize the bounding box of the collection.

3.1 Structure and the Pseudo-PR-Tree

The formal definition of the PR-Tree utilizes a building block known as a pseudo-PR-Tree. The following descriptions will assume 2D points are being indexed for ease of understanding.

The pseudo-PR-Tree is defined recursively on the set of input rectangles S:
If $|S| < B$, the pseudo-PR-Tree node is a regular leaf containing all the rectangles in S. Else the node contains four priority leaves, each containing B rectangles with the most extreme edge values for each dimension (min/max pairing), and two children, whose subtrees each roughly contain half of the remaining rectangles in S. The split edge orientation alternates between dimensions on each level of the pseudo-PR-tree.

The PR-Tree is built bottom-up using pseudo-PR-Trees. The lowest level is solely composed of leaves (both priority and regular) of the pseudo-PR-Tree built on the input data set. Thus all the input rectangles are at the lowest level of the PR-Tree, and are stored in blocks. The next level takes as input the bounding boxes of each leaf in the previous level. A pseudo-PR-Tree is built on those rectangles, and the PR-Tree level is built out of the leaves of the pseudo-PR-tree, with each rectangle containing a pointer or offset to the leaf in the lower level which it represents. Thus the fan-out of the tree is B, as a block of rectangles is represented by just the bounding box. This process continues until the input to a level fits within a block.

In regards to extending the pseudo and PR-Trees to higher dimensions, the only modification to be made is that the pseudo-PR-Tree contains more priority leaves (two for each dimension). Since the PR-Tree is composed of bounding boxes of rectangles in a pseudo-PR-Tree arrangement, the dimension has no effect on the structure.

3.2 Construction Implementation

All of our implementations make use of the Transparent Parallel I/O Environment (TPIE) [13]. TPIE allows application programs to access unstructured collections of data, part of which is stored on disk, in a way which abstracts the interface to disk. The most important constructs utilized in TPIE are streams and blocks. A stream is an ordered collection of contiguous elements, allowing access to T sequential elements in $\Theta(T/B)$ I/Os, and random access of an element in $\Theta(1)$ I/Os. A block is an object of size B storing a constant amount of administrative data, a variable number of elements of some fixed sized type, and links to other blocks (represented as IDs), the total size of which is no more than B.

The implementation of the PR-Tree construction algorithm followed the more simple, however less efficient top-down method of directly following the pseudo-PR-Tree and PR-Tree definitions at each node. Although there exists a $O(sort(N))$ bulk-loader for the PR-Tree, it deviates from the structure definition as it constructs the tree splits first and then fills in the priority leaves. Although this detail is insignificant with regards to the theoretical performance (as the kd-tree query analysis bound only requires an approximate even split of the data at each node). It was felt that it may impact the real-world measurements as cases may be constructed where the priority leaves position skews the tree split to a less balanced (and intuitively less optimal) value; thus the more strict method was utilized.

The construction of the PR-Tree is relatively simple, as the actual tree structures may in fact be abstracted away. For example, since only the contents of the pseudo-PR-Tree leaves are pertinent to the PR-Tree, the actual pseudo-PR-Tree structure does not have to exist (as no traversals of it will ever be made).
The PR-Tree construction consists of:

- Reading in a TPIE stream of input rectangles (with each rectangle being flagged as a leaf).
- Constructing a pseudo-PR-Tree on the input rectangles, appending each regular and priority leaf of the tree to a TPIE stream.
- For each block of rectangles in the output stream, constructing a rectangle describing the bounding box of the block, with the offset value being the TPIE stream offset of the start of the block. Append the rectangle to a new stream describing the next layer.
- Constructing a pseudo-PR-Tree on the next layer stream, appending each regular and priority leaf of the tree to a new output TPIE stream.
- Prepending the latest output stream (the pseudo-PR-Tree leaves of the new layer) to the previous layer. This will require a traversal and update of the offset values of the latest layer.
- Continuing processing layers until the number of rectangles in a layer fits within a block. In that case, prepend those rectangles to the final output stream and exit.

It should be noted that the latest and previous layer streams may be cleared and re-used in the loop to minimize storage requirements.

Pseudo-PR-Tree layer

To construct the pseudo-PR-Trees the following steps are performed:

- For each dimension (min/max pairing) i, extract the rectangles destined to the priority leaves:
  - Sort the input rectangles by dimension i.
  - Copy B rectangles from the end of the sorted stream. If the stream size is less than B, insert pad rectangles into the leaf to fill in the remainder, then return.
  - Write the priority leaf to the output stream.
  - Truncate the input stream to delete the copied rectangles.
- If the number of remaining rectangles is 0, exit.
- Else, if the number of remaining rectangles is less than or equal to a block, write them out to the output stream as a leaf, using pad rectangles if needed.
- Else:
  - Sort the remaining rectangles by the current split dimension.
  - Set the split dimension for the next level.
  - Create two new streams of rectangles, with each containing approximately half of the input.
  - Recurse upon each of the two children and their appropriate input streams.

As previously noted, the pseudo-PR-Tree structure is implied. During execution, the priority leaves are extracted and saved to the output stream, and the process continues on the remainder with no effort being made to store the tree structure or even the positions of the leaves within the tree.

3.3 Query Implementation

The query procedure on the PR-Tree is fairly straight forward, it follows:

- Begin at offset 0 in the PR-Tree stream (root).
- Read in a block of rectangles.
- If the block is a leaf, test each rectangle for intersection.
- Else, for each non-pad rectangle in the block test for intersection with the query rectangle.
  - If intersection occurs, store the rectangle's offset value in an array.
- Recurse on each offset value stored in the array (Begin at offset in the PR-Tree stream).

This in effect traverses the PR-Tree from root to leaves, visiting blocks whose bounding boxes intersect the query, and recording the intersecting input rectangles.

To measure the I/O efficiency of the query procedure, counts were taken on the number of nodes (internal and leaves) visited, and the number of input rectangles (in the PR-Tree leaves) that intersect the query. Since each node consists of one block of rectangles, an I/O is equivalent to visiting a node, therefore the total number of I/Os performed is equivalent to the the number of nodes visited.

3.4 Update Implementation

The update procedures of the PR-Tree were performed in the same manner as R-Tree updates. Therefore after repeated updates, the PR-Tree structure will no longer abide by its definition, and should lose its theoretical query bounds. Numerous update heuristics exist, but we use a simple one to minimize the number of controllable factors affecting performance. Since node elements may be arbitrarily moved to newly created split nodes located towards the end of the stream, the relative offset indexing strategy used in the construction of the PR-Tree will not work. Therefore before any updates are performed, the PR-Tree is parsed and each offset value is changed into an absolute offset. For example, a bounding box at stream position a with offset value b, is set to offset value $a + b$. Upon completion of the updates, the tree is again parsed to revert the absolute offsets to relative offsets to retain compatibility with the query procedures.

Insertion
The insertion procedure is as follows:
Locate the desired destination leaf of the rectangle to be inserted via depth first search. At each node, the area of each bounding box is computed if the bounding box were extended to enclose the rectangle to be inserted. The next node in the traversal is the child whose bounding box increases in area the least. This ensures that the rectangle to be inserted will cause the least disruption to the bounding boxes of the PR-Tree. At the desired leaf, if room exists (pad rectangles), the rectangle is inserted. The parents of the leaf are recursively updated to reflect the new bounding box of the leaf. At each update, a new bounding box of the updated node is computed to update the parent of the node. If the desired leaf is full, a new block is created at the end of the TPIE PR-Tree stream, with the desired leaf contents and rectangle to be inserted arbitrarily split between the desired and new leaves. Upon completion of the split, new bounding boxes are computed for the desired and new leaves. The parent of the desired leaf is updated to reflect the new bounding box, and the bounding box and offset pointing to the newly created leaf are recursively inserted into the parent node of the desired leaf. In the case where the root node needs to be split, two new leaves are appended to the TPIE stream, with the root contents split between them. The original root node (at the start of the stream), now contains the bounding boxes and offsets to the newly created leaves. This allows the newly created root node to remain at the beginning of the stream to simplify querying (can still begin at offset 0).

Deletions
The deletion process is similar to the insertion, with the following differences: The the target leaf containing the rectangle to be deleted is found using depth first search, expanding upon each node whose bounding box encloses the rectangle to be deleted. Upon locating the target leaf, the occurrence of the rectangle to be deleted is set to a pad rectangle. The contents of the leaf are scanned to place all the pad rectangles at the end of the block. If the target leaf still contains some rectangles, the bounding box is recomputed and the parents of the leaf are recursively updated (in a manner similar to the insert procedure). Otherwise the delete procedure is recursively called on the PR-Tree to delete the element in the leaf's parent pointing to it. The updates to the parents and possible node deletions are performed as for a leaf.

It should be noted that the deletion of nodes and leaves incurs fragmentation of the PR-Tree. It is possible if desired to merge two under-full nodes or leaves to avoid this issue, however this would again severely alter the PR-Tree structure, and introduce another parameter in the evaluation of the updated PR-Tree. Another possible approach is to periodically defragment the PR-Tree structure. Since no nodes would be merged the query performance would be unaffected, however the size of the PR-Tree would decrease as deleted blocks are removed.

3.5 Notes

Overall, the implementation of the PR-Tree is fairly straightforward due to its similarity to R-Trees. As previously mentioned it is possible to utilize R-Tree heuristics to attempt to maintain the optimal query bounds of the PR-Tree after updates, however no guarantee of performance can be made. It is possible to retain the PR-Tree query bound performance by utilizing the External Logarithmic method [14], at the cost of increasing the insertion and deletion bounds.

4 External Priority Search Tree

The External Priority Search Tree (EPS-Tree) proposed by Arge et. al [10] is a dynamic structure for answering 3-sided queries on points in two dimensions. 3-sided queries are fundamental sub-problems of indexing in emerging data models [11]. The problem can easily be solved statically, by using a persistent B-Tree on the x-coordinates of the points, and deletion times equal to the y-coordinates of the points [12]. A 3-sided query (x_1, x_2, y) can then be answered in the optimal $O(\log_B N + T/B)$ I/Os by doing a range query (x_1, x_2) on the tree at time y.

Much previous work has been dedicated to indexing two-dimensional points for 3-sided range searching. Most of the proposed structures perform well in practise, but have sub-optimal worst-case update times, and their query performance degenerates after updates. The EPS-Tree however can perform updates in the optimal $O(\log_B N)$ I/Os amortized, while still retaining its query bound of $O(\log_B N + T/B)$.

The EPS-Tree consists of a weight-balanced base tree [12] with leaf parameter B and branching parameter $\frac{1}{4}B$ on the x-coordinates of the points. Each node v also indexes $\Theta(B)$ points from each of its $\Theta(B)$ children's x-ranges in its *query data structure*, $Q(v)$, so that each point is stored in the query data structure of exactly one node. The points indexed in a node v's query data structure from one of its children v_i's x-range (called the Y-set of v_i and denoted $Y(v_i)$) are the ones with the largest y value which are not stored in the query data structure of an ancestor of v. A detailed explanation of the query data structures is withheld due to space limitations. Their important properties are summarized as follows:

- Linear space usage ($O(B)$ blocks for $O(B^2)$ points).
- Can be built in $O(B)$ I/Os using a sweep-line algorithm.
- Static, but by using an update buffer of size B can be updated in $O(1)$ I/Os amortized.
- 3-sided range queries are performed in $O(1+T/B)$ I/Os.
- A parameter α controls the trade-off between storage redundancy and access overhead, with the redundancy being $r \leq 1 + 1/(\alpha - 1)$ and the access overhead being $A \leq \alpha^2 + \alpha + 1$ ([10], Theorem 4).

To perform a query $q = (x_1, x_2, y)$ on a node v of the EPS-Tree, q is first performed on that node's query data structure $Q(v)$. q is recursively performed on a child v_i of v if either v_i's x-range contains one of the end points x_1 or x_2 of the query, or if all of $Y(v_i)$ is returned by the query on $Q(v)$. To query an EPS-Tree its root is queried. Queries are performed in the optimal $O(\log_B N + T/B)$ I/Os, a proof is withheld due to space limitations, but is detailed in [10].

In order to insert a point p into the EPS-Tree it is first inserted into the base tree. During the traversal of the root to leaf path where p will be stored we must identify the node v which will store p in its query data structure. This is done by querying $Q(v)$ of each node v visited to identify $Y(v_i)$ for v's child v_i along the search path, and inserting p at the appropriate level. Inserting p into $Q(v)$ may cause $Y(v_i)$ to contain more than B points in which case the lowest such point must be deleted from $Q(v)$ and added to $Q(v_i)$, called a bubble down operation.

If a node v splits into nodes v' and v'' as a result of insertion into the base tree, then its query data structure must be split as well, which may result in $Q(v')$ and $Q(v'')$ containing too few points from the x-range of some of their children. In this case multiple bubble up operations (analogous to bubble down operations) must be performed to sufficiently fill $Q(v')$ and $Q(v'')$. Once again a cost analysis is withheld, but it is shown in [10] that insert operations incur a total of $O(\log_B N)$ I/Os amortized.

Deletion of a point p is straight forward, each node v along the search path querying $Q(v)$ to identify the points in $Y(v_i)$ of v's child v_i along this path. If p is in this set, it is removed from $Q(v)$, which may require a bubble up operation from $Q(v_i)$. The principal of lazy deletion and global rebuilding is used in the base tree, in that p is marked as deleted, and the whole structure is rebuilt after $\Theta(N)$ delete operations. It is shown in [10] that deletions cost $O(\log_B N)$ I/Os amortized.

The authors of [10] also give algorithms for worse case deletions, which involves scheduling the bubble-up operations to be performed lazily, and is beyond the scope of this project. They also extend their structure for 4-sided range searching, although the update time is not provably optimal.

4.1 Implementation

The EPS-Tree was implemented as described in Section 4 using the C++ programming language and TPIE [13]. Our implementation of the EPS-Tree makes use of both TPIEs blocks and streams, it is described in the following two sections which cover its structure and algorithms separately.

4.1.1 Structure

An EPS-Tree is constructed entirely of blocks, along with a small amount of additional administrative data (e.g. the number of levels in the tree, and the block ID of the root). In total, TPIE blocks are used for 7 purposes: to store nodes and leaves, as index blocks and as storage blocks on query data structures, as update blocks for query data structures, and to store the size of each of the $\Theta(B)$ Y-sets stored in a node's query data structure (called Y-blocks). Each block has a size of exactly B_{OS} bytes, but since they each store a constant amount of administrative information the exact number of elements and links must be calculated, and differs for each type of block. For a block with k links of size $s(l)$ bytes each, and storing elements of size $s(e)$ bytes and containing an info structure of size $s(I)$ bytes, the number of elements stored is

$$\left\lfloor \frac{B_{OS} - s(I) - k \cdot s(l)}{s(e)} \right\rfloor. \tag{1}$$

In the case of nodes and index blocks, the number of links stored depends on the number of elements, and we calculate both as

$$\left\lfloor \frac{B_{OS} - s(I)}{s(l) + s(e)} \right\rfloor. \tag{2}$$

Since a node stores one more element (splitter) than the number of links it stores, we set the number of elements stored to be one less than this, resulting in $s(e)$ bytes of unutilized space.

Since the query data structures store points redundantly, a small amount of arithmetic is required in order to calculate the maximum number of points which can be stored in a node v's query data structure for a particular child v_i's x-range. If we let B_I be the capacity (number of links to storage blocks) of an index block as calculated with Equation 2, and B_S be the capacity of a storage block as computed with Equation 1 (with $k = 0$), then the total number of points which can be indexed by an index block is $B_I \cdot B_S$. Given that the query data structures have a maximum redundancy of $r \leq 1 + 1/(\alpha - 1)$, we can express the total number of distinct points which can be stored in a query data structure as

$$\left\lfloor \frac{B_I \cdot B_S}{1 + 1/(\alpha - 1)} \right\rfloor = \left\lfloor \frac{(\alpha - 1)(B_I \cdot B_S)}{\alpha} \right\rfloor,$$

which, when we choose $\alpha = 2$ as recommended in [10], evaluates to $\lfloor B_I \cdot B_S/2 \rfloor$. Letting B_N be a node's maximum number of children as calculated with Equation 2, our upper bound on the size of $Y(v_i)$ for a child v_i becomes $\lfloor \lfloor B_I \cdot B_S/2 \rfloor / B_N \rfloor$. As suggested in [10], the minimum number of points in $Y(v_i)$ is half of this.

4.1.2 Algorithms

All of the algorithms were implemented in a straight forward manner from the description in Section 4, however a few details omitted in [10] are included here.

Since the authors of [10] do not give a bulk-loading algorithm for the tree, one was designed which is asymptotically worse than repeated insertion, but should result in a tree which is more evenly balanced and has the capability to be more densely packed. The bulk-loading procedure begins by constructing the base tree. It accepts a stream of points to be loaded, and begins by sorting them by x-value. The required number of leaves in order to approximate the desired "fill factor" is calculated, and the leaves are created by scanning through the points, assigning $\Theta(B)$ consecutive points to each leaf. The block IDs of the leaves are stored consecutively in a stream. In a similar manner, the exact number of level-1 nodes is calculated based on the number of leaves and the desired fill factor, and they are created by scanning through the leaf ID stream and assigning leaves to nodes. The process continues upward to the level where only one node is created - the root. In general it is not possible to perfectly balance the weight of the nodes, as they will typically have degree differences of ± 1. This will result in weight differences on level i equal in magnitude to the weight of nodes on level $i - 1$. The problem of assigning children to parents in order to globally minimize this imbalance is akin to the string matching problem, and can be solved with dynamic programming. In lieu of this more complicated method, we assign children to parents based on minimizing the difference in degrees of nodes on a level. Occasionally our technique fails, and a node's weight may violate the constraints imposed by the weight-balanced B-Tree. However this is rare, and keeping the fill factor in the range of 5%-95% results in a successful bulk load. The total number of I/Os to build the base tree is: $\Theta(sort(N) + N/B + \sum_{i=1}^{\log_B N - 1} N/B^i) = \Theta(sort(N) + N/B)$.

Once the base tree has been built the query structures are filled from the top down. At the root, the x-sorted stream of points is partitioned into substreams for each of its child's x-ranges. Each of these child substreams is then sorted by y, and a number of the highest elements corresponding to the desired fill factor are taken from each for the root's query data structure. Each child substream has its remaining elements sorted by x again, and the query structures in the corresponding child's subtree are built recursively. The total number of I/Os at this level is $O(N/B)$ for partitioning the data into substreams, $O(Bsort(N/B))$ for sorting the substreams by y, and $O(sort(B^2)+B)$ for sorting the $\Theta(B^2)$ query data structure points by x and building the query data structure from them. The total cost for building the query data structures is then given by the recurrence $T(N) = N/B + Bsort(N/B) + sort(B^2) + B + Bsort(N/B-B) + BT(N/B-B) = Bsort(N/B) + sort(B^2) + BT(N/B-B)$, whose solution could not be found by the authors.

Another detail which is left out of the work of [10] is during the bubble-up of multiple points after a node split. The authors state that the highest point in $Q(v)$ for some node v can be found in $O(1)$ I/Os with a degenerate query on $Q(v)$. However the problem with this is that not knowing the contents of $Q(v)$, a degenerate query cannot be formed which only incurs $O(1)$ I/Os. Instead we must query the whole structure in $O(B)$ I/Os, and scan through the results to identify the top point in an additional $O(B)$ I/Os. Since $O(B)$ bubble-ups may need to be performed, this adds a factor of B to the cost of a node split, causing it to incur $O(Bweight(v))$ I/Os. Since $O(weight(v))$ insertions must be performed between splits, the amortized split cost is $O(B)$ per insertion. With a possible $O(\log_B N)$ splits this increases our insert bound to $O(B \log_B N)$ I/Os amortized.

The authors of [10] also do not describe how to query the query data structures when their update buffers are not empty. We experimented with two methods. The first sorts the contents of the query structure in x, y order, and sorts the contents of the update buffer in x, y, sequence number (time stamps for the updates) order. We scan through the contents of both in parallel, calculating the final state of each point in the query structure as well as finding each new point inserted from the update buffer. However since bubble-ups require the entire contents of the query structure to be reported, the additional $O(sort(B^2))$ I/Os turned out to be too expensive when performing updates. The method used scans through the entire update buffer for each point returned from the query structure. The final state of the point is determined, and any instances of it in the update buffer are marked as processed. Then to check for the possibility of additional points added, we scan through the update buffer. For each unprocessed point, we scan forward through the update buffer and look for further deletions and insertions of this point, marking them as processed. The total cost of this algorithm is then the desired $O(1 + T/B)$ I/Os, but can incur an additional $O(B^3)$ computation steps in the worst case.

5 4-Sided Queries: Experimental Evaluation

The following experiments were performed to thoroughly compare the rectangular query, construction, and update performance of the PR and R-Trees in terms of both I/O efficiency and running times and, except for the high-dimensional experiments, closely resemble those from [14]. The data structures were implemented in C++ using the TPIE [13] template library [1]. All data was stored in 64 bit floating point format and set to lie within the unit square centred at $(0.5, 0.5)$. An additional 64 bit integer was added to all rectangles for data structure- dependent information. A block size of 4k was used for all tests and internal memory usage was limited to 128M. Wall times were recorded on the dual-processor compute nodes of the CGM1 cluster [15]. Query I/O's were measured in terms of percent overhead. This is a percentage measure of the number of block reads and writes performed during the queries, over the block size of the output elements. This in effect describes the percentage of "excess" I/Os, where 100% would mean each I/O was spent reporting output to the results.

Throughout the rest of the paper, PR will be used to denote the PR-tree as constructed by the bulk loader described in Section 3.2. H and H4 will represent R-trees built using the 2D and 4D Hilbert bulk loaders described in Section 2.2 and R* will denote an R-Tree constructed entirely from the R* inserts from Section 2.1. The bulk loaded trees are filled to 100% capacity and the R* inserts are performed with all tunable parameters, such as the minimum fanout and reinsertion factor set to the values recommended in [2].

5.1 2D Simulated Data

The following set of experiments will utilize artificially created data sets exhibiting unique characteristics in order to produce measurable distinctions between the various four-sided window query data structures. The generated data will consist of various distributions of rectangles within the unit square. The data sets will be based off of the Synthetic data described in [14] in order to allow both the verification of the original results in [14] in addition to providing a more traditional platform for the additional tests. The data sets include:

[1]TPIE includes an unofficial R-tree implementation with functionality for Hilbert bulk loading as well as R* updates, albeit in 2D only. This codebase was used as a baseline for the R-tree development used here but all nontrivial algorithms, namely updates and Hilbert sorting, were reimplemented due to issues of robustness. They were also extended to support block caching and higher dimensions.

- Aspect: A collection of uniformly distributed rectangles within the unit square, with area fixed to 10^{-6}, and the aspect ratio orientation being randomly selected for each rectangle. The aspect ratios utilized are: 10, 100, 1000, 10000, 100000

- Size: A collection of uniformly distributed rectangles within the unit square, with an upper bound on the length of the largest side (the lengths are uniformly distributed). The sizes utilized are: 0.002, 0.005, 0.01, 0.05, 0.1, 0.2

- Skew: A collection of uniformly distributed points within the unit square, with each point's y value being transformed to y^c. Thereby generating an uneven distribution of the points, with a bias towards a y value of 0. The utilized values of c include: 1, 3, 5, 7, 9

- Random: A set of uniformly distributed rectangles within the unit square, with uniformly distributed shapes and sizes.

Each generated data set consisted of 10 000 000 rectangles. Queries consisted of 100 uniformly distributed squares of area 0.01, in addition to performing the same skew transform the the query rectangles to generate corresponding skewed queries with c values of 1,3,5,7, and 9. It should be noted that for the higher dimensional tests, the Aspect data set was not utilized since the fixed area limitation produced squares that exceeded the unit-hypercube. Instead the remaining data sets were extended to produce the appropriate hyper-rectangles abiding by the dataset descriptions.

5.1.1 Loading Time

The loading time of an online data structure is arguably much less important than the query time for most applications. Still, the construction time must be reasonable for a data structure to have any practical use, and can be used to further discriminate between solutions that have similar query performance. Thus, the load times for the Aspect, Size, and Skewed data sets were measured and the results for the Aspect data, which are entirely representative of all 3 tests, may be found in Figure 1. Overall, the PR tree fares worst with the R* tree second, and the Hilbert trees being most efficient. The varying times measured in the loading of the various structures may be easily attributed to the construction methods utilized. The strict PR-Tree loading method utilizes many sorts at each level to extract the extreme priority rectangles whereas the Hilbert bulk loaders require only a single sort for the whole tree and one scan for each level. The cost of R* update heuristics is shown to be quite high as constructing the tree by inserts is an order of magnitude slower than sorting and scanning.

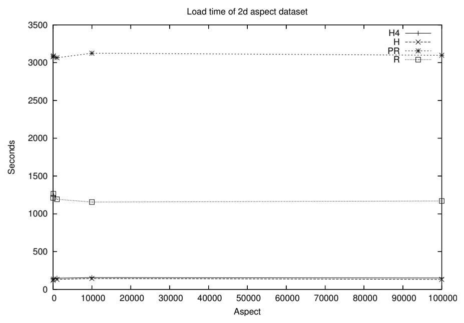


Figure 1: Time in seconds to initialize and load the R, PR, and Hilbert trees on the various 2D Aspect dataset instances. Performance on the other datasets is comparable.

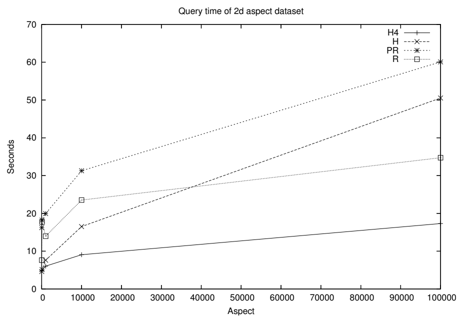
5.1.2 Query Performance

The query times for the Aspect, Size, Skewed-square queries, and Skewed-skewed queries may be viewed in Figure 2. Again, the PR-Tree tends to be the least efficient, with the R* tree being second and the Hilbert trees being most efficient. A notable exception is the case of the skewed-skewed queries data set where the query time of the PR-Tree remains constant and is exceeded by the R* tree as the amount of skew increases. The performance of the PR-Tree on skewed data may be attributed to it's utilization of the relative order of the rectangles in its construction, which does not change with skew. In the general case, the PR-Tree fared worse in terms of query time. This may be possibly attributed to the amount of in-memory computation performed at each tree node and that the PR-Tree was the only one utilizing a recursive strategy for the traversal.

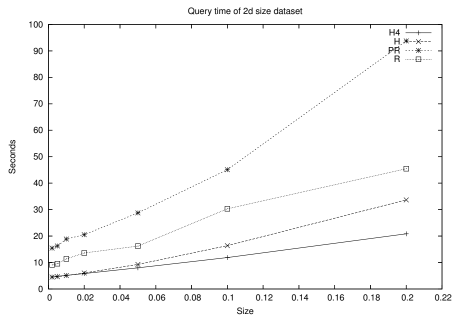
The I/O overhead for the data sets may be found in Figure 3. In general, the results tend to vary with the input data set, and again in the case of the skew-skewed queries set the PR-Tree performance remains constant and superior. The I/O overhead experiments illustrate the theoretical differences of the various structures. Furthermore, the asymptotic I/O performance may be inferred from the results as the values for the different data structures diverges. This may serve as an indicator of wall-time performance on even larger input data sets, as the number of I/Os performed dominates the execution time.

The lackluster performance of the R* tree shows that the greedy inserts cannot compete with a proper bulk loading algorithm. Furthermore, the trees created by inserts tended to be approximately two-thirds full which is necessarily a tremendous disadvantage compared to the fully-packed bulk loaded trees.

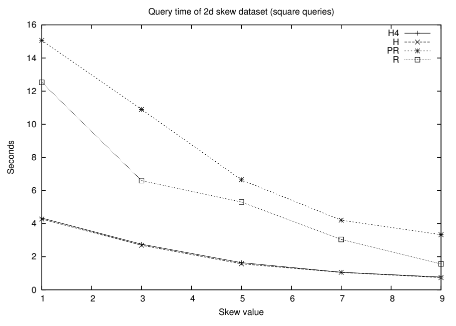
The difference between the 2D and 4D Hilbert loaders is very apparent in the results for the aspect data. Two rectangles in



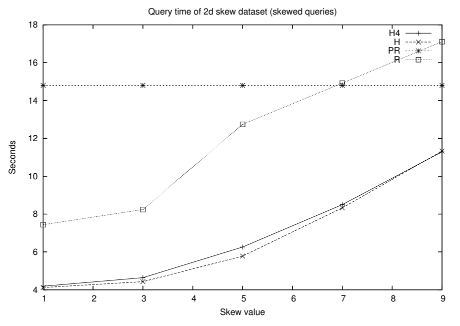
(a) Aspect



(b) Size



(c) Skew-Square Queries



(d) Skew-Skewed Queries

Figure 2: Time in seconds to query the generated 2D datasets over their various parameter values for the R, PR, and Hilbert trees.

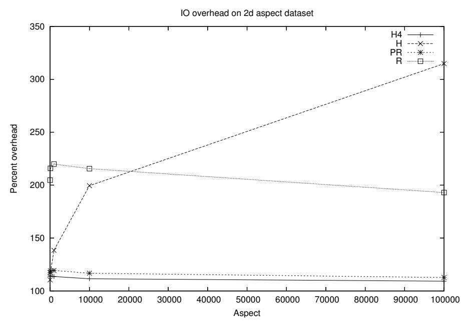
this dataset may have the same midpoints but occupy radically different areas due to differences in aspect and orientation. Thus, the H4 algorithm is better able to sort these rectangles as it uses their four extremities rather than just their midpoint. The same explanation applies to the size data, only the difference in shapes between the various input rectangles is less extreme in this case and the performance difference is accordingly less pronounced. Finally, the skewed data/skewed queries experiment again illustrates the invariance of the PR-Tree to skew, since the relative order of the rectangles and queries are preserved by the transformation.

5.1.3 Dimensionality

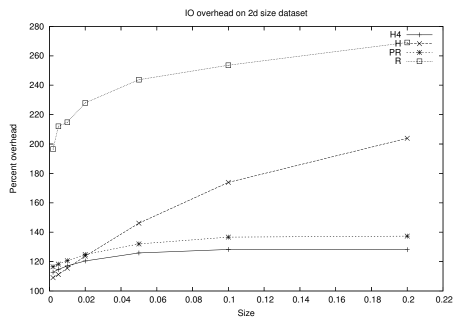
Figures 4, and 5 display the I/O overhead of queries on the 4 and 8 dimensional equivalents of the artificial data. In all cases, the PR-Tree outperforms the other data structures, with the R-Tree being least efficient. Unfortunately the R-Tree results for 8 dimensions are not available as they did not finish in time. The difference in performance between the PR and Hilbert trees may be attributed to the theoretical asymptotic differences between their query bounds. The PR-Tree performs queries in $O((\frac{N}{B})^{1-\frac{1}{d}} + \frac{T}{B})$ I/Os while the R-Trees require $O(\frac{dN}{B})$ I/Os. The structure and heuristics of the R-Trees are designed for 2D and simply do not scale gracefully to higher dimensions as discussed in Section 2.3.

5.1.4 Dynamic Updates

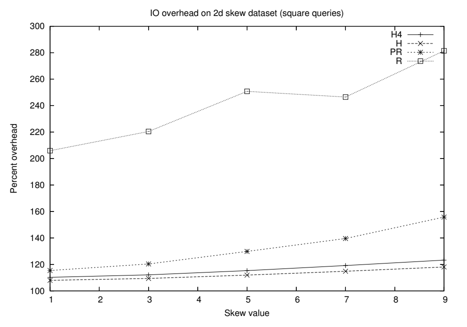
This experiment measures both the efficiency of insertions and deletions as well as their effect on query performance of the different data structures. The structures were initially loaded with 5 million rectangles generated with a random uniform distribution. 10 rounds of 1000 random updates comprising of equal numbers of random insertions and deletions were performed and the I/O cost of each round recorded in Figure 6(a). The query overhead of 100 square queries on the data was measured after the in initial loading as well as after each update round and the results are shown in Figure 6(b). It was expected that the R*-Tree would show



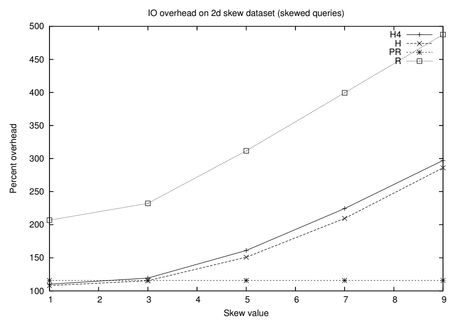
(a) Aspect



(b) Size



(c) Skew-Square Queries



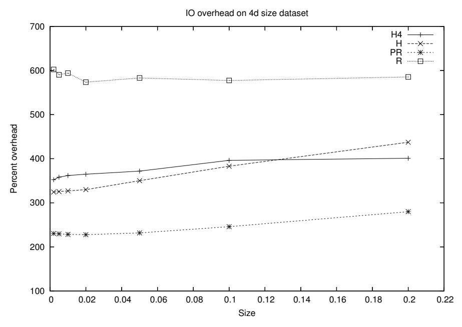
(d) Skew-Skewed Queries

Figure 3: I/O overhead (measured as a percentage of the number of block I/Os performed over the block size of the query output) on the various input data set instances, for the R, PR, and Hilbert trees.

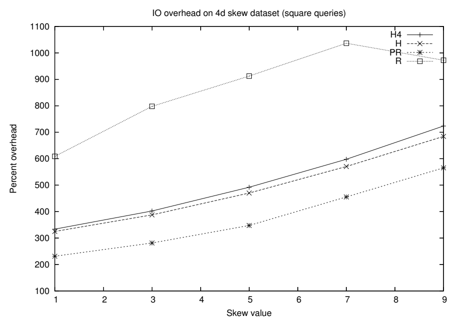
the least degradation due to updates since its leaves are not packed to 100% capacity whereas the bulk loaded trees are full. While the results do support this hypothesis, the H and H4 performance was only slightly affected. The PR-Tree is unsurprisingly impacted the most as no node splitting or merging heuristics are used to keep the tree well packed. The I/O cost of the updates themselves shown in Figure 6(a) is several orders of magnitude greater for the PR-Tree which is counterintuitive as the algorithm employed is actually simpler due to the absence of forced reinsertions. This difference is likely due to implementation issues, specifically, the PR-Tree was implemented based on TPIE Streams whereas the R-Trees are built upon block collections which are lower-level and better at handling random I/Os.

5.2 GIS Data

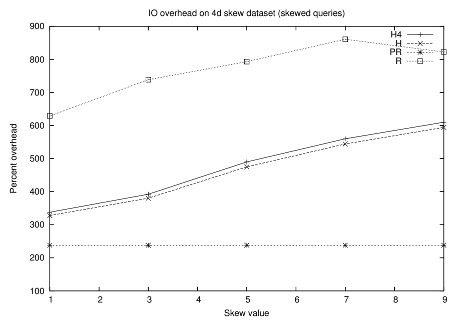
A large amount of GIS data is made freely available online by the National Atlas of the United States of America [9]. The data contains point, polyline and polygon vector data in ESRI Shapefile format corresponding to a broad range of geographic, demographic, and economic information. By running random queries of different sizes on this data, we hope to gain insight into how the different data structures would perform in a real world spatial database system. The data was preprocessed as follows. Features outside continental US were removed, all data was normalized to the unit square and duplicate geometry was removed (to help the EPS tree). In order to enlarge the resulting 611,254 features into a massive dataset, a bounding box was generated for each *segment* of each feature. The resulting data comprised of approximately 11 million rectangles and took up 444M on disk. The query performance of the same 100 queries used on the Size and Aspect data above is shown in Figure 7. The results are generally consistent with what has been seen for the simulated data in that the Hilbert R-Trees are the most efficient, followed by the PR and R trees. Interestingly, the PR tree's running time is greatly affected by the query size, despite its competitive I/O overhead. Again, this is more likely due to implementation issues and in-memory computation. Overall, the Hilbert and, to a lesser extent, R* heuristics show their value for real-world



(a) Size



(b) Skew-Square Queries



(c) Skew-Skewed Queries

Figure 4: I/O overhead (percentage of performed I/Os over the query output block size) on various 4D input data set instances.

applications with superior running times.

6 3-Sided Queries: Experimental Evaluation

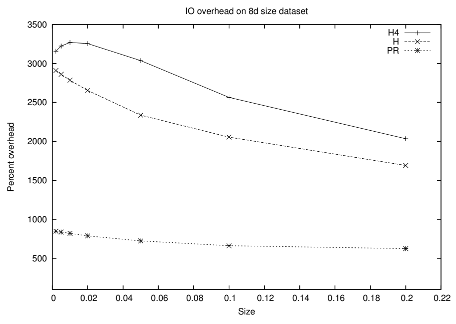
In evaluating the EPS-Tree vs. the other data structures, our primary concerns were with testing scalability, and identifying the constants hidden by asymptotic analysis. We design tests to explore these factors on synthetic and real data.

Aside from the structures described in the earlier sections of this paper, we also compare performance on a B+-Tree, which was already implemented in TPIE. A B+-Tree stores its leaves in a linked list, and so is particularly good at performing range queries. We perform a 3-sided query (x_1, x_2, y) on the B+-Tree by performing the range query (x_1, x_2), and then filtering the results based on their y value. We expect that the performance of the B+-Tree should deteriorate as y grows, since it is forced to filter through a large number of points which are not in the query output.

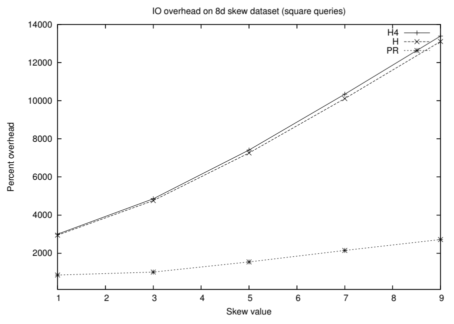
For all of our experiments, our maximum memory size M was set to 128MB, and our block size B_{OS} was set to 4kB. For fair comparison with R and PR-Trees, additional storage space was added to the representation of a two-dimensional point, so that each point consumes the same amount of space in memory and on disk as a two-dimensional rectangle. This is due to the way in which the R and PR-Trees were implemented for 3-sided queries on points, which involved representing points as rectangles with 0 extent, and representing 3-sided queries as rectangles with infinite extent in the positive y direction.

The theoretical groundwork of the PR-Tree still applies to points, and so the PR-Tree performance is expected to be consistent with the previous four sided results. The PR and R-Trees however may perform better on points than on rectangles since there is no overlap within the data and hence the partitions will result in non-overlapping bounding boxes. In the case of the PR-Tree the priority leaves will be disjoint as well.

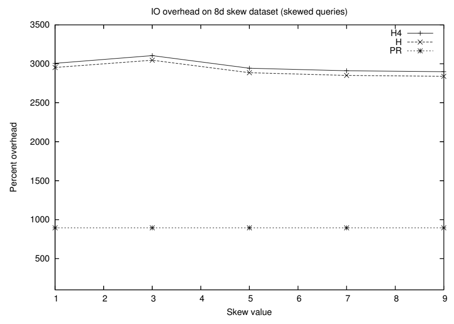
For the static tests, the EPS-Tree is bulk loaded using the algorithm described in Section 4.1.2. This is because surprisingly, the bulk-loading algorithm is much faster than repeated



(a) Size



(b) Skew-Square Queries



(c) Skew-Skewed Queries

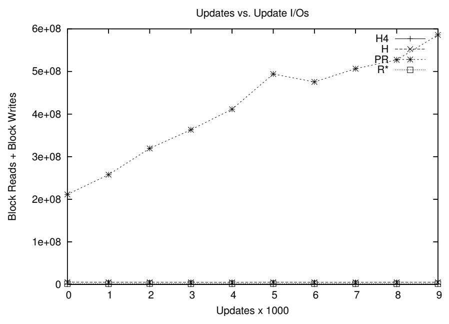
Figure 5: I/O overhead (percentage of performed I/Os over the query output block size) on various 8d input data set instances.

insertions. For example on one experimental platform, bulk loading a tree of 100,000 uniformly generated points took 6.89 seconds and 9,989 block I/Os, while inserting 100,000 points into an initially empty tree took 331.88 seconds and 2,575,198 block I/Os.

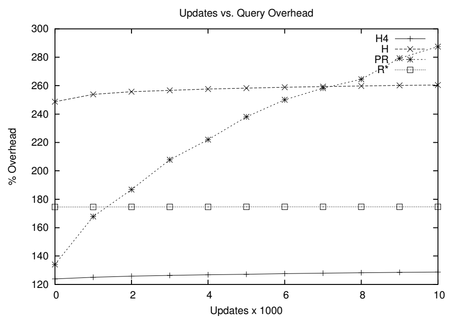
Our first test examines the size of each data structure on disk as a function of the raw size of the input it indexes. All of the data structures use linear space, but as Figure 8 shows, the constants are much higher for the EPS-Tree. This is not surprising because not only is each point stored in exactly two places (in a leaf of the base tree and a query data structure), but the query data structures have a redundancy of at most two, and because of the large number of blocks used in addition to those which store the raw data.

Our second test compares the average overhead for a batch of 100 3-sided queries as a function of the size of the data set. A total of two different data distributions and two different query distributions were chosen. The data distributions (pictured in Figure 12 of the Appendix) are *uniform*, for uniformly distributed points, and *diagonal*, for points uniformly distributed within a region bounded by 10% the size of the space in either direction of the diagonal. The query distributions are *uniform*, with each component of the query distributed uniformly, and *high-y*, where the y values of the queries are distributed uniformly in the top 20% of the space. We expect that this is where the EPS-Tree will have the largest advantage, since it should be able to avoid visiting a larger number of nodes which will not contribute to the query output. Similarly we expect the EPS-Tree to perform well on the diagonal distribution. This is because queries with a larger x_1 and x_2 will terminate near the top of the tree because this is where most of their results will be stored, and queries with a smaller x_1 and x_2 as well, since the root will not propagate queries to children whose entire Y-set has not been reported.

The results, shown in Figure 9 tend to support these hypotheses, although the EPS-Tree performs substantially poorer than the other techniques. This is not surprising, since the access overhead of the query data structures themselves is $A \leq \alpha^2 + \alpha + 1 = 7$ when $\alpha = 2$. This could account for a 700% overhead alone, and further loading the index blocks, Y blocks, nodes and leaves should increase the overhead substantially. For the high-y queries the overhead is increased,

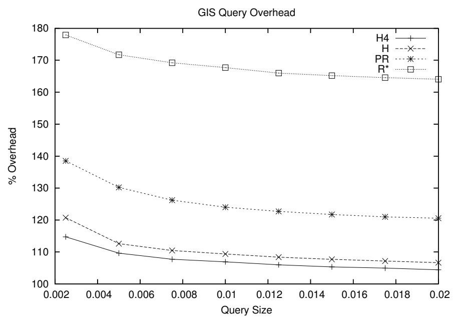


(a) Update I/Os (blocks read + blocks written)

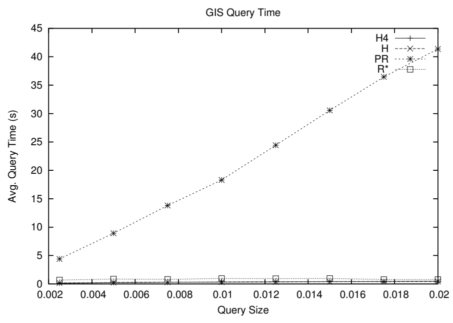


(b) Query Overhead

Figure 6: Update performance of each data structure for square 4-sided queries on uniformly distributed rectangles. Each tree was bulk loaded with 5,000,000 rectangles, and had 10 batches of 1000 updates each applied. The number of I/Os performed for the updates are shown in (a), while the query overhead after each batch of updates is shown in (b).



(a) Query Overhead

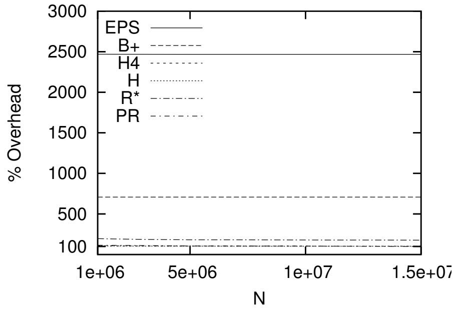


(b) Query Time

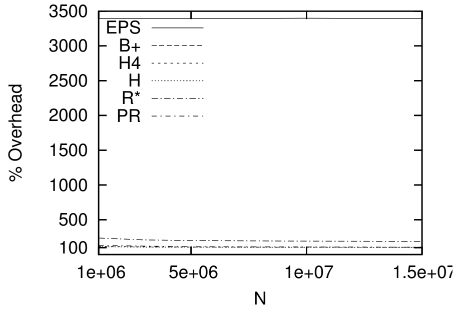
Figure 7: Query overhead and query time for rectangular queries on the GIS data set. The query overhead shown is for a total of 100 uniform 4-sided queries, and 10 independent random trials were performed.

due to the smaller size of the result set. However the overhead of the B+-Tree is increased as expected. The combination of diagonal data with high-y queries is crippling for the B+-Tree, due to the presence of queries with small x_1 and x_2 which contain no results. The PR and R-Trees perform exceptionally well in all cases. This is because the point data does not overlap and is hence the ideal case for both structures, preventing them from visiting children unnecessarily during a query. All of the data structures scale well with increasing data set sizes, with the overhead remaining nearly constant across the different trials.

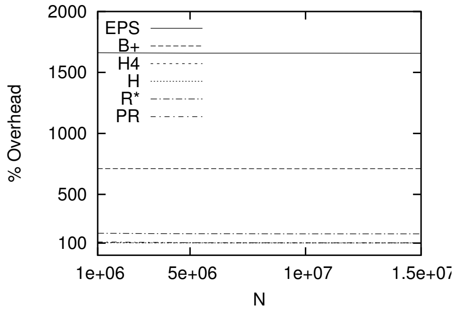
Our next test examines the update efficiency, as well as the query time after updates of each data structure for 3-sided queries on points. The same experiment was performed as in Figure 6 where each tree is bulk loaded with 5,000,000 points, and has 10 batches of 1000 updates each applied, each batch having an equal number of insertions and deletions. The number of I/Os performed for each batch, as well as the query overhead after each batch are plotted in Figure 10. Most of the data structures perform equally, except for the EPS-Tree which performs many more I/Os for each update operation. This expense is likely attributed to the cost for keeping the query data structures updated, which requires expensive bubble up and bubble down operations. The query overhead of the EPS-Tree increases slightly after each batch of updates. This is surprising, since the application of updates for the EPS-Tree means a larger number of computation steps must be performed scanning the update buffer of each query data structure, but the number of disk operations should remain the same. Our only explanation is that the tree has grown in size due to the inserts, but has not decreased in size since deletions are performed lazily. Unfortunately due to the slowness of updates on the EPS-Tree we were not able to test



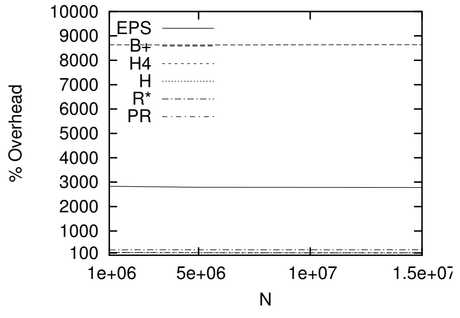
(a) Uniform/Uniform



(b) Uniform/High-Y



(c) Diagonal/Uniform



(d) Diagonal/High-Y

Figure 9: The percent overhead for 100 random 3-sided queries on each data structure, with each data and query distribution as the number of points N is varied.

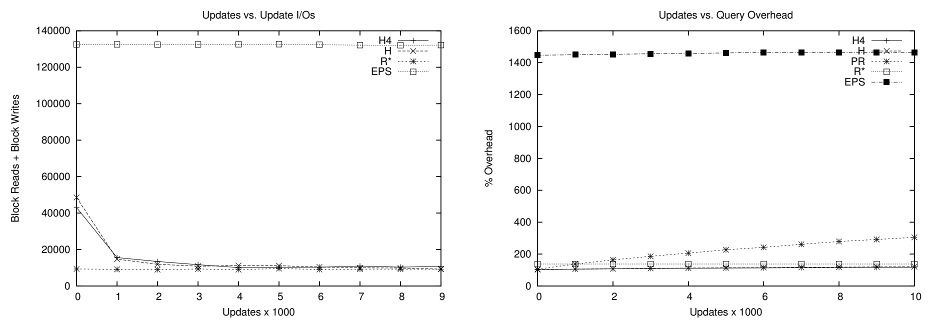
with a large enough number of deletes to force the tree to be globally rebuilt. Similarly to the 4-sided dynamic update experiment, the PR-Tree exhibits degrading performance with each successive round of updates as no heuristics are utilized to perform insertions and deletions. Therefore the optimal query bound is lost as the structure deviates from the PR-Tree definition.

Our final test is a repeat of the test shown in Figure 9, except using the GIS data as described in Section 5.2. To convert the GIS data to points, the median of each rectangle was used. The resulting point distribution is shown in Figure 12 (c) of the appendix. The results, shown in Figure 11 (a) are consistent with those of Figure 9, demonstrating that our tests on simulated data are accurate for some real world data as well. We also measured the number of I/Os necessary to construct the indexes, shown in Figure 11 (b), where we find the EPS and B-Trees to be most efficient, with the Hilbert Trees following shortly thereafter. The R*-Tree is least efficient because it was not bulk loaded, but rather created with repeated insertions. The results for the PR-Tree are not shown since it is much less efficient than the others and has a tendency to make the other bars in the plot indistinguishable.

7 Conclusions and Future Work

We have evaluated 6 data structures for 3-sided queries on points in two dimensions, 4 of which were additionally evaluated on rectangle queries in 2, 4, and 8 dimensions. One of the structures, the R*-Tree, relies on heuristics and does not have provably optimal query bounds, while the External Priority Search (EPS) Tree and the Priority R (PR) Tree do. The EPS-Tree also is dynamic and has optimal update time amortized. Our experiments compare the data structures on different types and dimensionality of simulated data, as well as real GIS data.

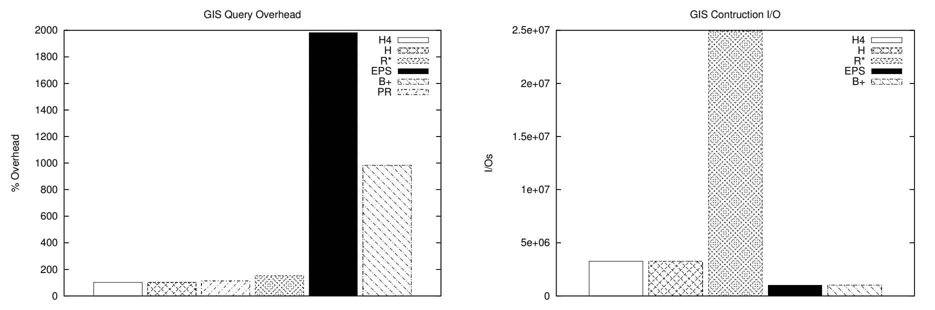
We find that the EPS-Tree's size, a result of redundantly stored points and a large number of administrative blocks, inhibits its performance. Our data set sizes may still be too small



(a) Update I/Os (blocks read + blocks written)

(b) Query Overhead

Figure 10: Update performance of each data structure for uniform 3-sided queries on uniformly distributed points. Each tree was bulk loaded with 5,000,000 points, and had 10 batches of 1000 updates each applied. The number of I/Os performed for the updates are shown in (a), while the query overhead after each batch of updates is shown in (b).



(a) Query Overhead

(b) Construction I/Os

Figure 11: Query overhead and index construction I/Os for 3-sided queries on the GIS data set, converted to 2D points. The query overhead shown is for a total of 100 uniform 3-sided queries, and 10 independent random trials were performed.

to rule out the possibility of asymptotic performance which overrules the constant factors, since the trees constructed for our tests have only a small number of levels (≤ 4). However testing with data large enough for the trees to have a significant number of levels would take weeks, and as such was not feasible within our time frame.

The PR-Tree was less I/O efficient than the Hilbert loaded R-Trees in all the 2D experiments except for 4-sided queries on the skewed points. Even in this case, however, the running time was still significantly greater. Unsurprisingly, the trees created with R* inserts exhibited consistently poorer efficiency than their bulk-loaded counterparts. In the higher dimensional experiments, the PR-Tree consistently out performed the other structures, thereby supporting its asymptotic query bound. However, it would be interesting to benchmark it against a more realistic solution in higher dimension such as the X-Tree. The lack of update heuristics in this implementation of the PR-Tree was made apparent in the observable degradation of performance after successive updates. A possible remedy would be the utilization of the External Logarithmic method [14], to allow for the retention of the optimal query bound after updates; the implementation and associated experiments will have to be deferred to future work. Finally, the standardization of implementation techniques (recursion, stream and block use), would allow for more accurate comparisons between the PR-Tree and other structures in terms of execution time, and the number of I/Os required in the construction and update of the structure. Overall, the experiments show the 4d Hilbert R-Tree (H4) to be the most practical datastructure for 3 and 4-sided queries on 2D points and rectangles despite relying on heuristics while the PR-Tree is the best choice of structures tested for higher dimension

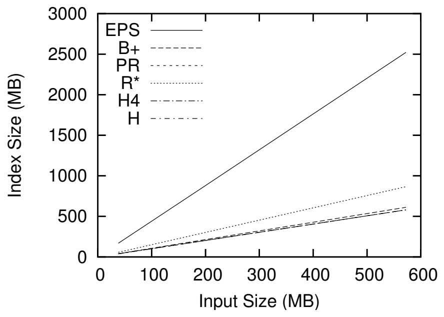


Figure 8: The size of the index created by each data structure as a function of the size of the raw input data. The number of points corresponding to the domain shown here is 1,000,000 to 15,000,000.

There are still many more possibilities for evaluation that time and space will not allow, such as whether the bulk-loaded trees give better query performance than those constructed with repeated insertion. It would also be beneficial to evaluate all of the data structures on the same system, so we could determine if the relative performance as measured by wall time is equal to the relative performance as measured by number of I/Os. This would give an indication of the relative cost of in-memory computation of the structures, vs. the cost of accessing disk, and as such validates the I/O model for analyzing these structures. Unfortunately testing on a common platform was infeasible, as tests involving the EPS-Tree caused hard to diagnose kernel panics on our largest and most powerful experimental platform, and so they were relegated to a much slower PC.

There is plenty of room for further optimizations as well, particularly in the EPS-Tree. For example the points do not need to be stored in the leaves at all, since they are never accessed there. The tree would work properly with only the splitters to guide the queries and the query data structures to answer them. Other optimizations are possible, for example in identifying the highest point stored in a query data structure, we need not formulate a degenerate query at all. The index information stored should identify which block contains the highest point, and it can then be read in $O(1)$ I/Os. This would lower the insert cost to the optimal $O(\log_B N)$ I/Os amortized.

We also never experimented with different block sizes and memory sizes. The performance of the data structures themselves could depend on these variables to a high degree.

8 Appendix

Figures 12 and 13 display the data distributions used for two-dimensional points and rectangles respectively.

References

[1] A. Guttman. R-trees: *A Dynamic Index Structure for Spatial Searching*. Proc ACM SIGMOD. 1984.

[2] N. Beckmann, H.P. Kriegel, R Schneider, B Seeger. *The R*-tree: an efficient and robust access method for points and rectangles*. Proc ACM SIGMOD. 1990.

[3] P. Rigaux, M. Scholl, A. Voisard. *Spatial Databases with Application to GIS.* Morgan Kaufman Publishers. 2002

[4] I. Kamel, C Faloutsos. *Hilbert R-tree, and improved R-Tree using Fractals.* Proc. VLDB. 1994

[5] B. Moon et al. *Analysis of the Clustering Properties of Hilbert Space-filling Curve*. Submitted to IEEE Transactions on Knowledge and Data Engineering, March 1996.

[6] D Moore. *Hilbert C Implementation*. Dept. of Computational and Applied Math. Rice University. 2000.

[7] AR Butz. *Alternative Algorithm for Hilbert's Space-Filling Curve*. IEEE Trans. Comp., April. 1971

[8] S Berchtold, DA Keim and HP Kriegei. *The X-Tree: An Index Structure For High-Dimensional Data*. Proc VLDB 22. Bombay, India. 1996.

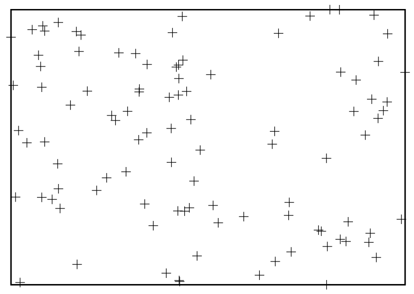
[9] *The National Atlas of the United States of America*. http://www.nationalatlas.gov/. United States Dept. of the Interior. 2005

[10] L. Arge, V. Samoladas, and J.S. Vitter. *On two-dimensional indexability and optimal range search indexing.* In Proceedings of the 18th ACM Symposium on Principles of Database Systems, pages 346357, 1999.

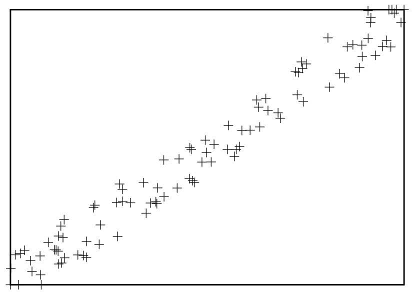
[11] P.C. Kanellakis, S. Ramaswamy, D.E. Vengroff and J.S. Vitter. *Indexing for data models with constraints and classes.* Journal of Computer and System Sciences, 52(3):589-658, 1996.

[12] L. Arge. *External-memory geometric data structures.* Lecture notes of EEF Summer School on Massive Data Sets, Aarhus, 2002.

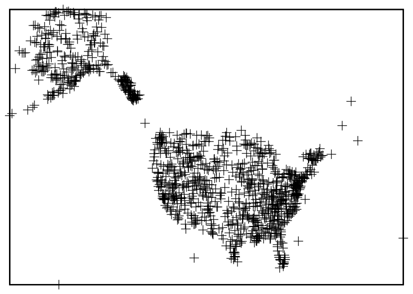
[13] TPIE, A Transparent Parallel I/O Environment, "http://www.cs.duke.edu/TPIE". August 19, 2005.
L. Arge, O. Procopiuc, and J.S. Vitter. *Implementing*



(a) Uniform



(b) Diagonal



(c) GIS (1 of every 10,000 features sampled)

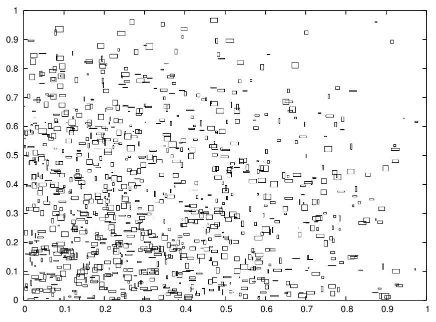
Figure 12: Data distributions for 3-sided queries on points.

I/O-efficient data structures using TPIE. In Proc. European Symposium on Algorithms, 88-100, 2002.

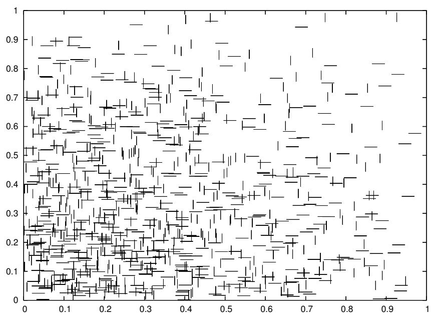
[14] L. Arge, M. de Berg, H. Haverkort, and K. Yi. *The Priority R-Tree: A practically Efficient and Worst-Case Optimal R-Tree.* Proceedings of the 2004 ACM SIGMOD international conference on Management of data, pages 347-358, 2004. “http://citeseer.ist.psu.edu/arge04priority.html”

[15] CGM User’s Manual, “http://cgm1.cs.dal.ca/UserManual.html”. August 15, 2005.

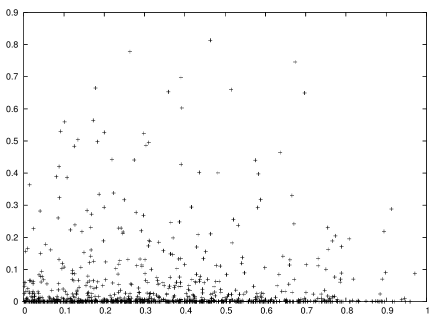
[16] Mokbel, M., Aref, W.G., Kamel, I. 2002. *Performance of Multi-Dimensional Space-Filling Curves.* Proceedings of the International Conference on Geographic Information Systems.



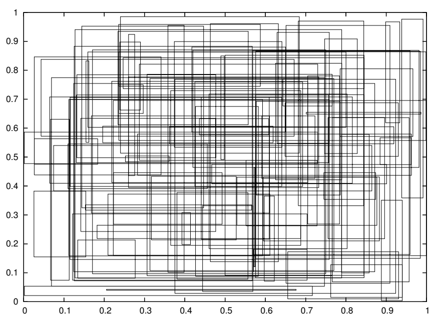
(a) Size 0.02



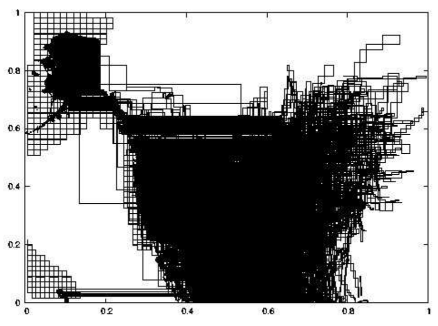
(b) Aspect 1000



(c) Skew 5



(d) Random Input for Updates



(e) GIS Data

Figure 13: Data distributions for 4-sided queries on rectangles. Every 10,000th rectangle is sampled except for (d) and (e) where every 50,000th and single rectangle were sampled respectively.