# Path Caching: A Technique for Optimal External Searching

Sridhar Ramaswamy and Sairam Subramanian

Department of Computer Science
Brown University
Providence, Rhode Island 02912

**CS-94-27**
May 1994

# Path Caching: A Technique for Optimal External Searching

(Extended Abstract)

**Sridhar Ramaswamy***
Brown University
sr@cs.brown.edu

**Sairam Subramanian**†
Brown University
ss@cs.brown.edu

## Abstract

External 2-dimensional searching is a fundamental problem with many applications in relational, object-oriented, spatial, and temporal databases. For example, interval intersection can be reduced to 2-sided, 2-dimensional searching and indexing class hierarchies of objects to 3-sided, 2-dimensional searching. *Path caching* is a new technique that can be used to transform a number of time/space efficient data structures for internal 2-dimensional searching (such as segment trees, interval trees, and priority search trees) into I/O efficient external ones. Let $n$ be the size of the database, $B$ the page size, and $t$ the output size of a query. Using path caching, we provide the first data structure with optimal I/O query time $O(\log_B n + t/B)$ for 2-sided, 2-dimensional searching. Furthermore, we show that path caching requires a small space overhead $O(\frac{n}{B}\log_2\log_2 B)$ and is simple enough to admit dynamic updates in optimal $O(\log_B n)$ amortized time. We also extend this data structure to handle 3-sided, 2-dimensional searching with optimal I/O query-time, at the expense of slightly higher storage and update overheads.

## 1 Introduction and motivation

The successful realization of any data model in a large-scale database requires supporting its language features with efficient secondary storage manipulation. Consider the relational data model of [Cod]. While the declarative programming features (relational calculus and algebra) of the model are important, it is crucial to support these features by data structures for searching and updating that make optimal use of secondary storage. B-trees and their variants $B^+$-trees [BaM, Com] are examples of such data structures. They have been an unqualified success in supporting external dynamic 1-dimensional range searching in relational database systems.

The general data structure problem underlying efficient secondary storage manipulation for many data models is external dynamic $k$-dimensional range searching. B-trees, which are good for 1-dimensional range searching, are inefficient for handling more general problems like two and higher dimensional range search.

The problem of multi-dimensional range searching in both main memory and secondary memory has been the subject of much research. Many elegant data structures like the priority search tree, segment tree, and interval tree have been proposed for use in main memory for special cases of 2-dimensional range searching. In this paper, we introduce a new technique called *path caching* that can be used to convert many of these in-core data structures into secondary storage data structures. These data structures have optimal query time at the expense of small storage overheads. The technique is also simple enough to allow updates in optimal amortized time.

We first introduce our model for secondary storage algorithms and then look at the performance of B-trees for 1-dimensional range searching. We make the standard assumption that each secondary memory access transmits one page or $B$ units of data, and we count this as one I/O.[1] The efficiency of our algorithms is measured in terms of the number of I/O operations that they perform. Let $R$ be a relation with $n$ tuples and let the output of a query on $R$ have $t$ tuples. Our I/O bounds are expressed in terms of $n, t$ and $B$ and all constants are independent of these three parameters.

* *Contact Author.* Address: Dept. of Computer Science, Brown University, Box 1910, Providence, RI 02912. Tel: 401-863-7662. Fax: 401-863-7657. Research supported by ONR Contract N00014-91-J-4052, ARPA Order 8225.

† Address: Dept. of Computer Science, Brown University, Box 1910, Providence, RI 02912. Tel: 401-863-7673. Fax: 401-863-7657. Research supported by NSF PYI award CCR-9157620, together with matching PYI funds from Honeywell Corporation, Thinking Machines Corporation, and Xerox Corporation. Additional support provided by ARPA contract N00014- 91-J-4052 ARPA Order No. 8225

[1] We will use the words page and disk block interchangeably, as also in-core and main memory. Also, the symbol log used without a base defaults to base 2.

A $B^+$-tree on attribute $x$ of the $n$-tuple relation $R$ uses $O(\frac{n}{B})$ disk blocks. The following operations define the problem of *external dynamic 1-dimensional range searching* on relational database attribute $x$: (1) Find all tuples such that for their $x$ attribute $a_1 \leq x \leq a_2$. If the output size is $t$ tuples, then the $B^+$-tree can answer this query in $O(\log_B n + t/B)$ disk I/O's in the worst case. If $a_1 = a_2$ and $x$ is a key then this is key-based searching. (2) Inserting or deleting a given tuple into the $B^+$-tree can be done in $O(\log_B n)$ disk I/O's in the worst case. It can be shown that the performance of $B^+$-trees is optimal for this problem. The problem of *external dynamic k-dimensional range searching* on relational database attributes $x_1, \ldots, x_k$ generalizes 1-dimensional range searching to $k$ attributes, with range searching on $k$-dimensional intervals.

Many efficient algorithms exist for 2-dimensional range searching and its special cases (see [ChT] for a detailed survey). Most of these algorithms are not efficient when mapped to secondary storage. However, the practical need for good I/O support has led to the development of a large number of external data structures, which do not have good theoretical worst-case bounds but have good average-case behavior for common spatial database problems. These includes the grid-file [NHS], various quad-trees [Sama, Samb], z-orders [Ore] and other space filling curves, k-d-B-trees [Rob], hB-trees [LoS], cell-trees [Gün], and various R-trees [Gut, SRF]. For these external data structures there has been a lot of experimentation but relatively little algorithmic analysis. Their average-case performance (e.g., some achieve the desirable static query I/O time of $O(\log_B n + t/B)$ on "average" inputs) is heuristic and usually validated through experimentation. Moreover, their worst-case performance is much worse than the optimal bounds achievable for dynamic external 1-dimensional range searching using $B^+$-trees(see [KRV] for a more complete reference on the field). In this paper, we are interested in obtaining algorithms with good worst-case performance. Using path caching, we study tradeoffs between time and space in secondary memory.

Two special cases of 2-dimensional range searching have been studied extensively in the literature. The first one is dynamic interval management in secondary storage. This problem is crucial to indexing in constraint databases and temporal databases [KKR, KRV]. It is shown in [KRV] that the key component of dynamic interval management is answering *stabbing queries*. Given a set of input intervals, to answer a stabbing query for a point $q$ we have to report all intervals that intersect $q$. Elegant solutions exist for this problem in main memory. The segment tree [Ben], interval tree [Edea, Edeb], and the priority search tree [McC] can all solve this problem well. Of these, the priority search tree solves a slightly more general problem (3-sided queries) with optimal query and update times and uses optimal storage. Many algorithms have been presented to solve this problem in secondary memory. These include [BlGa, BlGb, IKO]. The first I/O optimal solution for this problem appeared in [KRV]. [KRV] reduces dynamic interval management to stabbing queries, which in turn reduce to a special case of 2-dimensional range searching called *diagonal corner queries* (see Figure 1). Diagonal corner queries can be answered in optimal time $O(\log_B n + t/B)$ using optimal storage $O(\frac{n}{B})$. The solution presented in [KRV] is fairly involved and does not support deletion of points.

In this paper, we present a data structure for solving a more general version of this problem, namely 2-sided queries (see Figure 1). We use path caching and obtain bounds of $O(\log_B n + t/B)$ I/O's for query time and $O(\log_B n)$ for amortized updates. The data structure occupies $O(\frac{n}{B} \log\log B)$ storage.

The second important special case of 2-dimensional range searching is 3-sided range searching (see Figure 1). The priority search tree can answer 3-sided queries in-core in time $O(\log n + t)$, using storage $O(n)$. The update time is $O(\log n)$ in the worst-case. All these bounds are optimal. It is shown in [KRV] that answering 3-sided queries efficiently is key to solving the problem of indexing classes. Indexing classes is the natural generalization of indexing in the context of object-oriented databases and is very important to their good performance (see [KiL, ZdM] for more information on this area). [KKD, LOL] present solutions to the problem of indexing classes. However, their algorithms are based on heuristics and cannot guarantee good worst-case performance.

Previous attempts to answer 3-sided queries in secondary memory by implementing priority search trees in secondary memory [IKO, KRV] did not have optimal query times. [IKO] uses optimal storage but answers queries in $O(\log n + t/B)$ time. [KRV] improves on this, answering queries in $O(\log_B n + \log B + t/B)$ time using optimal storage. Neither of them allow inserts and deletes from the data structure. We present a data structure to solve this problem using path caching that answers queries in optimal time $O(\log_B n + t/B)$, but uses storage $O(\frac{n}{B} \log B \log\log B)$ and performs updates in $O(\log_B n \log^2 B)$ time. In addition to these data structures, path caching can also be applied to other main memory data structures to obtain optimal query times at the expense of small space overheads. By doing this, we improve on the bounds of [BlGb] for implementing segment trees in secondary memory.

To summarize, we present a simple technique called path caching that can be used to transform many in-core data structures to efficient secondary storage structures. We show how to use path caching to implement priority search trees in secondary memory

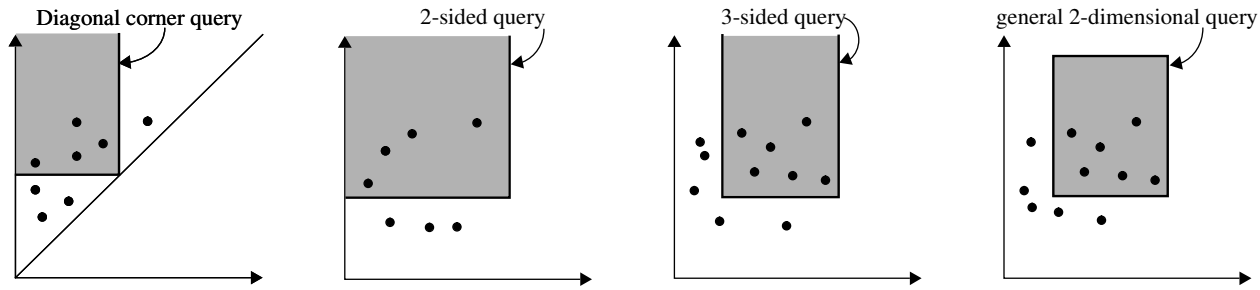


Figure 1: Diagonal corner queries, 2-sided, 3-sided and general 2-dimensional range queries.

for answering general 2 and 3-sided queries with optimal I/O. We also apply this technique to improve on existing bounds for segment trees in secondary memory, and to implement a restricted version of interval trees in secondary memory.

The rest of the paper is organized as follows. Section 2 explains the general principles behind path caching by applying it to segment trees. Section 3 explains the application of path caching to priority search trees to obtain reasonably good worst-case bounds for 2-sided range searching. We also present results about the application of path caching to 3-sided range searching, segment trees and interval trees. Section 4 applies recursion and path caching to 2-sided queries to improve on the bounds of Section 3. It also briefly touches on applying similar ideas to 3-sided queries. Section 5 shows how updates to the data structures can be handled in optimal amortized time. We finally present our conclusions and open problems in Section 6.

## 2 Path caching

Let us illustrate the idea of path caching by applying it to the main memory data structure *segment tree*. The segment tree is an elegant data structure that is used to answer stabbing queries on a collection of intervals.

Before we discuss the use of path caching in this context we give a brief description of the segment tree; a more complete treatment can be found in [Ben]. For ease of exposition we will assume that none of the input intervals share any endpoints.

To build a segment tree on a set of $n$ intervals we first build a binary search tree $T$ on the $2n$ endpoints of the intervals. The endpoints $e_1, e_2, \cdots, e_k$ are stored at the leaves of the search tree in sorted order. With each node $x$ in the tree we associate a half-open interval[2] called the *cover-interval* of $x$. If $x$ is a leaf node containing the endpoint $e_j$ then the cover-interval of $x$ is the half-open interval $[e_j, e_{j+1})$. If $x$ is an internal node then its cover-interval is the union of the cover-intervals of its children. To answer stabbing queries we store each input interval $I$ in up to $2\log n$ nodes of the tree. These nodes are called *allocation nodes* of interval $I$. A node $x$ is an allocation node of interval $I$ if $I$ contains the cover-interval of $x$ and does not contain the cover-interval of $x$'s parent. The intervals stored at node $x$ are placed in a list $CL(x)$ called the *cover-list* of $x$.

Given a query point $q$, let $P$ be path of $T$ from the root to the leaf $y$ such that $q$ is in the cover-interval of $y$. It is not hard to show that the intervals containing the query point $q$ are exactly those intervals that are stored at the nodes on $P$. The time required to answer such a query is $O(\log n + t)$ (where $t$ is number of intervals that contain $q$), which is optimal. The data structure occupies $O(n \log n)$ space because each interval is stored in at most $2\log n$ nodes of the tree.

Let us now try to implement this data structure in secondary storage. Given a block-size of $B$ it is easy to see that we require at least $t/B$ I/O's to output all the intervals. Also, it can be shown that we require $\Omega(\log_B n)$ I/O's to identify the path to $y$. Thus an ideal implementation of segment trees in secondary memory would require $O(\frac{n}{B}\log n)$ disk blocks of space and would answer stabbing queries with $O(\log_B n + t/B)$ I/O's.

To lower the time required to locate the root-to-$y$ path $P$ we can store the tree $T$ in a blocked fashion by mapping subtrees of height $\log B$ into disk blocks. The resulting search structure is called the *skeletal B-tree* and is similar in structure to a $B$-tree (see Figure 2). With this blocking, and a searching strategy similar to B-trees we can locate a $\log B$-sized portion of $P$ with every I/O. If the cover-list of each node is stored in a blocked fashion (with $B$ intervals per block) then we could examine the cover-list $CL(x)$ of each node $x$ on $P$ and retrieve the intervals in $CL(x)$ $B$ at a time. A closer look reveals that this approach could result in a query time of $O(\log n + t/B)$. This is because even though we can identify $P$ in $O(\log_B n)$ time we still have to do at least $O(\log n)$ I/O's, one for each cover-list on the path (see Figure 3). These I/O's may be wasteful if the cover-lists contain fewer than $B$ intervals. To avoid paying the additional $\log n$ in the query time we need to avoid *wasteful* I/O's (ones that return fewer than $B$ intervals) as much as possible. In particular, if the number of wasteful I/O's are smaller than the number

[2] A half-open interval $[a, b)$ contains all the points between $a$ and $b$ including $a$ but excluding $b$.

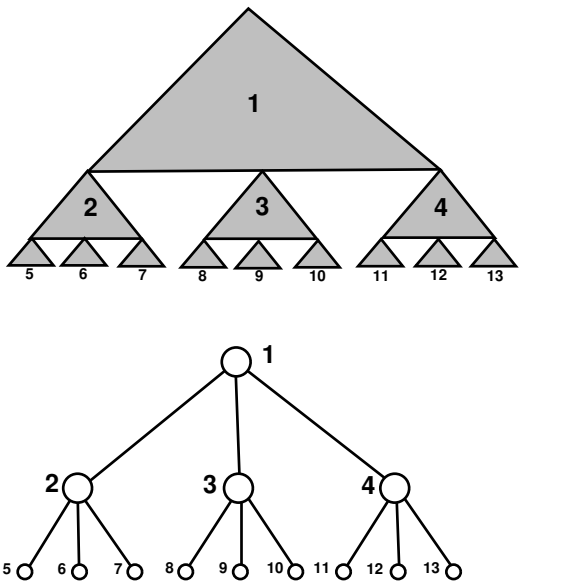


Figure 2: Constructing the skeletal graph

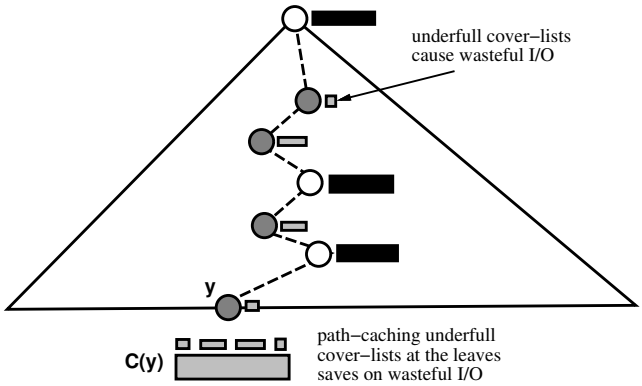


Figure 3: Underfull cover-lists along the query path result in wasteful I/O's. Path-caching alleviates this problem.

of *useful* I/O's (ones that return $B$ intervals) then the I/O's required to report all the intervals on $P$ will be $\leq 2t/B = O(t/B)$[3]. This combined with fact that we need $O(\log_B n)$ time to identify $P$ would give us the desired query time.

Consider a node $x$ on $P$ such that its cover-list has at least $B$ intervals. It is then easy to see that the first $I/O$ at node $x$ will be useful. In fact all but the last I/O at node $x$ will return $B$ intervals. This implies that the number of wasteful I/O's at node $x$ is upper-bounded by the number of useful I/O's at that node. Thus the problem nodes are only those that have fewer than $B$ intervals stored at them.

This leads us to the idea of **path caching:** *If we coalesce all the cover-lists on $P$ that have $\leq B$ elements and store it in a cache $C(y)$ at $y$ then we could look at $C(y)$ instead of looking at $\log n$ possibly underfull cover-lists.* If $C(y)$ is stored in a blocked fashion then retrieving intervals from it would cause at most one wasteful I/O instead of $\log n$ wasteful ones (see Figure 3). The time for reporting all the intervals would then be $\leq 2t/B + 1$. This combined with the time for finding $P$ would give us the desired query time.

We therefore make the following modification to the segment tree:
*For each leaf $y$ identify the underfull cover-lists $CL_1, \cdots CL_k$ (cover-lists that contain less than $B$ intervals) along the root-to-$y$ path. Make copies of the intervals in all the underfull cover-lists and store it in a cache $C(y)$ in $y$. Block $C(y)$ into blocks of size $B$ on to secondary memory.*

From the above discussions we can see that using this modified version of a segment tree we can answer stabbing queries with $O(\log_B n + t/B)$ I/O's. The only thing left to analyze is the amount of storage required for the modified data structure. The number of disk blocks required to block the search tree $T$ is $O(n/B)$. The total number of intervals in all the cover-lists is $O(n \log n)$. These can be stored in $O(\frac{n}{B} \log n)$ disk blocks. At each leaf $y$ we have a cache $C(y)$ that contains up to $B \log n$ intervals (from the $\log n$ nodes along the root-to-$y$ path). Therefore to store the caches from the $2n$ leaves we need $2n \log n$ disk blocks. Putting all of this together we see that the space required is $O(n \log n)$ blocks.

The space overhead can be reduced to the optimal value $O(\frac{n}{B} \log n)$ by performing two optimizations: (1) Building caches at the leaf nodes of the skeletal tree instead of the complete binary tree (we therefore have to build only $O(n/B)$ path-caches); and (2) By requiring the query to look at $O(\log_B n)$ path-caches instead of just one (this would allow us to build smaller path-caches). We can get a secondary memory implementation of the segment tree that requires $O(\frac{n}{B} \log n)$ space and answers queries with $O(\log_B n + t/B)$ I/O's.

## 3 Priority search trees and path caching

In this section, we apply the idea of path caching to priority search trees and use them to solve special cases of 2-dimensional range searching. We first consider 2-sided 2-dimensional queries that are bound on two sides and free on the other two as shown in Figure 1. As before, $n$ will indicate the number of data items, $B$ the size of the disk page, and $t$ the number of items in the query result.

Let us consider the implementation of priority search trees in secondary memory for answering 2-sided queries. The input is a set of $n$ points in the plane. The priority search tree is a combination of a heap and balanced bi-

[3] In other words each wasteful I/O will be *paid for* by performing a useful one.

nary search tree. The solution proposed in [IKO] works as follows: Find the top $B$ points (on the basis of their $y$ values) in the input set, store them in a disk block and associate this disk block with the root of the search tree. Divide the remaining points into two equal sets based on their $x$ values. From these two sets, choose the top $B$ points in each set and associate them with the two children of the root. Continue this division recursively. The skeletal structure for the binary tree itself is stored in a B-tree (called the skeletal B-tree). It is clear that to store $n$ points in this fashion, we will use only $O(n/B)$ disk blocks.

As illustrated in Figure 4, each node in the priority search tree defined above corresponds to a rectangular region in the plane that contains all the points stored in that node. Furthermore, the tree as a whole defines a hierarchical decomposition of the plane. As is shown in [IKO], this organization has the following crucial property: A point in a node $x$ can belong in a 2-sided query if (1) the region corresponding to $x$'s parent is completely contained within the query or, (2) the region corresponding to $x$ or the one corresponding to its parent is cut by the left side of the query.

With this division, we can show that a 2-sided query with $t$ points in the output can be answered by looking at only $O(\log n + t/B)$ disk blocks. To do that, we classify nodes that contain points inside the query into four categories as follows:

- *The corner:* This is the node whose region contains the corner of the query.
- *Ancestors of the corner:* These are nodes whose regions are cut by the left side of the query and there can be at most $O(\log n)$ such nodes.
- *Right siblings of the corner and the ancestors:* These are nodes whose parents' regions are cut by the left side of the query. There can be at most $O(\log n)$ such nodes.
- *Descendants of right siblings:* There can be an unbounded number of them, but for every such node, its parent's region has to be completely contained inside the query. That pays for the cost of looking into these nodes. That is, for every $k$ descendant blocks that are partially cut by the query, there will be at least $\frac{k}{2}$ blocks that lie completely inside the query.

The algorithm proceeds by locating the nodes intersecting the left side of our query. This is done by performing a search on the skeletal B-tree. The nodes are examined to find the points inside the query. Next, right siblings of these nodes and their descendants are examined in a top-down fashion until the bottom boundary of the query is crossed. In this algorithm, for each node examined, we perform one I/O operation. The corner, ancestor, and sibling nodes can cause wasteful I/O's but there are at most $O(\log n)$ such nodes. For every descendant of a "sibling" that is examined, its parent would have contributed an useful I/O. From this analysis, we can conclude that we can answer 2-sided queries in $O(\log n + t/B)$ I/O's.

We now show how to avoid the $\log n$ wasteful I/O's by caching the data in the ancestor and sibling nodes. We store two caches associated with the corner. One cache will contain all the data in the ancestors sorted in right-to-left (largest $x$ value first) fashion. Call this cache the $A$-list. The second cache will contain all the data in the siblings stored in top-to-bottom (largest $y$ value first) fashion. Call this cache the $S$-list. Using these caches we can answer 2-sided queries in $O(\log_B n + t/B)$ I/O's. In order to answer a 2-sided query, we simply look at the skeletal B-tree and locate the corner in $O(\log_B n)$ time. We then look at the caches (performing at most two wasteful I/O's) to determine which points from the ancestors/siblings fall into the query. After this, we look into the descendants if necessary. As discussed above, any wasteful query that is caused by examining a descendant can be counted off against a useful query that is the result of examining its parent. The descendants thus pay for looking into them through their parents. The following lemma follows.

**Lemma 3.1** *Given $n$ input points on the plane, the data structure described above answers any 2-sided query containing $t$ points using $O(\log_B n + t/B)$ I/O's. The storage used is $O(\frac{n}{B}\log n)$ disk blocks of size $B$ each.*

Now, we show that we can bring the storage overhead down to $O(\frac{n}{B}\log B)$. We do this by observing that maintaining caches of size $O(\log n)$ at each node is wasteful. We cut the total path length of $\log n$ into $\log_B n$ segments of size $\log B$. We maintain $A$-lists and $S$-lists at each node as before. However, to construct these lists at a node we only examine the ancestors and siblings that are in the $\log B$ segment of the root-to-node path that the node belongs to. Thus the lists at any node contain at most $O(\log B)$ disk blocks. Therefore the total storage required comes down to $O(\frac{n}{B}\log B)$. To answer a query, we now have to look at a total of $\log n/\log B = \log_B n$ $A$-lists and $S$-lists(one for each of the $\log B$-sized subpaths). The descendants of siblings are handled as they were in the previous construction. We get the following theorem.

**Theorem 3.2** *Given $n$ input points on the plane, path caching can be used to construct a data structure that answers any 2-sided query using $O(\log_B n + t/B)$ I/O's. Here $t$ is the output size of the query. The data structure requires $O(\frac{n}{B}\log B)$ disk blocks of storage.*

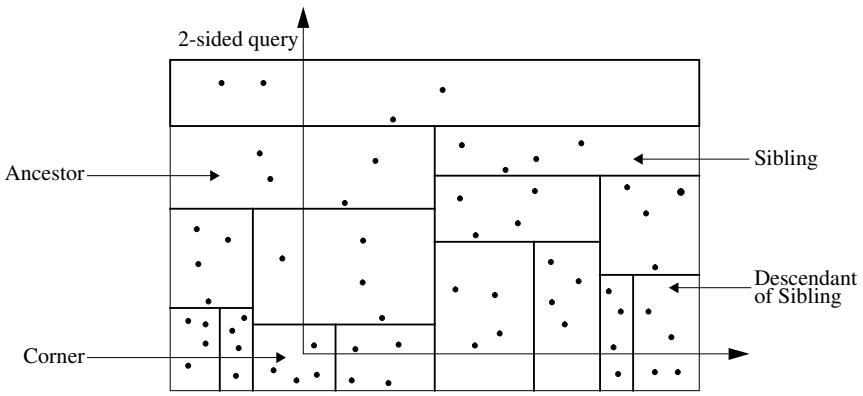


Figure 4: Binary tree implementation of Priority Search Tree in secondary memory showing corner, ancestor, sibling and sibling's descendant. Here, $B$ is 4.

In the next section, we show how we can combine the idea of path caching with recursion to make the storage required even less while keeping queries efficient.

Using similar ideas, we can obtain the following bounds for 3-sided queries, segment trees and interval trees (details will be given in the full version of the paper).

**Theorem 3.3** *Given $n$ input points on the plane, path caching can be used to construct a data structure that answers any 3-sided query using $O(\log_B n + t/B)$ I/O's. Here $t$ is the output size of the query. The data structure requires $O(\frac{n}{B}\log^2 B)$ disk blocks of storage.*

**Theorem 3.4** *We can implement Segment Trees in secondary memory so that a point enclosure query containing $t$ intervals can be answered in $O(\log_B n + t/B)$ I/O's. For $n$ input intervals, the storage used is $O(\frac{n}{B}\log n)$ disk blocks.*

**Theorem 3.5** *We can implement Interval Trees in secondary memory so that a point enclosure query containing $t$ intervals can be answered in $O(\log_B n + t/B)$ I/O's. For $n$ input intervals, the storage used is $O(\frac{n}{B}\log B)$ disk blocks.*

In the following sections, we explore 2-sided queries in more detail to improve the bounds obtained in this section. We then discuss algorithms for updating the resulting data structures.

## 4 Using recursion to reduce the space overhead

In this section we describe how to extend the ideas of section 3 to develop a recursive data structure that has a much smaller space overhead and still allows queries to be answered in optimal time. Due to space limitations we restrict our ourselves to the problem of answering general 2-sided queries by using a secondary memory priority search tree. Similar ideas can be used to get better space overheads for the other data structures as well.

We first describe a two level scheme for building a secondary memory priority search tree that requires only $O(\frac{n}{B}\log\log B)$ storage while still admitting optimal query performance. We then briefly describe a multi-level version of this idea that requires only $O(\frac{n}{B}\log^* B)$ storage.

Recall that the basic scheme divides the points into regions of size $B$. A careful look at this scheme shows that the $\log B$ overhead is due to the fact that the ancestor and sibling caches of each of the $n/B$ regions can potentially contain $\log B$ blocks. To reduce the space overhead we could either (1) reduce the amount of information stored in each region's cache or (2) reduce the total number of regions.

A closer look shows that to get optimal query time, with any region, we must store the information associated with $\log B$ of its ancestors. This is because in the priority tree structure the path length from any block to the root is $O(\log n)$. Thus to achieve a query overhead of at most $\log_B n$ we must divide such a path into no more than $\log_B n$ pieces. Since $\log_B n = \log n/\log B$, we see that with each node we must store the information associated with $O(\log B)$ of its ancestors. We therefore turn to the second idea. To get a linear space data structure we build a basic priority search tree that divides the points into regions of size $B\log B$ instead of $B$. We thus have $n/B\log B$ regions.

To build the caches associated with each of the regions we proceed in a slightly different fashion. First, we sort the points in each region $R$ right-to-left (i.e. largest to smallest) according to their $x$-coordinates. We store these points ($B$ at a time) in a list of disk blocks associated with $R$. In the same fashion we also sort the points top-to-bottom (i.e. largest to smallest) and store them in a list of disk blocks associated with $R$.

Thus the points in each region are blocked according to their $x$ as well as their $y$ coordinates. These lists are called the $X$-list and the $Y$-list of $R$ respectively.

To build the ancestor cache associated with a region $R$ we look at its $\log B$ immediate ancestors. From each of the ancestor's $X$-list we copy the points in the first block. We then sort all these points right-to-left according to their $x$-coordinates and store them in a list of disk blocks associated with region $R$. These blocks constitute the ancestor list ($A$-list) of $R$. Similarly, to build the sibling cache ($S$-list) of $R$ we consider the first blocks from the $Y$-lists of the siblings of $R$ and its ancestors. Adding up the all the storage requirements we get the following lemma.

**Lemma 4.1** *The total storage required to implement the top-level priority search tree and the associated $A$, $S$, $X$, and $Y$ lists is $O(\frac{n}{B})$.*

Unlike in the basic scheme we now have regions which contain $O(B \log B)$ points or $O(\log B)$ disk blocks. To complete our data structure we therefore build secondary level structures for each of these regions. For each region we build a priority search tree as per Lemma 3.1. In other words, we divide the region into blocks of size $B$ and build ancestor and sibling caches (as before) for each of the blocks. In this case the height of any path is at most $O(\log \log B)$; therefore for each region we use all of its ancestors and the siblings to construct the ancestor and sibling caches.

We now count the space overhead incurred due the priority search trees built for each region. For each block in a given region $R$ we need no more than $O(\log \log B)$ disk blocks to build its ancestor and sibling caches. This follows from the fact that the priority search tree of $R$ has height $O(\log \log B)$. A given region $R$ has $O(\log B)$ disk blocks; therefore the space required for storing the ancestor and sibling caches of all the blocks in $R$ is $O(\log B \log \log B)$. Adding this over all the regions we get the following lemma.

**Lemma 4.2** *The total storage required by the two level data structure is $O(\frac{n}{B} \log \log B)$.*

### 4.1 Answering queries using the two-level data structure

We now show how to answer 2-sided queries using this data structure. To answer such a query we proceed as follows: As in the basic scheme we first determine the region $R$ (in the top level priority search tree) that contains the corner of the query. As discussed in section 3, the points in the query belong to either the region $R$, or one of $R$'s ancestors $Q$, or a sibling $T$ of $Q$ (or $R$), or to a descendant of some sibling $T$.

To find the points that are in the ancestors and their siblings we look at the $\log_B n$ ancestor and sibling caches along the path from $R$ to the root. From these caches we collect all the points that lie inside the query. However, just looking at the these two caches is not enough to guarantee that we have collected all the points from the ancestors and their siblings. This is because we only use one block from each ancestor (and each sibling) to build the $A$ and $S$-lists.

To collect the other points in these regions that are in our 2-sided query we examine the $X$ and $Y$-lists of the ancestors and their siblings respectively. These lists are examined block by block until we reach a block that is not fully contained in our query. The $X$-list of an ancestor $Q$ of $R$ is examined if and only if all the points from $Q$ that were in the ancestor cache of $R$ are found to be inside the 2-sided query. Similarly the $Y$-list of a sibling $T$ (along the path from $R$ to the root) is examined if and only if the points from $T$ that are in the sibling cache are all inside our 2-sided query.

We claim that this algorithm will correctly find all points in the ancestor and the sibling regions that are in our query. We now show that all the points in the ancestor regions are found correctly. The case for the siblings of the ancestors is similar. Consider some ancestor region $Q$ of $R$ and its associated right-to-left ordering of the points as represented in its $X$-list. Since $Q$ is an ancestor, it is cut by the vertical line of the 2-sided query (see Figure 4). Therefore, all the points in $Q$ that are to the right of the vertical line are in the query and the rest aren't. Thus, all the points in the query are present in consecutive disk blocks (starting from the first one) in the $X$-list of $Q$. We therefore need to examine the $i$th block in the $X$-list if and only if all the previous $i-1$ blocks are completely contained in our query. Since the first block is part of $R$'s ancestor cache we need to look at the $X$-list of $Q$ if and only if all of the points of $Q$ that are in the ancestor cache of $R$ are contained in the 2-sided query.

To account for the time taken to find these points we note that there are $O(\log_B n)$ caches that we must look at. It is not hard to see that apart from these initial lookups we only look at a disk-block from a region $Q$ if our last I/O yielded $B$ points (inside the query) from this region. Therefore all the other I/O's are paid for. Thus the time to find these points is $O(\log_B n + (t_A + t_S)/B)$ where $t_A$ and $t_S$ denote the number of points contained in the ancestors and their siblings.

To find the points in the query that are in the descendants of the siblings, we use the same approach as the basic scheme. We find the points in all these regions by scanning their $Y$-lists. We traverse a region $Q$ if and only if its parent $P$ is fully contained in the query. An argument similar to the one above shows that all such points are found by this algorithm, and that the number of I/O's required is $O(t_D/B)$. Here $t_D$ denotes the number of points contained in the descendants of

the siblings.

To find the points in the query that are in region $R$ we use the second level priority tree associated with $R$. Let $t_R$ denote the number of points in $R$ that belong to the query. We find these points by asking a 2-sided query inside region $R$. This requires at most $O(t_R/B)$ I/O's (by the arguments in section 3).

Therefore, the total number of I/O's required to answer a two sided query is $O(\log_B n + t/B)$ where $t$ is the size of the output. This in conjunction with Lemma 4.2 gives us the following theorem.

**Theorem 4.3** *There exists a secondary memory static implementation of the priority search tree that can be used to answer general 2-sided queries using $O(\log_B n + t/B)$ I/O's; where $t$ is the size of the output. This data structure requires $O(\frac{n}{B}\log\log B)$ disk blocks of space to store $n$ points.*

### 4.2 A multilevel scheme to further lower the space over-head

It is possible to reduce the space overhead further by using more than two levels. The idea is the same as before. At the second stage instead of building a basic priority search tree for each region we build a tree that contains regions of size $B\log\log B$ and build the $X$, $Y$, $A$, and $S$ lists same as before. A three level scheme gives us a space overhead of $O(\frac{n}{B}\log\log\log B)$ while maintaining optimum query time. If we carry this multilevel scheme further then we get a data structure with the following bounds.

**Theorem 4.4** *There exists a secondary memory static implementation of the priority search tree that can be used to answer general 2-sided queries using $O(\log_B n + t/B + \log^* B)$ I/O's; where $t$ is the size of the output. This data structure requires $O(\frac{n}{B}\log^* B)$ disk blocks of space to store $n$ points.*

These ideas can be applied to 3-sided queries as well, to reduce the space overhead incurred by the sibling caches. In particular we can get the following bounds for answering 3-sided queries.

**Theorem 4.5** *There exists a secondary memory static implementation of the priority search tree that can be used to answer general 3-sided queries using $O(\log_B n + t/B + \log^* B)$ I/O's; where $t$ is the size of the output. This data structure requires $O(\frac{n}{B}\log B\log^* B)$ disk blocks of space to store $n$ points.*

## 5 A fully dynamic secondary memory data structure for answering 2-sided queries

In this section we show how to dynamize the two-level scheme discussed in section 4. Our data structure is fully dynamic, i.e. it can handle both additions and deletions of points. The amortized time bound for an update is $O(\log_B n)$. In this abstract we only give a brief overview of the dynamization. The details will be given in the full paper.

Before we describe our dynamic data structure we first discuss an alternate way of visualizing the top level priority search tree in the two level scheme. In this tree a node corresponds to a region of size $B\log B$. We partition this priority search tree into subtrees of height $\log B - \log\log B$. Each such subtree is considered a *super node*. As in section 3, in order to build the ancestor and sibling cache of any region $R$, we only consider those ancestors (and their siblings) of $R$ that are in the same super node as $R$. Considering subtrees of height $\log B - \log\log B$ (instead of $\log B$) does not change the query times because we are now dealing with regions of size $B\log B$.

The advantage of viewing these subtrees as super nodes is that now we can isolate the duplication of information due to the $S$ and $A$-lists to within a super node, since none of the caches ever cross the super node boundary. The layer of super nodes (regions) immediately following the last layer in a super node $N$ can be thought of as children of $N$ in this visualization. Note that each supernode $N$ contains $B/\log B$ regions and has $O(B/\log B)$ super nodes as its children. Also note that the number of super nodes on any path in the tree is $O(\log_B n)$.

Our dynamic data structure associates an update buffer $U$ of size $B$ with each super node $N$ in the tree. It also associates an update buffer $u$ (also of size $B$) with each region $R$. These buffers are used to store incoming updates until we have collected enough of them to account for rebuilding some structure. To process a query, we first use the algorithm from section 4 to collect the points that have been entered into the data structure. We then look through the associated update buffers to add new points that have been added and to discard old points that have been deleted by an unprocessed update.

We now need to be careful in claiming optimal query time because the points that we collect by searching the priority search structures may then have to be deleted when we look at the respective update buffers. However, it can be shown that for every $B\log B$ points we collect we can lose at most $B$ points, thus resulting in at most two wasted I/O's for $\log B$ useful ones. Therefore, the loss of points due to unprocessed deletes is very small and does not affect the overall query performance.

Whenever an update occurs, we first locate the super node $N$ where the update should be made. The update could be a point insertion or a deletion. We then log the update in the associated update buffer $U$. If the buffer does not overflow we don't do anything. If $U$ overflows

then we take all the updates collected and propagate them to the regions in $N$ where they should go. For instance, a point insertion is trickled down to the region of $N$ that contains its coordinates. In each such region we then log the update into the local update buffer $u$ associated with it. We now rebuild the $X$ and $Y$ lists of each region in $N$ taking into account the updates that have percolated into it. We also rebuild each region's ancestor and sibling caches; again taking into account the updates that have percolated into that region.

The number of I/O's required to rebuild the $A$, $S$, $X$, and $Y$ lists of one region $R$ is $O(\log B)$. Therefore, the number of I/O's required to rebuild all the caches is $(B/\log B)\times O(\log B) = O(B)$. Since we do this only once in $B$ updates the amortized cost of rebuilding the caches is $O(1)$ per update.

If for none of the regions in $N$ any of their buffers overflow, we don't do anything further. Otherwise for each region $R$ whose update buffer has overflowed we rebuild the second level priority search tree associated with it. As before, we take into account the updates in the buffer $u$ of $R$.

The number of I/O's required to rebuild the priority search tree associated with a given region $R$ is $O(\log B \log\log B)$. This is because we need to rebuild the caches of $\log B$ blocks each of which could contain up to $\log\log B$ disk-blocks of information. Since this is done only once every $B$ updates the amortized time per update is $O((\log B \log\log B)/B) = O(1)$.

For every super node $N$; once every $B\log B$ updates we do a rebuild. This rebuilding keeps the same $x$-division as the old one. Keeping the $x$-divisions same we change the $y$-lines of the regions so that each region now contains exactly $B\log B$ points. Using these new regions structure we rebuild the $A$, $S$, $X$, and $Y$ lists for each region as well as the secondary level priority search trees associated with each region. Note that it is important to keep the same $x$-division to preserve the underlying binary structure of the priority search tree. We cannot view the regions in a given supernode in isolation since they are part of a bigger priority search tree.

To keep the invariant that each region in $N$ contain $B\log B$ points we may have to push points into its children or we may have to borrow points from them. These are logged as updates in the corresponding supernodes. Pushing points into a node is equivalent to adding points to that region while borrowing points from a node is the same as deleting points from the region. These updates may then cause an overflow in the buffers associated with one or more of those supernodes. We repeat the same process described above with any such supernode.

It is easy to show that the number of I/O's required to rebuild a super node is $O(B\log\log B)$. Since a rebuild is done once every $B\log B$ updates, the amortized time required for one such rebuild is $O(1)$. However, since we push updates down when we rebuild super nodes, we may have to do up to $\log_B n$ rebuilds (along an entire path) due to a single overflow. Therefore, the amortized cost of a rebuild is $O(\log_B n)$.

A moment of thought reveals that pushing points down is not enough to keep the priority search tree balanced. Repeated additions or deletions to one side can make subtrees unbalanced. We therefore periodically rebuild subtrees in the following manner.

With each node in the tree we associate a size which is the number of points in the subtree rooted at that node. We say that a node is unbalanced if the size of one of its children is more than twice the size of its other child. Whenever this happens at a node $R$ we rebuild the subtree rooted at $R$. The number of I/O's required to rebuild a priority search tree with $x$ points is $O((x/B)\log_B x + (x/B)\log\log B)$. This is because we need to rebuild the secondary level priority search trees as well as the primary level tree along with all the caches at both levels. Since a subtree of size $x$ can get unbalanced only after $O(x)$ updates we get an amortized rebuilding time of $O((\log_B x + \log\log B)/B) = O(1)$.

Summing up all the I/O's that are required to rebuild various things we see that the total amortized time for an update is $O(\log_B n)$. We therefore have the following theorem.

**Theorem 5.1** *There exists a fully dynamic secondary memory implementation of the priority search tree that can be used to answer general 2-sided queries using $O(\log_B n + t/B)$ I/O's; where $t$ is the size of the output. The amortized I/O-complexity of processing both deletions and additions of points is $O(\log_B n)$. This data structure requires $O(\frac{n}{B}\log\log B)$ disk blocks of space to store $n$ points.*

Similar ideas can be used to get a dynamic data structure for answering 3-sided queries as well. The time to answer queries is still optimal but the time to process updates is not as good. In particular we get the following bounds for answering 3-sided queries.

**Theorem 5.2** *There exists a fully dynamic secondary memory implementation of the priority search tree that can be used to answer general 3-sided queries using $O(\log_B n + t/B)$ I/O's; where $t$ is the size of the output. The amortized I/O-complexity of processing both deletions and additions of points is $O(\log_B n \log^2 B)$. This data structure requires $O(\frac{n}{B}\log B \log\log B)$ disk blocks of space to store $n$ points.*

## 6 Conclusions and open problems

Special cases of 2-dimensional range searching have many applications in databases. We have presented

a technique called path caching which can be used to implement many main memory data structures for these problems in secondary memory. Our data structures have optimal query performance at the expense of a slight overhead in storage. Furthermore, our technique is simple enough to allow inserts and deletes in optimal or near optimal amortized time.

There seem to be some fundamental obstacles to implementing many main memory data structures in secondary memory. We believe that studying space-time tradeoffs, as we have done, is important in understanding the complexities of secondary storage structures. The hope is that this will eventually help us develop efficient data structures that will provide good worst case bounds on querying as well as update times. As of today, we have to rely on heuristics—that may or may not perform well at all times—to handle many of these problems.

Specifically, the important problem of dynamic interval management that we highlighted in [KRV] remains open. Can we solve this problem optimally using $O(\frac{n}{B})$ storage, answering queries in $O(\log_B n + t/B)$ time, while being able to perform updates in $O(\log_B n)$ worst-case time?

**Acknowledgments:** We thank Paris Kanellakis for helpful discussions on this area.

## References

[BaM] R. Bayer and E. McCreight, "Organization of Large Ordered Indexes," *Acta Informatica* 1 (1972), 173–189.

[Ben] J. L. Bentley, "Algorithms for Klee's Rectangle Problems," Dept. of Computer Science, Carnegie Mellon Univ. unpublished notes, 1977.

[BlGa] G. Blankenagel and R. H. Güting, "XP-Trees - External Priority Search Trees," FernUniversität Hagen, Informatik–Bericht Nr. 92, 1990.

[BlGb] G. Blankenagel and R. H. Güting, "External Segment Trees," FernUniversität Hagen, Informatik–Bericht, 1990.

[ChT] Y.-J. Chiang and R. Tamassia, "Dynamic Algorithms in Computational Geometry," *Proceedings of IEEE, Special Issue on Computational Geometry* 80(9) (1992), 362–381.

[Cod] E. F. Codd, "A Relational Model for Large Shared Data Banks," *CACM* 13(6) (1970), 377–387.

[Com] D. Comer, "The Ubiquitous B-tree," *Computing Surveys* 11(2) (1979), 121–137.

[Edea] H. Edelsbrunner, "A new Approach to Rectangle Intersections, Part I," *Int. J. Computer Mathematics* 13 (1983), 209–219.

[Edeb] H. Edelsbrunner, "A new Approach to Rectangle Intersections, Part II," *Int. J. Computer Mathematics* 13 (1983), 221–229.

[Gün] O. Günther, "The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases," *Proc. of the fifth Int. Conf. on Data Engineering* (1989), 598–605.

[Gut] Antonin Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. 1984 ACM-SIGMOD Conference on Management of Data* (1985), 47–57.

[IKO] C. Icking, R. Klein, and T. Ottmann, *Priority Search Trees in Secondary Memory (Extended Abstract)*, Lecture Notes In Computer Science #314, Springer-Verlag, 1988.

[KKR] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz, "Constraint Query Languages," *Proc. 9th ACM PODS* (1990), 299–313.

[KRV] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter, "Indexing for Data Models with Constraints and Classes," *Proc. 12th ACM PODS* (1993), 233–243, (A complete version of the paper appears in Technical Report 93-21, Brown University.).

[KKD] W. Kim, K. C. Kim, and A. Dale, "Indexing Techniques for Object-Oriented Databases," in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. H. Lochovsky, eds., Addison-Wesley, 1989, 371–394.

[KiL] W. Kim and F. H. Lochovsky, eds., *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley, 1989.

[LoS] D. B. Lomet and B. Salzberg, "The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance," *ACM Transactions on Database Systems* 15(4) (1990), 625–658.

[LOL] C. C. Low, B. C. Ooi, and H. Lu, "H-trees: A Dynamic Associative Search Index for OODB," *Proc. ACM SIGMOD* (1992), 134–143.

[McC] E. M. McCreight, "Priority Search Trees," *SIAM Journal of Computing* 14(2) (1985), 257–276.

[NHS] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Transactions on Database Systems* 9(1) (1984), 38–71.

[Ore] J. A. Orenstein, "Spatial Query Processing in an Object-Oriented Database System," *Proc. ACM SIGMOD* (1986), 326–336.

[OSB] M. H. Overmars, M. H. M. Smid, M. T. de Berg, and M. J. van Kreveld, "Maintaining Range Trees in Secondary Memory: Part I:

Partitions," *Acta Informatica* 27 (1990), 423–452.

[Rob] J. T. Robinson, "The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proc. ACM SIGMOD* (1984), 10–18.

[Sama] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, 1989.

[Samb] H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, 1989.

[SRF] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The $R^+$-Tree: A Dynamic Index for Multi-Dimensional Objects," *Proc. 1987 VLDB Conference, Brighton, England* (1987).

[SmO] M. H. M. Smid and M. H. Overmars, "Maintaining Range Trees in Secondary Memory: Part II: Lower Bounds," *Acta Informatica* 27 (1990), 453–480.

[ZdM] S. Zdonik and D. Maier, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann, 1990.