Università di Pisa
Dipartimento di Informatica

Technical Report: TR-00-16

Revised version of "Efficient Cross-Trees for External Memory"

Roberto Grossi Giuseppe F. Italiano

November 10, 2000

ADDRESS: Corso Italia 40, 56125 Pisa, Italy. TEL: +39 050 887111 FAX: +39 050 887226

Revised version of “Efficient Cross-Trees for External Memory”

Roberto Grossi * Giuseppe F. Italiano †

November 10, 2000

Abstract

Due to a printing problem, the revised version of our paper [19] has been replaced by the (shorter) submitted version. In this technical report, we include the revised version that has not been published by mistake. In particular, we describe efficient methods for organizing and maintaining large multidimensional data sets in external memory. This is particular important as access to external memory is currently several order of magnitudes slower than access to main memory, and current technology advances are likely to make this gap even wider. We focus particularly on multidimensional data sets which must be kept simultaneously sorted under several total orderings: these orderings may be defined by the user, and may also be changed dynamically by the user throughout the lifetime of the data structures, according to the application at hand. Besides standard insertions and deletions of data, our proposed solution can perform efficiently *split* and *concatenate* operations on the whole data sets according to any ordering. This allows the user: (1) to dynamically rearrange any ordering of a segment of data, in a time that is faster than recomputing the new ordering from scratch; (2) to efficiently answer queries related to the data contained in a particular range of the current orderings. Our solution fully generalizes the notion of B-trees to higher dimensions by carefully combining space-driven and data-driven partitions. Balancing is easy as we introduce a new multidimensional data structure, *the cross-tree*, that is the cross product of balanced trees. As a result, the cross-tree is competitive with other popular index data structures that require linear space (including k-d trees, quad-trees, grid files, space filling curves, hB-trees, and R-trees).

*Dipartimento di Informatica, Università di Pisa, Italy (grossi@di.unipi.it). Supported in part by the Italian MURST project “Algorithms for Large Data Sets: Science and Engineering” and by the United Nations Educational, Scientific and Cultural Organization under contract UVO-ROSTE 875.631.9.

†Dipartimento di Informatica, Sistemi e Produzione, Università di Roma “Tor Vergata”, Italy (italiano@info.uniroma2.it). Partially supported by EU ESPRIT Long Term Research Project ALCOM-IT under contract no. 20244.

1 Introduction

Let S be a set whose items are sorted with respect to $d > 1$ total orders $\prec_1, \ldots, \prec_d$, and which is subject to dynamic operations, such as insertions and deletions of single items, split and concatenate operations performed according to any chosen order $\prec_i$ ($1 \leq i \leq d$). Let $\mathcal{P}$ be a searching problem defined on input set S with N items, and let $\mathcal{P}(x, S)$ denote its solution for a query item x. Problem $\mathcal{P}$ is *decomposable* if we can answer query $\mathcal{P}(x, S)$ by first partitioning set $S = S' \cup S''$ and computing the answers to queries $\mathcal{P}(x, S')$ and $\mathcal{P}(x, S'')$ recursively, and then combining them through a suitable associative operator $\Diamond$. Formally, $\mathcal{P}$ is said to be $f(N)$-*decomposable* if and only if $\mathcal{P}(x, S) = \Diamond(\mathcal{P}(x, S'), \mathcal{P}(x, S''))$ for any partition $S = S' \cup S''$ and any query item x, where $\Diamond$ is an operator whose computation requires $O(f(N))$ time. Throughout, we assume that $f(N)$ is a nondecreasing and smooth function, i.e., $f(\Theta(N)) = \Theta(f(N))$, and that $f(N) = o(N)$. Some simple examples of $O(1)$-decomposable searching problems include: membership queries, where $\Diamond$ is the logical or function; closest point queries, where $\Diamond$ is the minimal distance; range queries, where $\Diamond$ is the list append operation. Convex hull searching is not decomposable as the fact that a point $x \in S$ belongs to the convex hull of S' or S'' does not necessarily imply that x belongs to the convex hull of $S = S' \cup S''$. The definition of decomposable *search* problems can be extended also to the decomposable *set* problems in which the query item is not specified (e.g., finding the minimum weight item, where $\Diamond$ is the minimum). Thus, we shall denote a generic solution to a decomposable problem $\mathcal{P}$ by $\mathcal{P}(S)$.

In this paper, we consider *order decomposable problems*. A problem $\mathcal{P}$ is $f(N)$-*order* decomposable with respect to total order $\prec_i$ if $\mathcal{P}(S) = \Diamond(\mathcal{P}(S'), \mathcal{P}(S''))$ for any *ordered partition* $S = S' \cup S''$ (i.e., $x' \prec_i x''$ for all $x' \in S'$ and $x'' \in S''$), where operator $\Diamond$ takes $O(f(N))$ time. Problem $\mathcal{P}$ is $f(N)$-*order decomposable* if it is $f(N)$-order decomposable with respect to any total order $\prec_i$, $1 \leq i \leq d$. For example, performing multidimensional range queries is $O(1)$-order decomposable, and convex hull searching is $O(\log_2 N)$-order decomposable. Many other examples of order decomposable problems can be found in basic data structures, computational geometry, database applications and statistics [9, 29, 33].

In the *static* case, there is an obvious divide-and-conquer algorithm for these problems on a sorted set S: split S into two (equally sized) subsets S' and S'', solve the problem recursively on S' and S'', and combine their solutions in $O(f(N))$ time. We aim at maintaining a *dynamic* set S with d total orders, for constant $d > 1$, under insertions of a single item, deletions of a single item, and re-arrangements of any of the total orders $\prec_1, \ldots, \prec_d$ on S by means of split and concatenate operations. In several cases, executing splits and concatenates is faster than performing a batch of insertions and deletions to achieve the same goal. Our queries involve finding the solution $\mathcal{P}(R)$ for the items contained in the subset $R \subseteq S$ defined by some ranges in the orders $\prec_1, \ldots, \prec_d$.

In this paper, we deal with the following *multiordered set splitting and merging problem* introduced in [18]:

split$(S, z, \prec_i)$ Split S into S' and S'' according to item z and the specified total order $\prec_i$ $(1 \leq i \leq d)$. That is, $x' \prec_i z$ and $z \prec_i x''$ for all $x' \in S'$ and $x'' \in S''$. S is no longer available after this operation.

concatenate$(S', S'', \prec'_i, \prec''_i)$ Combine S' and S'' together according to their respective i-th total orders $\prec'_i$ and $\prec''_i$ $(1 \leq i \leq d)$ into a new set $S = S' \cup S''$. After this operation, S' and S'' are no longer available. The items in the resulting set S undergo the new order $\prec_i$ obtained by concatenating $\prec'_i$ and $\prec''_i$. That is, $x \prec_i y$ in S if and only if one of the following three conditions holds:

1. $x \prec'_i y$ and $x, y \in S'$;
2. $x \prec''_i y$ and $x, y \in S''$;
3. $x \in S'$ and $y \in S''$.

insert(z, S) Insert item z into set S according to all orders $\prec_1, \ldots, \prec_d$.

delete(z, S) Delete item z from set S.

range$(\langle a_1, b_1 \rangle, \ldots, \langle a_d, b_d \rangle, S)$ Let $R = \{z \in S : a_i \prec_i z \prec_i b_i, \text{ for } 1 \leq i \leq d\}$. Find solution $\mathcal{P}(R)$ to problem $\mathcal{P}$ restricted to the items in region R only.

For $d = 1$, the recursive nature of order decomposable problems gives an immediate tree structure, and each of the above operations can be simply implemented with $O(f(N) \log_2 N)$ time (with $O(f(N))$ time per tree node) by using a 2-3-tree [2] or a B-tree [6, 12]. However, the multidimensional version of this problem $(d > 1)$ is much more complicated: indeed, maintaining $d > 1$ total orders on the same set S, while splitting or concatenating each order independently of the others, makes things highly non-trivial due to the interplay among different orders.

In previous work [18] we introduced efficient techniques and data structures for this problem in main memory, and showed how these data structures could efficiently implement multidimensional priority queues, multidimensional search trees, and concatenable interval trees. These applications improved many previously known results on decomposable problems under split and concatenate operations, including membership query, minimum weight item, range query, and convex hulls.

Our Results. The main result of this paper is to make the main memory technique of [18] suitable for efficient external memory implementation. To achieve this goal, we have to prove and exploit other features and properties of the technique introduced in [18], and implement it differently. This produces an external memory technique for solving order decomposable problems on S under insertions, deletions, splits, concatenates and range queries, yielding new and efficient concatenable multidimensional data structures. Differently from other approaches, our technique is based more on simple geometric properties rather than on underlying sophisticated data structures, and exploits the fact that some data structures can be built on (pre)sorted items more efficiently. Balancing is easy as we introduce a new multidimensional data structure, *the cross-tree*, that is the cross product of balanced trees.

The model we consider for external memory is quite standard [1, 44]: it consists of a fast memory of size M, and a number D of slow independent disks. Data transfers between disks and internal memory occur in *blocks* of size B, each single data transfer accessing a single block per disk in parallel. It is assumed that the fast memory is large enough to accommodate items from the disks, e.g., $M \geq 2BD$. The objective is to minimize the number of transferred blocks, called *inputs/outputs or I/Os*, performed by an algorithm or required to access a data structure. A single I/O handles BD items, one block of B items per disk. The I/O complexity is typically expressed in terms of the input size and the "machine" parameters B, M and D. For example, the I/O complexity of sorting N constant size keys in external memory is $Sort(N) = \Theta(\frac{N}{BD} \log_{M/B} \frac{N}{B})$ I/Os [1, 31]. The general problem of sorting variable length keys in external memory is discussed in [3]. For the sake of presentation, we will assume here that the number of disks is $D = 1$; however, our results smoothly extend to $D > 1$ parallel disks by using the disk striping technique [37] with a block size of $B' = BD$ and $D' = 1$.

In the simple case $d = 1$, B-trees [6, 12] are very popular data structures that can be successfully employed in decomposable search problems analogously to concatenable 2-3-trees [2]. For higher dimensions ($d > 1$), no provably good external memory data structures for splitting and concatenating along any dimension were previously known in the literature. We show how to maintain S with the following I/O bounds: $O(\log_B N)$ for the insertion or the deletion of a single item, where N is the number of items currently in S; $O((N/B)^{1-1/d})$ for splits and concatenates along any order; $O((N/B)^{1-1/d}))$ plus an optimal output sensitive cost for rectangular range queries, i.e., an $O(r/B)$ cost for retrieving r *pointers* to the items. This is helpful when processing the output lists subsequently, e.g., by answering to a boolean query. Furthermore, we get an optimal cost when actually scanning the pointed items as there is *no replication of data*. Namely, if each item occupies b bytes, we require optimal $O(br/B)$ I/Os to scan the content of all retrieved items. This is not always possible with the known multidimensional data structures as pointed out in [24].

We require that our external memory algorithms must use optimal storage. In our case, the space required is linear, i.e., $O(N/B)$ disk blocks. In general, we have to face some problems that do not appear in internal memory, especially when we have to maintain dynamically data structures with optimal storage. Consider for instance the simple case of moving an item u of indegree h from its block to a different block. The h incoming pointers could be in different blocks and, when no specific block strategy is adopted, moving u might cause the explicit update of all these pointers. If h is not bounded a priori, this single update becomes costly as we have to pay $\Theta(h)$ I/Os. More details on this problem can be found in [14].

Related Work. There is a great deal of work on decomposable searching problems. They were introduced by Bentley [8] for dynamizing static data structures. The initial goal was to support insertions with low amortized times, without affecting much of the query efficiency. Other dynamization techniques were subsequently given (see [11, 33] for many references to the literature). The main idea behind these techniques is to partition a big data structure into a collection of small data structures, called *blocks*, and to tune properly the number of blocks in order to obtain a good tradeoff between queries and updates. Later on, van Kreveld and Overmars [26, 27] introduced two new operations to this dynamic setting: split and concatenate. Namely, they showed how to maintain a set of multidimensional points subject to the following repertoire of operations: range queries, insertion of a point, deletion of a point, split and concatenate along one coordinate.

In external memory, rectangular range queries on ordered decomposable problems can be done with quad-trees [15], k-d trees [7] and many other data structures, such as grid files [30], space filling curves, e.g. [32], hB-trees [28], and R-trees [20] just to cite a few (see the survey [16] for a more complete scenario and paper [24] for a discussion of their worst-case complexity). These data structures were originally designed to support some operations for windowing problems in computer graphics and databases, and in several cases it was rather difficult to keep them balanced [35, 40]. In contrast, our cross-tree can be easily kept balanced. Many other powerful data structures for range queries were devised subsequently and we refer the reader to [11] for a comprehensive survey on this topic and a list of references. Recently, some elegant data structures [5, 24, 36, 38, 42, 43] were devised to support fast range queries in external memory, and Arge *et al.* [4] have dealt with some decomposable problems in external memory. However, none of these data structures seems to be able to support efficiently split and concatenate along any coordinate.

Ravi and Singh [39], following up on a previous lower bound of Hellerstein *et al.* [22], considered the complexity of range searching in a simple index tree model. In this model, they proved a lower bound of $\Omega((N/B)^{1-1/d} + r/B)$ for a range query retrieving the pointers to r out of N items, where B is the capacity of index nodes. This lower bound holds on pointer machines and for non-replicating index structures, which occupy linear space. Our range query matches this lower bound, and uses simply the list append operator as operator $\Diamond$: consequently, it can be easily implemented on a pointer machine. As cross-trees use some auxiliary pointers for supporting efficiently split and concatenate operations, they are not strictly considered non-replicating in the terminology of Ravi and Singh: however, we do not believe this to be a substantial infringement of the non-replicating assumption. Ravi and Singh [39] introduced also the O-tree, a pointer-based strictly non-replicating data structure which also matches their lower bound. Once again, O-trees support only queries, insertions and deletions but are not able to support efficiently split and concatenate operations.

The remainder of this paper consists of seven sections. In Section 2 we present our data structure, which consists of two levels. The first level operates a space-driven partition of the points into basic squares and is described in more details in Section 3. The second level, considered in Section 4, is based upon a data-driven multilevel hierarchical partition. The two levels are next combined in Section 5. Section 6 illustrates how the overall data structure can be used to implement the operations, while Section 7 mentions some further applications of our techniques. Section 8 discusses the multidimensional case, and Section 9 contains some concluding remarks.

2 Our Data Structure

Similarly to [18], we define a two-level data structure: the *lower* level data is only locally kept sorted, while the *upper* level data follows a hierarchical and global order. Specifically, the lower level organizes the embedding space of data items according to their "spatial" coordinates so as to facilitate their location search. This is done by partitioning the space into a collection of disjoint *basic squares*, forming a flexible grid. Any row or column formed by basic squares needs only local changes to preserve a global notion of balancedness. Order inside a basic square is strictly not a requirement of our data structures, but can be maintained efficiently. The upper level consists of a multidimensional tree, called the *cross-tree*, whose leaves are in one-to-one correspondence with the basic squares of the lower level. This upper level organizes the data items according to their "relative" coordinates in the orderings, by exploiting their value comparisons. Contrarily to k-d trees [7], quad-trees [15] and their variants in external memory, cross-trees can be easily and dynamically kept balanced by means of node splits and merges as they are "cross product" of one-dimensional balanced trees. In other words, cross-trees extend the natural notion of balance to higher dimensions: *in order to update a cross-tree, it is sufficient to work on one of the (one-dimensional) balanced trees and broaden its updates to the whole cross-tree.*

The power of our data structure for external memory lies in this two-level organization: we can run a fast search in the upper level at the price of an increased update cost, while we can run a fast update in the lower level at the price of an increased search cost. The resulting good performance is obtained by a proper trade-off between total search and update costs and by carefully tuning some parameters that bound the size of the two levels. Differently from other approaches, our technique is based more on simple geometric properties rather than on underlying sophisticated data structures, and exploits the fact that some data structures can be built on (pre)sorted items more efficiently. As a result, our data structure may be particularly attractive as a multidimensional index tool for many kinds of terabyte applications. An example is shown in Figure 1.

For the sake of simplicity, we start out by describing our data structure in the case of $d = 2$ total orders, which we denote by $\prec_X$ and $\prec_Y$. We will see later on (Section 8) how to extend this to the more general case. Let N be the number of items in S, where $N > M$. Each item $z \in S$ is associated with a point $(X(z), Y(z))$ in the Cartesian plane, such that $X(z)$ is the rank of z in S with

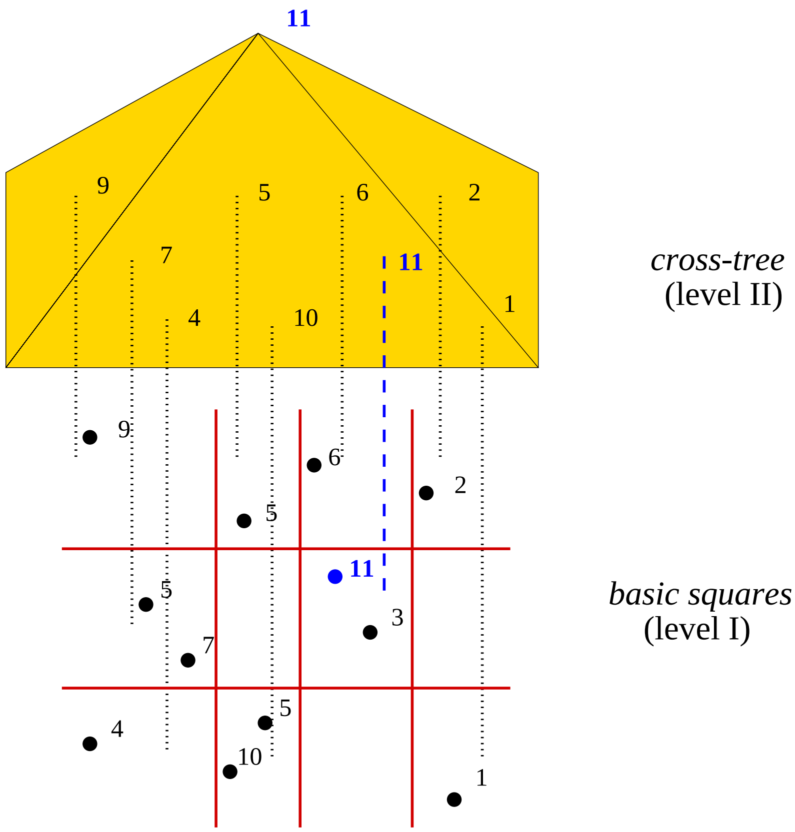


Figure 1: An example of the two-level data structure for maintaining the solution $\mathcal{P}(S)$ to the decomposable problem "find the maximum weight element in S" with $\diamondsuit = \max$ and $d = 2$. The cross-tree is shown in detail in Figures 2–3.

respect to order $\prec_X$ and $Y(z)$ is the rank of z in S with respect to $\prec_Y$. Note that $X(z)$ and $Y(z)$ are affected by our dynamic operations and so the mapping from z onto point $(X(z), Y(z))$ changes dynamically throughout the sequence of operations. Starting from N items in S, we obtain N points in the Cartesian plane, which can be stored in the form of a $N \times N$ *sparse* and *dynamic* matrix $\mathcal{M}$. We use $\mathcal{M}$ as it is more suitable for external memory implementation. The operations in S can be simulated by a certain number of operations in $\mathcal{M}$ by suitably using the ranks of the involved items. For any integers h_1, h_2, v_1, v_2 (with $1 \leq h_1 \leq h_2 \leq N$ and $1 \leq v_1 \leq v_2 \leq N$), we use $\mathcal{M}[h_1, h_2; v_1, v_2]$ to denote the submatrix of $\mathcal{M}$ that contains entries $\mathcal{M}[i, j]$ with $h_1 \leq i \leq h_2$ and $v_1 \leq j \leq v_2$. We call this submatrix a *region*. The operations defined on the matrix $\mathcal{M}$ are:

h_split$(\mathcal{M}, i)$ Split $\mathcal{M}$ horizontally at row i and obtain two new matrices $\mathcal{M}_1$ and $\mathcal{M}_2$, such that $\mathcal{M}_1 = \mathcal{M}[1, i; 1, N]$ and $\mathcal{M}_2 = \mathcal{M}[i + 1, N; 1, N]$. In other words, $\mathcal{M}_1$ is given by the first i rows of $\mathcal{M}$ and $\mathcal{M}_2$ is given by the last $(N-i)$ rows of $\mathcal{M}$. $\mathcal{M}$ is no longer available after the operation.

h_concatenate$(\mathcal{M}_1, \mathcal{M}_2)$ Let $\mathcal{M}_1$ have size $m_1 \times N$ and $\mathcal{M}_2$ have size $m_2 \times N$. We meld $\mathcal{M}_1$ and $\mathcal{M}_2$ horizontally and produce a matrix $\mathcal{M}$ of size $(m_1 + m_2) \times N$, such that $\mathcal{M}[1, m_1; 1, N] = \mathcal{M}_1$ and $\mathcal{M}[m_1+1, m_1+m_2; 1, N] = \mathcal{M}_2$. In other words, the first m_1 rows of $\mathcal{M}$ are given by $\mathcal{M}_1$ and the last m_2 rows of $\mathcal{M}$ are given by $\mathcal{M}_2$. This operation assumes that $\mathcal{M}_1$ and $\mathcal{M}_2$ have the same number of columns. $\mathcal{M}_1$ and $\mathcal{M}_2$ are no longer available after the operation.

v_split$(\mathcal{M}, j)$ and *v_concatenate*$(\mathcal{M}_1, \mathcal{M}_2)$ they are similarly defined.

set$(i, j, w, \mathcal{M})$ Update $\mathcal{M}$ by setting $\mathcal{M}[i, j] = w$. This corresponds either to an insertion (if w is nonempty) or to a deletion (if w is empty). It causes the implicit renumbering of the horizontal and vertical rankings.

range$(h_1, h_2, v_1, v_2, \mathcal{M})$ Find the solution $\mathcal{P}(R)$ to problem $\mathcal{P}$ restricted to the nonempty entries contained in region $R = \mathcal{M}[h_1, h_2; v_1, v_2]$.

Our technique works for a general matrix $\mathcal{M}$. However, in the remainder of this paper we restrict ourselves to the special case where each row or column of $\mathcal{M}$ contains a constant number of points. This is without loss of generality, as a row or a column with s points can be represented by a sequence of $\Theta(s)$ columns or rows with a constant number of points: this transformation does not affect the achieved bounds and can be easily maintained throughout our sequence of operations.

Before describing the two levels of our data structures for the splitting and merging problem, we need some preliminary definitions. Let $X = \{x_1, x_2, \ldots, x_q\}$ be a sorted sequence of q elements, according to some total order $\prec$: $x_1 \prec x_2 \prec \cdots \prec x_q$. Let $I_1, \ldots, I_s$ be a partition of X into adjacent intervals, so that for $1 \le i \le s-1$ all the elements in I_i precede all the elements in I_{i+1}. For $1 \le i \le s$, let $|I_i|$ denote the size of interval I_i, defined as the number of elements in I_i.

Definition 2.1 (Size Invariant) *Let $k \ge 1$ be a positive integer. The adjacent intervals $I_1, \ldots, I_s$ satisfy the* size invariant of order k *if the following two conditions are met: (a) $|I_i| \le k$, $1 \le i \le s$; and (b) $|I_i| + |I_{i+1}| > k$, $1 \le i \le s-1$.*

The size invariant of order k in Definition 2.1 implies that the number s of intervals is $O(q/k)$. We remark that the size invariant can be easily maintained when an element is deleted from X or a new element is inserted into X. This can be done as follows. When an element x is inserted into interval I_i, we check whether the size of I_i still satisfies condition (a) of Definition 2.1. If condition (a) is violated, $|I_i| = k+1$, and we split I_i into two intervals, say I'_i and I''_i such that all the elements of I'_i precede all the elements of I''_i. Next, if condition (b) is violated for I_{i-1} and I'_i (resp. I''_i and I_{i+1}), we merge I'_i (resp. I''_i) with I_{i-1}

(resp. I_{i+1}). If an element x is deleted from interval I_i, then we have still to check that condition (b) is met for I_{i-1} and I_i and for I_i and I_{i+1}. In the worst case, this implies again a constant number of interval merges and splits.

3 Level I: The Basic Squares

We now describe the first level of our data structure for a matrix $\mathcal{M}$, which operates a space-driven partition of the points. We refer to the N nonempty entries of $\mathcal{M}$ as the *points* of $\mathcal{M}$ and let k be a slack parameter, where k is an integer with $1 \leq k \leq N$. We handle the *sparse* $N \times N$ matrix $\mathcal{M}$ *as if it were a dense matrix* of size $\Theta(N/k + k) \times \Theta(N/k + k)$. We then tune k according to the chosen problem $\mathcal{P}$. We group adjacent rows and columns of matrix $\mathcal{M}$ into respectively *horizontal and vertical stripes* so as to satisfy the size invariant of order k: namely, any horizontal (resp. vertical) stripe contains at most k rows (resp. columns), and any two adjacent horizontal (resp. vertical) stripes contain more than k rows (resp. columns). This guarantees that each stripe contains at most $O(k)$ points and that the total number of horizontal and vertical stripes is $O(N/k)$. The partition into horizontal and vertical stripes induces a partition of $\mathcal{M}$ into $O(N^2/k^2)$ squares, such that each square intersects no more than k rows and k columns. These are the *basic squares* in $\mathcal{M}$. For each such basic square, we maintain a solution to problem $\mathcal{P}$ for the points in the square.

We store these points in external memory blocks of size B according to this partition scheme so as to guarantee a fast access to the basic squares in a stripe along the two dimensions, i.e., both horizontally and vertically. Ideally, we would like to get an $O(\sqrt{N/B})$ cost by organizing all the N points into a "dense" $(\sqrt{N} \times \sqrt{N})$ square grid and by covering the grid with a number of $(\sqrt{B} \times \sqrt{B})$-shaped squares, such that the union of these (disjoint) squares is the grid and each square is stored into one block. However, this organization seems quite difficult to maintain as the matrix is both sparse and dynamic: we insert and delete points in a sparse order and perform split and concatenate operations as well. We will instead follow a different approach whose cost is $O(N/k + k/B)$, which is still $O(\sqrt{N/B})$ if we choose $k = \lceil \sqrt{NB} \rceil \leq N$.

We store points in external memory by vertical stripes, as follows. Let σ be a vertical stripe, and assume without loss of generality that σ does not fit into one block, i.e., $k = \Omega(B)$ (otherwise, we can always stuff a number of adjacent vertical stripes into the same block and then access a stripe with a single I/O). We group *adjacent* basic squares of σ into $O(k/B)$ disjoint groups. These groups satisfy the size invariant of order B on the total size of their basic squares, with the notable exception of basic squares with size larger than B. Each basic square of this kind contains $\Omega(B)$ points, and so it forms a singleton group on its own and is stored in a list of blocks. We implement this list by means of a B^+-tree [12] that can be updated, split and concatenated in $O(\log_B k)$ I/Os. To this end, we maintain a $\prec_{\mathrm{B}}$-order on the points in the basic square at hand. Note that the size of the basic square remains proportional to the number of points in the square, as the size of this B^+-tree is proportional to the number of points divided by B. The remaining groups all contain the basic squares with size bounded by B and each group can be stored into a single block. Although unnecessary, we also keep the points in each single-block group sorted by $\prec_{\mathrm{B}}$, i.e., a single-node B-tree. Note that a vertical stripe can be stored in at most $O(k/B)$ different blocks.

Lemma 3.1 *The points contained in the basic squares belonging to either a horizontal or a vertical stripe in $\mathcal{M}$ can be retrieved in the correct order (i.e., left to right and bottom-up) with $O(N/k + k/B)$ I/Os.*

Proof: A vertical stripe can be easily accessed with $O(k/B)$ I/Os. Consider now a horizontal stripe σ. As σ contains $O(k)$ points, at most $O(k/B)$ blocks can have all of their points completely inside σ. Furthermore, at most $O(N/k)$ blocks can contain points both from σ and from either of its neighbor horizontal stripe, as there are no more than two such blocks per basic square. Hence, the total number of different blocks required to access a horizontal stripe in our vertically oriented organization is $O(N/k + k/B)$. $\square$

We conclude this section by observing that the total number of blocks needed to store the matrix $\mathcal{M}$ is optimal, i.e., $\Theta(N/B)$. Furthermore, we can maintain efficiently the size invariant for any two adjacent stripes and their blocks with a constant number of scans.

4 Level II: The Cross-Tree

In the second level, the basic squares are uniquely associated with the leaves of a *cross-tree* [18], a multidimensional hierarchical data structure that describes recursively the partition of $\mathcal{M}$ into its basic squares and that supports efficiently *split and concatenate operations*.

A cross-tree [18] describes a balanced decomposition of a bidimensional set, and it is based upon two balanced trees (such as 1-2-trees, 2-3-trees [2] or B$^+$-trees [12]). Let T_H and T_V be two balanced trees, whose leaves are on the same level and correspond to the horizontal and vertical stripes respectively, and which recursively describe a multilevel horizontal and vertical partition into stripes. Then a *cross-tree* $CT(T_H \times T_V)$ for dimension $d = 2$ is the cross product of T_H and T_V. In particular:

- For each pair of nodes $u \in T_H$ and $v \in T_V$ on the same level, there is a node α_{uv} in $CT(T_H \times T_V)$ with no more than b^2 children, where b is the maximum degree of u and v.
- For each pair of edges $(u, \hat{u}) \in T_H$ and $(v, \hat{v}) \in T_V$, such that u and v are on the same level, there is an edge $(\alpha_{uv}, \alpha_{\hat{u}\hat{v}})$ in $CT(T_H \times T_V)$.

An example illustrating a cross-tree is shown in Figure 2. One important property of a cross tree is the following. Let π be a leaf-to-root path in either balanced tree (say T_H). Then, all the cross tree nodes that are composed with

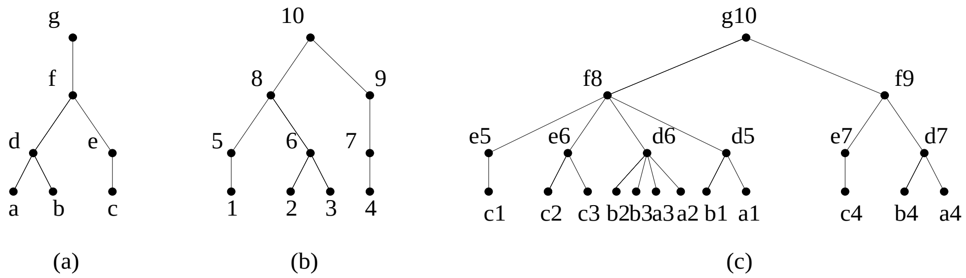


Figure 2: An example of cross-tree $CT(T_H \times T_V)$ for the two trees T_H and T_V respectively shown in (a)–(b) and corresponding to the horizontal and vertical stripes of the basic squares illustrated in Figure 1. Here, T_H and T_V have maximum degree $b = 2$ and so the cross-tree has maximum degree $b^2 = 4$. The actual representation of $CT(T_H \times T_V)$ is shown in (c).

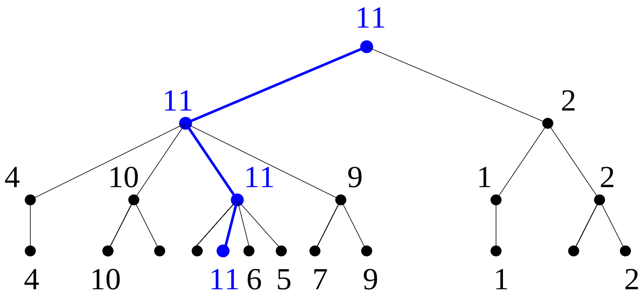


Figure 3: The cross-tree of Figure 2 with nonempty solutions stored in its nodes for the max-weight problem treated in Figure 1.

nodes from π, induce a subtree of $CT(T_H \times T_V)$ which is isomorphic to T_V. For instance in Figure 2, the cross-tree part identified by nodes `b1`, `b2`, `b3`, `b4`, `d5`, `d6`, `d7`, `f8`, `f9`, `g10`, is isomorphic to T_V and corresponds to leaf-to-root path $\pi = \{$`b`, `d`, `f`, `g`$\}$ in T_H. Analogously, the part identified by `a3`, `b3`, `c3`, `d6`, `e6`, `f8`, `g10`, is isomorphic to T_H and corresponds to path $\pi = \{$`3`, `6`, `8`, `10`$\}$ in T_V. This property is clearly important if we have to split (or concatenate) cross-trees in response to splits (or concatenates) in their trees T_H and T_V. We refer the reader to [18] for further properties and details on cross-trees.

As problem $\mathcal{P}$ is decomposable, we store the solutions of the basic squares in the corresponding leaves of the cross-tree and percolate these solutions up in a heap-like fashion by using operator $\diamondsuit$ as shown in Figure 3. We are assuming that each solution takes constant space, and we treat the case of solutions requiring more space later on.

We store the cross-tree $CT(T_H \times T_V)$, built according to the partition of $\mathcal{M}$ into basic squares, in an optimal number $\Theta(N/B)$ of external memory blocks. As we want to support efficient split and concatenate operations, we choose to use two *weight-balanced B-trees* T_H and T_V in our cross tree: they are a weight-balanced variant of B-trees and 2-3-trees, introduced by Arge and Vitter [5]. We bound their maximum number of children by $b = \sqrt{B}$, so that the cross-tree has branching factor $\Theta(B)$. The weight-balanced B-trees have several interesting features, including leaf insertions, leaf deletions, splits and concatenates in $O(\log_B |T|)$ I/Os per operation. Each operation only involves the nodes in a leaf-to-root path and their children. If the two trees have different heights, we replace the root of the lower tree with a chain of $O(1)$ unary nodes so that they both have the same height. Consequently, the cross-tree $CT(T_H \times T_V)$ has height $O(\log_B(N/k))$ and has a total number of $\Theta(N/B)$ nodes (i.e., blocks). The following lemma is easy to derive.

Lemma 4.1 *The cross-tree leaves corresponding to the basic squares in either a horizontal or a vertical stripe in $\mathcal{M}$ can be retrieved (together with their ancestors) in bottom-up order, with $O(N/k)$ I/Os.*

5 Putting the Two Levels Together

We now have all the ingredients for a general view of our data structure. The basic squares in the lower level are associated with the leaves of the cross-tree in the upper level. The recursive nature of decomposable problem $\mathcal{P}$ suggests us that its solution $\mathcal{P}(S) = \diamondsuit(\mathcal{P}(S'), \mathcal{P}(S''))$ can be stored in the root of the cross-tree and that $\mathcal{P}(S'), \mathcal{P}(S'')$ can be stored recursively in its two children. Each internal node therefore stores the solution to $\mathcal{P}$ for the items in S associated with its descendent leaves. Our basic data structure comprises matrix $\mathcal{M}$ and cross-tree $CT(T_H \times T_V)$. Trees T_H and T_V have $O(N/k)$ leaves, one for each stripe of $\mathcal{M}$, and a total of $O(N/k)$ nodes. Consequently, the resulting cross-tree $CT(T_H \times T_V)$ has $O(N^2/k^2)$ leaves, one for each basic square of $\mathcal{M}$, and a total of $O(N^2/k^2)$ nodes. Our data structure has the following features:

1. For each nonempty basic square of $\mathcal{M}$, we keep its points sorted according to an order $\prec_{\mathrm{B}}$ (which is not necessarily equal to $\prec_X$ or $\prec_Y$) by means of a B^+-tree. Searching, inserting and deleting a point takes $O(\log_B k)$ I/Os. We introduce order $\prec_{\mathrm{B}}$ because some data structures can be (re)built more efficiently on a sorted set of points.

2. Each cross-tree node corresponds to a region of matrix $\mathcal{M}$. If a node is empty then its corresponding region is empty. A leaf corresponds to a basic square. An internal node α corresponds to a *region* $\mathcal{M}[h_1, h_2; v_1, v_2]$, and has no more than B children $\alpha_1, \ldots, \alpha_j$ corresponding to an orthogonal partition of $\mathcal{M}[h_1, h_2; v_1, v_2]$ into smaller regions. For example, if α has $j = 4$ children, they correspond to the regions: $\mathcal{M}[h_3, h_2; v_1, v_3]$, $\mathcal{M}[h_3, h_2; v_3, v_2]$, $\mathcal{M}[h_1, h_3; v_3, v_2]$, and $\mathcal{M}[h_1, h_3; v_1, v_3]$, for some $h_1 \leq h_3 \leq h_2$ and $v_1 \leq v_3 \leq v_2$. In other words, the smaller (non-overlapping) regions corresponding to $\alpha_1, \ldots, \alpha_j$ can be united to produce the region corresponding to α.

3. For each nonempty basic square of $\mathcal{M}$, we examine its points and store their solution to problem $\mathcal{P}$ into their corresponding cross-tree leaf. We then percolate this information up in the cross-tree: let α be an internal node, and let $s_1, \ldots, s_j$ be the solutions stored in the $j \leq B$ children of α. As $\mathcal{P}$ is a decomposable problem, the internal node α stores its solution $\diamondsuit(s_1, \ldots, s_j)$ for the points in its corresponding region in $\mathcal{M}$. (Recall that $\diamondsuit$ is associative and so is well-defined also for an arbitrary number of arguments.)

We assumed so far that the solutions take constant space each. When this is not the case, an internal node can store *implicitly* its corresponding solution $\mathcal{P}(R)$: namely, the node stores the $O(f(N))$ actions taken to compute $\mathcal{P}(R) = \diamondsuit(\mathcal{P}(R'), \mathcal{P}(R''))$ and its (say, two) children store only the leftover pieces of $\mathcal{P}(R')$ and $\mathcal{P}(R'')$ that are not employed to build $\mathcal{P}(R)$. A full solution is available only at the root. In order to obtain the solution in the other nodes, we can use a simple top-down procedure to obtain $\mathcal{P}(R'), \mathcal{P}(R'')$ from $\mathcal{P}(R)$ recursively along a downward path, and another simple bottom-up procedure to obtain $\mathcal{P}(R)$ from $\mathcal{P}(R'), \mathcal{P}(R'')$ along an upward path. We therefore say that $\diamondsuit$ is *invertible* if we can keep $O(f(N))$ bookkeeping information associated with any solution $\mathcal{P}(R)$ so that we can compute $\diamondsuit^{-1}(\mathcal{P}(R))$ with $O(f(N))$ I/Os. Operator $\diamondsuit$ corresponds to executing the bottom-up procedure while the inverse operator $\diamondsuit^{-1}$ corresponds to the top-down procedure. At this point a comment is in order. As cross-tree degree is upper bounded by B, we could require $O(f(N)\,B)$ I/Os to (re)compute the solution in a node. While this does not change the I/O complexity of the split and concatenate operations, it may change the one of insertions and deletions. In this case, we say that $\diamondsuit$ is *strongly invertible* if we can obtain the solution $\mathcal{P}(R')$ in any children of a node, along with what remains $\mathcal{P}(R - R')$ from its solution $\mathcal{P}(R)$, with $O(f(N))$ I/Os. For example, if $\diamondsuit$ is the destructive list append with cost $f(N) = O(1)$, we can simply keep a pointer to the last item in the appended lists to "de-append" and "re-append" any of them with $O(1)$ I/Os by means of $\diamondsuit^{-1}$. In summary, when dealing with point 3 above, if $\diamondsuit$ is strongly invertible and solution $\diamondsuit(s_1, \ldots, s_j)$ is not of constant size, we use $\diamondsuit^{-1}$ to store the $O(f(N)\,j)$ actions needed to compute $\diamondsuit(s_1, \ldots, s_j)$ into α itself and store only the leftover of solutions $s_1, \ldots, s_j$ into the children of α. We can indeed recover any of $s_1, \ldots, s_j$ in $O(f(N))$ I/Os by applying $\diamondsuit^{-1}$ to the solution in α, which is itself recursively computed from the parent of α.

Since we have all the details for our data structure, we can now analyze its performance.

Theorem 5.1 *We can insert in, delete from, concatenate and split T_H or T_V in order to update cross-tree $CT(T_H \times T_V)$ with $O(f(N)(N/k))$ I/Os, where $f(N) = o(N)$.*

Proof: We only analyze the complexity of splitting as the other operations have an analogous analysis. Without loss of generality, assume that T_H is split into T_1 and T_2 (horizontal split).

Splitting T_H takes $O(\log_B(N/k))$ time, and the cost of splitting the cross-tree $CT(T_H \times T_V)$ accordingly consists of two tasks: (1) updating the cross-tree topology and (2) recomputing the solutions to $\mathcal{P}$ by means of operators $\diamondsuit$ and $\diamondsuit^{-1}$.

Let n_ℓ be the number of nodes of T_V on level ℓ, where $0 \leq \ell \leq h = O(\log_B(N/k))$. In task (1), we examine $O(n_\ell)$ nodes per level in the cross-tree and execute $O(n_\ell)$ work on them. By summing over all the levels ℓ, we obtain a total cost of

$$\sum_{\ell=0}^{h} O(n_\ell) = O(|T_V|) = O(N/k),$$

since the path that is splitting T_H induces a subtree of $CT(T_H \times T_V)$ which is isomorphic to T_V.

In order to analyze task (2), we proceed as follows. We call a cross-tree node α nonempty if its corresponding region in $\mathcal{M}$ contains at least one point, and we call it empty otherwise. Since we traverse no more than $O(N/k)$ empty nodes by the analysis of task (1), we have to bound the cost for the nonempty nodes and take their cost $f(N)$ into account. We define I_ℓ to be the set of nonempty nodes involved, on level ℓ, by the splitting operation. Their total contribution can be expressed as

$$O(\sum_{\ell=0}^{h} \sum_{\alpha \in I_\ell} f(r_\alpha)),$$

where r_α denotes the number of points in the region corresponding to α, with $1 \leq r_\alpha \leq N$. Note that the cost of applying $\diamondsuit$ and $\diamondsuit^{-1}$ to an internal node is proportional to the number of its children involved in the operation, and therefore is charged to them in the total contribution. As $f(r_\alpha) \leq f(N)$, we have that:

$$O(\sum_{\ell=0}^{h} \sum_{\alpha \in I_\ell} f(N)) = O(\sum_{\ell=0}^{h} |I_\ell| f(N)) = O(f(N) \sum_{\ell=0}^{h} |I_\ell|) = O(f(N)(N/k)),$$

where $|I_\ell| = O(n_\ell)$ by the split operation.

Combining the analysis of tasks (1) and (2) and adding the $O(\log_B(N/k))$ splitting cost of T_H yields the claimed bounds. $\square$

By an analogous argument, we can prove:

Theorem 5.2 *Building the cross-tree $CT(T_H \times T_V)$ requires $O(f(N)(N^2/k^2))$ I/Os and the space used is $O(f(N)(N^2/(k^2B)))$ blocks, where $f(N) = o(N)$.*

6 Implementing the Operations

We now show how to use our data structure for solving an order decomposable problem $\mathcal{P}$. To keep our presentation general, we introduce a few functions that help us to state some I/O and space bounds.

$P(k)$ The number of I/Os required to preprocess an $O(k)$-point stripe and solve problem $\mathcal{P}$ for every basic square in the stripe. We will exploit the fact that "large" basic squares are already $\prec_{\mathrm{B}}$-ordered to determine $P(k)$, and assume that $P(k) \geq k/B$ as we need to scan the k points, and that $P(k)/k$ is a smooth nondecreasing function.

$U(k)$ The number of I/Os required to update the solution to problem $\mathcal{P}$ for a basic square in an $O(k)$-point stripe after its preprocessing. We assume that $U(k) \geq \log_B k$, since we have to update at least the B$^+$-tree in the basic square.

$S(k)$ The number of blocks occupied by an $O(k)$-point stripe. We also assume that $S(k) \geq k/B$ is a smooth nondecreasing function.

In most of our applications, we will have $P(k) = O(k/B)$, $S(k) = O(k/B)$, $U(k) = O(\log_B k)$ (and $k = \lceil \sqrt{NB} \rceil \leq N$).

Preprocessing N points requires their external memory sorting in lexicographic order with $Sort(N)$ I/Os and an external memory sorting of all the points in every basic square in $\prec_{\mathrm{B}}$-order. Next, the B$^+$-trees are built in a total of $\Theta(N/B)$ I/Os. It takes further $O(P(N))$ I/Os to scan the stripes, determine the solutions in their basic squares and to group the basic squares into blocks. Further $O(f(N)(N^2/k^2))$ I/Os are needed to build the cross-tree and recompute the solution in its nodes by means of operator $\diamondsuit$ according to Theorem 5.2. In summary, the preprocessing step takes a total $O(Sort(N) + P(N) + f(N)(N^2/k^2))$ I/Os, where $Sort(N)$ is the I/O complexity of sorting N points.

We now describe how to perform the dynamic operations on $\mathcal{M}$.

***Operations** h_split, v_split:* We perform $v\_split(\mathcal{M}, j)$ as follows. Column j might fall inside a vertical stripe σ, which must necessarily be split. We examine the basic squares of σ. Given a basic square, we scan its points according to their $\prec_{\mathrm{B}}$-order and produce two $\prec_{\mathrm{B}}$-*ordered* lists with a simple scan: one list contains all the points whose second coordinate is smaller than or equal to j and the other list contains the remaining points. We split the basic square into two squares and, if necessary, build two B$^+$-trees by using the two $\prec_{\mathrm{B}}$-ordered lists. By repeating this for all the $O(N/k)$ basic squares in σ, which contains $O(k)$ points, we split it into new stripes σ_1 and σ_2, with $O(k/B + N/k)$ I/Os by Lemma 3.1. We check to see if we can combine σ_1 and σ_2 with their neighbor stripes to maintain the size invariant of order k. For any two such stripes to be merged, we examine their basic squares in pairs (a square per stripe), such that the two squares are on the same horizontal stripe. We take their two $\prec_{\mathrm{B}}$-ordered lists of points and merge them to build a B$^+$-tree on the resulting list with a linear number of I/Os. Again, this requires $O(k/B + N/k)$ I/Os. It is worth noting that splitting and merging stripes preserve the order of their presorted points. We maintain the size invariant of order B to adjust the number of blocks storing these basic squares, with further $O(N/k + k/B)$ I/Os. After that, we recompute the solutions in the $O(N/k)$ basic squares contained in a constant number of stripes with $O(P(k))$ I/Os. It remains to split cross-tree $CT(T_H \times T_V)$

to reflect the split operation on σ. We split tree T_V at the leaf w corresponding to σ. We split w into two new leaves w_1 and w_2, corresponding to the split of σ into σ_1 and σ_2. If σ_1 or σ_2 are combined with their neighbor stripes, we do the same on w_1 and w_2 and their neighbor leaves. Globally, we create no more than $O(N/k)$ leaves corresponding to the new basic squares in $O(1)$ stripes, and we update their ancestors by a bottom-up traversal of the cross-tree, with $O(N/k)$ I/Os by Lemma 4.1. We then take $O(f(N)(N/k))$ I/Os to update the solutions in the internal nodes by means of $\diamondsuit$ (and possibly of $\diamondsuit^{-1}$). We therefore spend a total of $O(f(N)(N/k)+k/B+P(k)) = O(N/k+P(k))$ I/Os, as $P(k) \geq k/B$. The implementation of *h_split* is completely analogous.

***Operations** h_concatenate, v_concatenate:* They are the inverse of *h_split* and *v_split* respectively, and can be implemented in a total of $O(f(N)(N/k)+P(k))$ I/Os.

***Operation** range$(h_1, h_2, v_1, v_2, \mathcal{M})$:* We perform it by executing an *h_split* at row $(h_1 - 1)$ followed by another *h_split* at row h_2. We let r be the number of rows of $\mathcal{M}$. These two horizontal splits produce three matrices $\mathcal{M}_1, \mathcal{M}_2$, and $\mathcal{M}_3$, such that $\mathcal{M}_1$ consists of the first $(h_1 - 1)$ rows of $\mathcal{M}$, $\mathcal{M}_2$ contains the rows of $\mathcal{M}$ between h_1 and h_2, and $\mathcal{M}_3$ contains the last $(r - h_2 + 1)$ rows of $\mathcal{M}$. We produce submatrix $\mathcal{M}[h_1, h_2; v_1, v_2]$ by carrying out two additional vertical splits on $\mathcal{M}_2$ at columns $(v_1 - 1)$ and v_2, respectively. We give the solution that is stored in the root of the resulting cross-tree, which corresponds to region $\mathcal{M}[h_1, h_2; v_1, v_2]$, and then assemble back the pieces by a proper sequence of four *concatenate* operations. Since it can easily be mimicked by a proper sequence of $O(1)$ *split* and *concatenate* operations, a *range* operation takes $O(f(N)(N/k)+P(k))$ I/Os. It is worth noting that we do not actually need to perform splits and concatenates for a range query. It is sufficient to search for the boundaries indicated by the splits and concatenates, and report the relevant items from the boundary basic squares as indicated by the process of applying $\diamondsuit^{-1}$ and $\diamondsuit$. The interior basic squares are reported in full. This alternative approach does not change the asymptotic complexity but makes the range query algorithm much faster in practice.

***Operation** set$(i, j, w, \mathcal{M})$:* We use it to perform either an insertion (if w is nonempty) or a deletion (if w is empty). Let us assume first that we have only insertions. We locate the basic square corresponding to position (i, j) by traversing a path π of length $O(\log_B(N/k))$ from the cross-tree root to the proper leaf. If necessary, we insert (i, j) into the B$^+$-tree in the basic square. Since we use the $\prec_{\mathrm{B}}$-order, this takes $O(\log_B k)$ I/Os. We then update the solution in the basic square with $U(k)$ I/Os and link it to its relative cross-tree leaf if the basic square becomes nonempty. We propagate the new solution upwards along path π by applying operator $\diamondsuit$ to its nodes, with $O(f(N))$ I/Os per node (and possibly with the help of $\diamondsuit^{-1}$). The total cost is $O(f(N)\log_B(N/k))$ I/Os. As setting $\mathcal{M}[i, j] = w$ can cause the insertion of a new row at position i and a new column at position j, the horizontal stripe and the vertical stripe whose intersection gives the updated basic square can violate the size invariant of order k. Since we are treating insertions, this means that each such stripe σ has increased its number of rows (columns) to $k+1$. Consequently, we split σ into two stripes σ_1 and σ_2 of at most $\lceil (k+1)/2 \rceil$ rows (columns) each in order to maintain the size invariant, with $O(P(k)+f(N)(N/k))$ I/Os (as in the split operation). This gives a total amortized cost of $O(f(N)\log_B(N/k) + U(k) + P(k)/k + f(N)(N/k^2))$ per insertion. In order to obtain good worst-case bounds for the single insertion, we spread this cost among the subsequent operations on the items in σ. To this end, we use Overmars' global rebuilding technique [33] by adding an extra worst-case cost $O(P(k)/k + f(N)(N/k^2))$ to each insertion, such that $\lceil (k+1)/2 \rceil - 1$ insertions are sufficient to cover the $O(P(k) + f(N)(N/k))$ splitting cost for σ. To perform deletions, we simply mark the given item as logically deleted and update its basic square with $U(k)$ I/Os and the solutions along the path π to its cross-tree leaf in a total of $O(\log_B(N/k) + U(k))$ I/Os. We apply again global rebuilding to bound the number of deleted items. In summary, *set* can be supported in $O(f(N)\log_B(N/k) + U(k) + P(k)/k + f(N)(N/k^2))$ I/Os.

This is summarized in the following theorem.

Theorem 6.1 *The splitting and merging problem on $N > M$ points can be solved with the following number of I/Os, for a parameter k, and for $f(N) = o(N)$:*

- *range, h_split, v_split, h_concatenate, v_concatenate in $O(f(N)(N/k) + P(k) + P(N)/N)$, with $P(k) \geq k/B$;*
- *set in $O(U(k) + f(N)(\log_B(N/k) + (N/k^2)) + P(N)/N)$, with $U(k) \geq \log_B k$.*

The number of occupied blocks is $O(S(N) + f(N)(N^2/(k^2B)))$ and the preprocessing cost is $O(Sort(N) + P(N) + f(N)(N^2/k^2))$ I/Os, where $Sort(N)$ is the I/O complexity of sorting N points.

In most of our applications $P(k) = O(k/B)$, $U(k) = O(\log_B k)$ and $S(k) = O(k/B)$:

Corollary 6.2 *When $P(k) = O(k/B)$, $U(k) = O(\log_B k)$, $S(k) = O(k/B)$, and $f(N) = O(1)$, the splitting and merging problem on N items can be solved with the following I/O bounds:*

- *h_split, v_split, h_concatenate, v_concatenate in $O(\sqrt{N/B}\,)$;*
- *range in $O(\sqrt{N/B} + r/B)$, where r is the number of points reported by the search;*
- *set in $O(\log_B N)$.*

The space required is $O(N/B)$ blocks and the preprocessing cost is $O(Sort(N))$ I/Os.

Proof: Fix $k = \lceil \sqrt{N\ B}\,\rceil \leq N$. The corollary is now a direct consequence of Theorem 6.1. □

7 Some Applications

We can apply Theorem 6.1 and Corollary 6.2 to a large set of order decomposable problems for basic data structures (e.g., member searching, predecessor, ranking, priority queues), in computational geometry (e.g., neighbor queries, union and intersection queries, interval queries, convex hull), and database applications (e.g., partial match queries, range queries). The reader is referred to [18] for more details on this.

We now sketch an alternative approach to solve our splitting and merging problem. We illustrate it by implementing a two-dimensional priority queue in external memory, in which split, concatenate, insertions and deletions have all the same cost: $O(Sort(\sqrt{N \log_2 B})) = O\left(\frac{\sqrt{N \log_2 B}}{B} \log_{M/B} \frac{\sqrt{N \log_2 B}}{B}\right)$ I/Os. This bound is asymptotically greater than the one given in Corollary 6.2. However, due to the typical values of B, we expect that it could be faster in practice.

We describe the main ideas by examining again the "ideal" memory organization in which the N points in matrix $\mathcal{M}$ are organized in a "dense" $(\sqrt{N} \times \sqrt{N})$ square grid. The result in Goodrich et al. [17] implies that an arbitrary $\sqrt{N}$ long path in the grid must require $\Theta(\sqrt{N/B})$ I/Os. Although $\mathcal{M}$ is *sparse* and *dynamic*, the approach leading to Corollary 6.2 obtains this bound in a simple and practical way by fixing the slack parameter k so as to minimize the term $N/k + k/B$. If $\mathcal{M}$ is static, we can improve N/k and obtain a better bound: we duplicate $\mathcal{M}$ and store one copy in column order and the other copy in row order (in a sequence of adjacent blocks). Consequently, we can access a row or a column in the grid by retrieving $\sqrt{N}/B$ blocks from the proper copy. This is not in contrast with the $\Theta(\sqrt{N/B})$ bound given by Goodrich *et al.*, as they allow for an arbitrary path in the square grid while here we only deal with some axis parallel paths. Basically, we would like to follow a similar approach when $\mathcal{M}$ is dynamic.

Theorem 7.1 *A two-dimensional priority queue for a set of N items can be maintained in external memory with the following I/O bounds:*

- *an item insertion or deletion, a split or concatenate of one order, a minimum-weight query in a region: $O(Sort(\sqrt{N \log_2 B}))$.*

The space required is $O(N/B)$ blocks and the preprocessing cost is $O(Sort(N))$ I/Os.

Proof: We replace $\mathcal{M}$ by a *hierarchy* of $\log_2 B$ smaller and smaller matrices that can be handled in a "loosely" fashion. In this way, the cross-tree is pruned in its $\log_2 B$ lower levels, and the matrix on level $\ell < \log_2 B$ basically stores the same information as the pruned cross-tree nodes that originally were on level ℓ. Each such matrix is stored in two copies according to its horizontal and vertical stripes, respectively. The splits and concatenates cause some swaps on its columns and its rows while producing some item deletions and insertions due to the re-computation of the solutions (previously kept in the pruned cross-tree nodes). We delay and buffer these swaps on the several levels of the matrix hierarchy. For a given level, the matrix copy stored according to its horizontal stripes uses a buffer to store the pending vertical swaps while the horizontal swaps are promptly executed. Analogously, the matrix copy with the vertical stripes uses a buffer to store the pending horizontal swaps while the vertical swaps are promptly executed. We apply the pending swaps to a given stripe with s items in lazy fashion, namely soon before accessing it. We do it by carefully sorting the swaps and the deleted/inserted items pending on the stripe, with $\mathit{Sort}(s)$ worst-case I/Os. Because of this buffering, our insertions and deletions have to collect the points scattered inside each stripe but we can pay $\mathit{Sort}(s)$ for handling that. As a result, each operation costs $\mathit{Sort}(s)$ per level, where $s = O(N/k + k)$ for the lowest level and $s = O(N/k)$ for the other levels. Since the number of cross-tree nodes is $O(N^2/(k^2/B^2))$ as a result of pruning, the cost of accessing the cross-tree is $O(N/(kB))$. This gives a total of $O(\mathit{Sort}(N/k + k) + \mathit{Sort}(N/k)\log_2 B + N/(kB))$ I/Os, which is

$$O\left(\frac{k}{B}\log_{M/B}\frac{k}{B} + \left(\frac{N/k}{B}\log_{M/B}\frac{N/k}{B}\right)\log_2 B\right)$$

as $\mathit{Sort}(N) = \Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$. Choosing $k = \lceil\sqrt{N\log_2 B}\,\rceil$ yields

$$O\left(\frac{\sqrt{N\log_2 B}}{B}\log_{M/B}\frac{\sqrt{N\log_2 B}}{B} + \left(\frac{\sqrt{N}}{B\sqrt{\log_2 B}}\log_{M/B}\frac{\sqrt{N}}{B\sqrt{\log_2 B}}\right)\log_2 B\right)$$

which is $O(\mathit{Sort}(\sqrt{N\log_2 B}\,))$. $\square$

We conclude this section by mentioning that our technique can be applied to graph problems as well, and in particular we describe an application to the dynamic maintenance of a minimum spanning tree of a graph. This problem consists of maintaining a minimum spanning tree of G subject to edge insertions, edge deletions and edge cost updates. We describe here a simple external memory algorithm that has the same cost as splitting/concatenating in a two-dimensional priority queue: either $O(\sqrt{|V|/B}\,)$ or $O(\mathit{Sort}(\sqrt{|V|\log_2 B}\,))$ I/Os per edge update in G with a total of $O(|E|/B)$ blocks storing the graph. Faster algorithms for main memory were recently given in [23].

In what follows, we refer to the edges in the minimum spanning tree as *tree edges* and to the edges not in the minimum spanning tree as *non-tree edges*. We only deal with connected graphs but we can take easily into account the case of disconnected graphs. We also assume that there is a unique minimum spanning tree in G throughout a sequence of operations: this is without loss of generality, as possible ties can be broken consistently. An important property is often used by dynamic minimum spanning tree algorithms: without any loss in generality, we can assume to work with bounded degree graphs by means of a standard technique [21] that transforms a graph G with $|E|$ edges and $|V|$ vertices into a bounded degree graph G' with $O(|E|)$ vertices and edges. This transformation

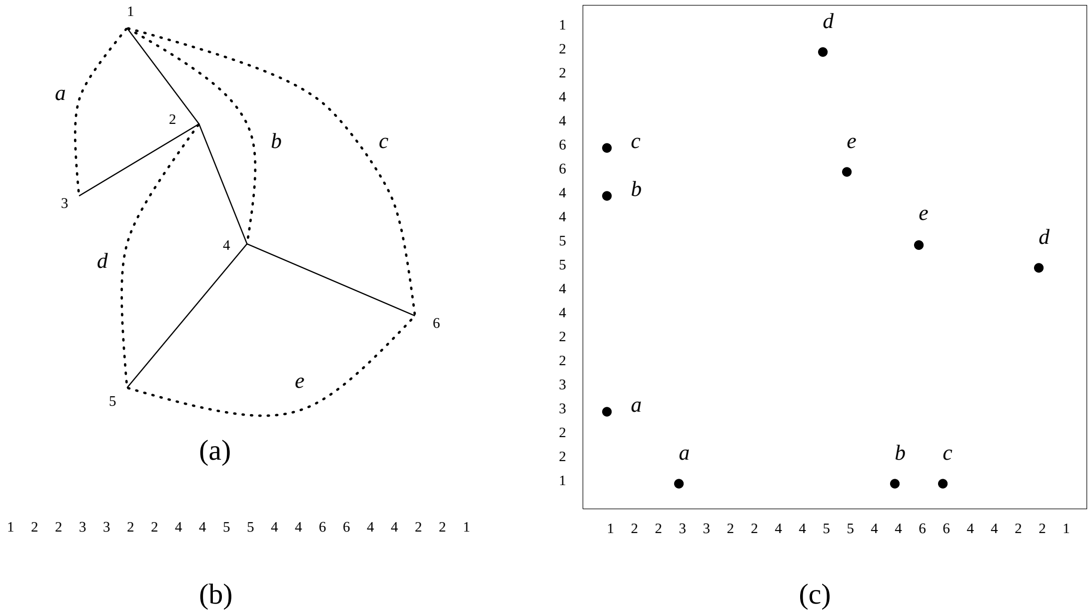
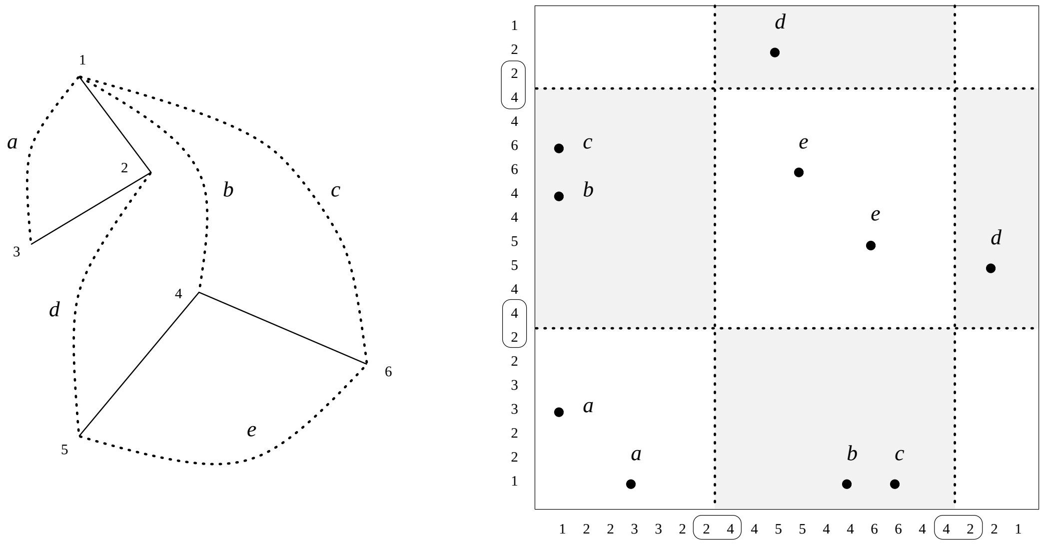


Figure 4: (a) A graph and its minimum spanning tree T in boldface (the edge weights are not shown). (b) Its Euler tour $ET(T)$. (c) Its corresponding two-dimensional priority queue $PQ(T \times T)$.

allows us to implement each update in G by a constant number of updates in G'.

Let T be the minimum spanning tree. Let $ET(T)$ be the Euler tour for T defined as follows: we draw T in the plane together with a contour which surrounds T closely. The contour passes near each edge twice, once in each direction. We go around this contour and append all the occurrences of the vertices encountered, two per edge, as shown in Figures 4a–b. We can compute the minimum spanning tree T and its Euler tour $ET(T)$ in external memory by using the algorithms of Chiang *et al.* [10]. We maintain $ET(T)$ as a B$^+$-tree [6, 12] subject to split and concatenate operations. We also set a pointer in each edge (u, v) to its two occurrences $ET(T)$ and a pointer in each vertex u to its $2d$ occurrences in $ET(T)$. Recall that $d = O(1)$ as we are operating with constant degree graphs. We can insert, delete, split and concatenate $ET(T)$ and maintain the above information pointed to by the vertices and the edges with $O(\log_B |V|)$ I/Os per operation on $ET(T)$.

We now turn to the non-tree edges and store them into a two-dimensional priority queue $PQ(T \times T)$. For each non-tree edge $e = (u, v)$, we let u_1v_1 and v_2u_2 denote its two occurrences in $ET(T)$ and $w(e)$ denote its weight. We then store e as two items (u_1, v_1) and (v_2, u_2) in $PQ(T \times T)$, both containing the value $w(e)$. We define $\prec_X$ and $\prec_Y$ according to the Euler tour: Given two items (u_1, v_1) and (w_1, z_1), we have $(u_1, v_1) \prec_X (w_1, z_1)$ if and only if u_1



(a) (b)

Figure 5: After deleting tree edge $(2, 4)$ in the graph, we obtain the graph in (a). Its edge replacement is the minimum-weight item in the four regions of $PQ(T \times T)$ shown in grey in (b).

precedes w_1 in $ET(T)$ and $(u_1, v_1) \prec_Y (w_1, z_1)$ if and only if v_1 precedes z_1 in $ET(T)$. An example of this transformation is shown in Figure 4. Since G is a bounded degree graph, each row and column of $PQ(T \times T)$ contains a constant number of points. We can modify $PQ(T \times T)$ by inserting and deleting an edge e (i.e., its two occurrences with an identical cost $w(e)$) and by splitting and concatenating either $\prec_X$ or $\prec_Y$ with either $O(\sqrt{|V|/B})$ or $O(Sort(\sqrt{|V| \log_2 B}))$ I/Os. We can also find the minimum-weight edge e in a rectangular axis parallel region in the same bounds as those of a range query. These operations are possible by using Corollary 6.2 and Theorem 7.1 on the set of $O(|E|)$ occurrences corresponding to the non-tree edges.

We now describe our implementation of dynamic minimum spanning tree operations. It is well known that the difficult operations for dynamic minimum spanning trees are tree edge deletions, as the other cases can be easily handled in logarithmic time. In particular, we only show here how to perform efficiently with the help of $PQ(T \times T)$ the following two tasks:

1. Finding a non-tree edge replacement f for a tree edge e to be deleted.

2. Performing an actual $swap(e, f)$, where e is a tree edge that must become a non-tree edge and f is a non-tree edge that must become a tree edge.

Let T_u and T_v denote the two trees resulting from T after removing $e = (u, v)$, with $u \in T_u$ and $v \in T_v$. Let u_1v_1 and v_2u_2 denote the occurrences in the Euler tour $ET(T)$ associated with edge $e = (u, v)$. Without loss of generality, we assume that $ET(T) = \alpha.u_1v_1.\beta.v_2u_2.\gamma$, where $ET(T_u) = \gamma\alpha$ is the Euler tour for T_u and $ET(T_v) = \beta$ is the Euler tour for T_v. Either $ET(T_u)$ or $ET(T_v)$ can be possibly empty.

Since T_u and T_v define a vertex partition, we execute task (1) by finding f as the minimum-weight non-tree edge in the cut (T_u, T_v). We have to take into consideration all the non-tree edges in this cut: note that a non-tree edge is in this cut if and only if it has exactly one endpoint in $ET(T_u)$ and the other in $ET(T_v)$. In order to locate efficiently all the non-tree edges in this cut we use our data structure for $PQ(T \times T)$, as follows. The two occurrences for e are stored as items (u_1, v_1) and (v_2, u_2). Due to the $\prec_X$- and $\prec_Y$-orders, all the edges in the cut (T_u, T_v) must be contained in the four regions $[\alpha u_1 \times v_1\beta v_2]$, $[u_2\gamma \times v_1\beta v_2]$, $[v_1\beta v_2 \times \alpha u_1]$, $[v_1\beta v_2 \times u_2\gamma]$ in $PQ(T \times T)$ because $ET(T_u) = \gamma\alpha$ and $ET(T_v) = \beta$ (see Figure 5). We can therefore execute four minimum-weight range queries on $PQ(T \times T)$ and take the minimum over the resulting edges.

We now show how to execute task (2). The crucial part is to update $PQ(T \times T)$ to reflect the swap in $ET(T)$. We first remove e from T and insert f into T as a tree edge by updating $ET(T)$ as follows. We locate the two occurrences pointed to by e in $ET(T)$ and decompose $ET(T)$ as $\alpha.u_1v_1.\beta.v_2u_2.\gamma$. Since $ET(T)$ is a circular list broken at an arbitrary point, we can "rotate" it to move u at the beginning. That is, we split α immediately before u_1 and concatenate it with γ to obtain $ET(T) = u_1v_1.\beta.v_2u_2.\gamma\alpha$, where $ET(T_v) = \beta$ and $ET(T_u) = \gamma\alpha$. Let $f = (z, w)$, where $z \in T_v$ and $w \in T_u$. We then locate two consecutive occurrences of z and two consecutive occurrences of w in order to decompose $\beta = \beta' z'.z''\beta''$ and $\gamma\alpha = \delta' w'.w''\delta''$ by means of $O(1)$ splits on the Euler tour. We finally create two occurrences z_1w_1 and w_2z_2 for f and insert them into the Euler tour after removing the two occurrences u_1v_1 and v_2u_2 for e. We obtain $ET(T) = \beta' z'.z_1w_1.w''\delta''.\delta' w'.w_2z_2.z''\beta''$ by means of $O(1)$ concatenates.

In summary, we have to modify the initial Euler tour by deleting u_1, v_1, v_2, u_2, by inserting z_1, w_1, w_2, z_2 and by performing a constant number of splits and concatenates. We perform the same operations on $\prec_X$- and $\prec_Y$-orders to modify $PQ(T \times T)$ accordingly. There can be $O(1)$ non-tree edges whose occurrences contain one of the deleted occurrences of u_1, v_1, v_2 and u_2. If this happens, we choose a different occurrence for u and v and relocate those non-tree edges in $ET(T)$ and $PQ(T \times T)$ by deleting and reinserting them in $O(\log_B |E|)$ time. In general, each update of the minimum spanning tree requires $O(1)$ insertions, deletions, splits, concatenates and range queries in $PQ(T \times T)$ plus a logarithmic cost to update the other tree data structures. This yields the following theorem.

Theorem 7.2 *A minimum spanning tree of a dynamic graph with vertex set V and edge set E can be maintained in external memory with either $O(\sqrt{|E|/B})$ I/Os or $O(Sort(\sqrt{|E| \log_2 B}))$ I/Os per operation in the worst case.*

Combining Theorem 7.2 together with an external memory version of sparsification [13] yields the following theorem:

Theorem 7.3 *A minimum spanning tree of a dynamic graph with vertex set V and edge set E can be maintained in external memory with either $O(\sqrt{|V|/B})$ I/Os or $O(Sort(\sqrt{|V|\log_2 B}))$ I/Os per operation in the worst case.*

8 The Multidimensional Case

Our technique can be easily extended to the multidimensional case, i.e., when we have $d > 2$ total orders $\prec_1, \ldots, \prec_d$. In this case, we reduce our problem to a matrix problem in which $\mathcal{M}$ is a d-dimensional matrix and item $x \in S$ is a d-dimensional point $(x_1, \ldots, x_d)$, such that x_i is the rank of x in S according to order $\prec_i$, $i = 1, \ldots, d$. We begin by defining the cross-trees for dimension $d > 2$. A cross-tree $CT(T_1 \times T_2 \times \cdots \times T_d)$ of d weight-balanced B-trees $T_1, T_2, \ldots, T_d$ having the same height and their leaves on the same level, is defined as follows:

- For each choice of nodes $u_1, u_2, \ldots, u_d$ on the same level, such that $u_i \in T_i$ for $1 \leq i \leq d$, there is a node $\alpha_{u_1 u_2 .. u_d}$ in $CT(T_1 \times T_2 \times \cdots \times T_d)$.
- For each choice of edges $(u_1, \hat{u}_1), (u_2, \hat{u}_2), \ldots, (u_d, \hat{u}_d)$, where $u_1, u_2, \ldots, u_d$ are on the same level and $(u_i, \hat{u}_i) \in T_i$ for $1 \leq i \leq d$, there is an edge $(\alpha_{u_1 u_2 .. u_d}, \alpha_{\hat{u}_1 \hat{u}_2 .. \hat{u}_d})$ in $CT(T_1 \times T_2 \times \cdots \times T_d)$.

We define the stripes in $\mathcal{M}$ by following the approach described before. In dimension two, a stripe is composed of $O(k)$ rows or columns, which can be seen as matrices of dimension one. We therefore define the stripes as the groups of the $O(k)$ adjacent matrices of dimension $(d-1)$, each such matrix containing $O(1)$ items. There are $O(N/k)$ stripes defined for each coordinate of $\mathcal{M}$ (i.e., we have a total of d sets of $O(N/k)$ stripes each) and satisfy the size invariant of order k (Definition 2.1) on their number of matrices of dimension $(d-1)$. In this way, a stripe contains still $O(k)$ points. By intersecting these d sets of stripes, we obtain a partition of $\mathcal{M}$ into $O((N/k)^d)$ basic squares, such that each basic square overlaps no more than k matrices of dimension $(d-1)$ in each direction. We keep the items in a basic square sorted by a $\prec_B$-order to be specified and build a cross-tree $CT(T_1 \times T_2 \times \cdots \times T_d)$ on the top of these basic squares, where T_i is a weight-balanced B-tree whose leaves are in one-to-one correspondence to the stripes for the i-th coordinate ($1 \leq i \leq d$). We then store the solutions in the cross-tree nodes exactly as before. As $|T_i| = O(N/k)$, we can use an argument analogous to the one presented in Theorem 5.1 to obtain the following lemma.

Lemma 8.1 *We can insert in, delete from, concatenate and split any tree T_i in order to update cross-tree $CT(T_1 \times T_2 \times \cdots \times T_d)$ in $O(f(N)(N/k)^{d-1})$ I/Os, where $f(N) = o(N)$.*

Theorem 6.1 can be generalized to dimension d:

Theorem 8.2 *The splitting and merging problem on N items undergoing $d > 1$ total orders can be solved with the following number of I/Os, for a parameter k ($1 \leq k \leq N$), and for an operator cost $f(N) = o(N)$.*

- $O(f(N)(N/k)^{d-1} + P(k) + P(N)/N)$ *for operations range, split, and concatenate along any coordinate, with* $P(k) \geq k/B$*;*

- $O(U(k)+f(N)(\log_B(N/k)+N^{d-1}/k^d)+P(N)/N)$ *for inserting and deleting a point with operation set, with* $U(k) \geq \log_B k$*;*

The total number of blocks required is $O(S(N) + f(N)(N^d/k^dB))$ *and the total preprocessing time is* $O(Sort(N) + P(N) + f(N)(N^d/k^d))$.

Proof: By Lemma 8.1, splitting and concatenating the cross-tree along one coordinate take $O(f(N)(N/k)^{d-1})$ I/Os. Scanning a stripe takes $O(k/B)$ I/Os for retrieving its $O(k)$ items plus $O((N/k)^{d-1})$ I/Os for traversing the cross-tree. The total cost of processing a stripe is therefore $O(f(N)\ (N/k)^{d-1} + P(k))$ I/Os. With this difference in mind, the operations can be implemented and analyzed as before. For example, the term $(N^{d-1}/k^d)f(N)$ in the complexity for *set* is due to a stripe overflow or underflow which requires splitting and concatenating the cross-tree every $\Theta(k)$ insert operations. Hence, the analysis of the d-dimensional case follows directly from the two-dimensional case. □

Theorem 8.2 finds applications to the d-dimensional versions of some order decomposable problems. For example, we can tune properly k (i.e., choose $k = \lceil B^{1/d} N^{1-1/d} \rceil$) to obtain external-memory multidimensional 2-3-trees:

Theorem 8.3 *A d-dimensional 2-3-tree storing N d-dimensional points can be maintained with the following I/O bounds:*

- *a point insertion or deletion in* $O(\log_B N)$*;*

- *a split or a concatenate along any coordinate in* $O((N/B)^{1-1/d})$*;*

- *a range search in* $O((N/B)^{1-1/d} + r/B)$*, where r is the number of points reported by the search.*

The space required is $O(N/B)$ *blocks and the preprocessing cost is* $O(Sort(N))$ *I/Os.*

9 Concluding Remarks

We have described efficient methods for organizing large multidimensional data sets in external memory, and for maintaining them simultaneously sorted under several total orderings. Besides standard insertions and deletions of single items, our technique is able to perform efficiently *split* and *concatenate* operations on the whole data sets according to any ordering. Differently from other approaches, our techniques are based more on simple geometric properties rather than on underlying sophisticated data structures, and exploit the fact that some data structures can be built on (pre)sorted items more efficiently. We can make balancing easy as with the help of a new multidimensional data structure, *the* *cross-tree*, which is the cross product of balanced trees and generalizes the notion of B-trees to higher dimensions by carefully combining space-driven and data-driven partitions. As a result, the cross-tree is competitive with other popular index data structures that require linear space, such as quad-trees [15], k-d trees [7], grid files [30], space filling curves, e.g. [32], hB-trees [28], and R-trees [20].

We conclude this section with few remarks about the practicality of our approach. In our data structures, records are not replicated and the records satisfying a given query can be efficiently scanned starting from their retrieved pointers. This is not always possible with known multidimensional data structures [24] and, as a result, retrieving records might cost more than a simple sequential scan of all the available records in practical situations [25]. Our data structure does not seem to suffer from such problems, especially if the range query is implemented without explicitly performing the splits and concatenates as previously discussed. As far as the other operations are concerned, we use global rebuilding [33] for handling stripe overflows and "weak" deletions in order to guarantee worst-case bounds instead of amortized ones. In a practical setting, we would suggest to avoid global rebuilding, which is known to be expensive due to the maintenance of multiple copies of the data structures, and simply achieve the amortized bounds by periodical reconstructions of the data structure.

Another important aspect to be considered in a practical setting is the proper tuning of the slack parameter k when the main memory size M is large. Although k is chosen to minimize the asymptotic I/O complexity of our algorithms, it could be used to make level II of our data structure (the cross-tree) fit completely in main memory, and store only level I (the basic squares) in external memory. This way, level II can be used as a fast internal memory directory for the items stored in external memory (in level I). For example, supposing $f(N) = O(1)$, $M > B^2$ and $N > B^3$, we can choose $k = \Theta(N/B)$ so that the total space usage of the cross-tree is less than M, and the split/concatenate cost is dominated by scanning the stripes with $O(N/k + P(k)) = O(B + N/B^2)$ I/Os instead of $O(\sqrt{N/B})$. Although the asymptotical complexity increases, this could make a great difference in practice, as the management of the cross-tree structure is entirely in main memory and the dominant cost for scanning the stripes can be alleviated by the look-ahead buffering strategies implemented in modern computing systems.

ACKNOWLEDGMENTS. We are indebted to Lars Arge and to Paolo Ferragina for helpful discussions.

References

[1] A. Aggarwal and J.S. Vitter, *The Input/Output complexity of sorting and related problems*, Communications of the ACM **31** (1988), 1116–1127.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms.* Addison–Wesley, Reading, MA, 1974.

[3] L. Arge, P. Ferragina, R. Grossi and J.S. Vitter. *On sorting strings in external memory.* Proc. 29th ACM Symposium on the Theory of Computing (1997), pp.540–548.

[4] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J.S. Vitter. *Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems.* Proc. 9th Annual ACM-SIAM Symp. on Discrete Algorithms (1998), 685–694.

[5] L. Arge and J.S. Vitter. *Optimal dynamic interval management in external memory.* Proc. 37th IEEE Symp. on Foundations of Computer Science (1996), 560–569.

[6] R. Bayer and E.C. McCreight. *Organization and maintenance of large ordered indexes.* Acta Informatica **3** (1972), 173–189.

[7] J.L. Bentley. *Multidimensional binary search trees used for associated searching.* Communications of the ACM **19** (1975), 509–517.

[8] J.L. Bentley. *Decomposable searching problems.* Information Processing Letters **8** (1979), 244–251.

[9] J.L. Bentley and J.B. Saxe. *Decomposable searching problems I. Static–to–dynamic transformation.* Journal of Algorithms **1** (1980), 301–358.

[10] Y-J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, and J.S. Vitter. *External–memory graph algorithms.* Proc. 6th ACM–SIAM Symp. on Discrete Algorithms (1995), pp.139–149.

[11] Y.-J. Chiang and R. Tamassia. *Dynamic algorithms in computational geometry.* Proceedings of the IEEE, Special issue on Computational Geometry, G. Toussaint, ed., **80** (1992) 1412–1434.

[12] D. Comer. *The ubiquitous B-Tree.* Computing Surveys **11** (1979), 121–137.

[13] D. Eppstein, Z. Galil, G.F. Italiano, and A. Nissenzweig, *Sparsification—A technique for speeding up dynamic graph algorithms. Journal of ACM*, vol. 44 (1997), 669–696.

[14] P. Ferragina and R. Grossi. *A fully dynamic data structure for external substring search.* Proc. 27th ACM Symp. on Theory of Computing (1995), 693–702. Full version in: *An external-memory indexing data structure and its applications.* To appear in Journal of the ACM.

[15] R.A. Finkel and J.L. Bentley. *Quad–trees: A data structure for retrieval of composite keys.* Acta Informatica **4** (1974), 1–9.

[16] V. Gaede and O. Günther. *Multidimensional access methods.* To appear in ACM Computing Surveys (1998).

[17] M.T. Goodrich, M.H. Nodine, and J.S. Vitter. *Blocking for external graph searching.* Algorithmica **16** (1996), 181–214.

[18] R. Grossi and G.F. Italiano. *Efficient splitting and merging algorithms for order decomposable problems.* Proc. 24th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science (1997) Springer-Verlag, pp.605–615. Full version in Information and Computation 154(1): 1-33 (1999).

[19] R. Grossi and G.F. Italiano. *Efficient cross-trees for external memory.* In "External Memory Algorithms and Visualization", James Abello and Jeffrey Scott Vitter eds., DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society Press, Providence, RI, 1999.

[20] A. Guttman. *R-trees: A dynamic index structure for spatial searching.* Proc. ACM SIGMOD International Conference on the Management of Data (1984), pp.47–57.

[21] F. Harary. *Graph Theory.* Addison–Wesley, 1969.

[22] J.M. Hellerstein, E. Koutsopias and C.H. Papadimitriou. *On the analysis of indexing schemes.* Proc. ACM Symp. on Principles of Database Systems (1997), pp.249–256.

[23] J. Holm, K. de Lichtenberg and M. Thorup, *Poly-logarithmic deterministic fully-dynamic graph algorithms for connectivity, minimum spanning forerst, 2-edge connectivity and biconnectivity*. Proc. 30th ACM Symp. on Theory of Computing (1998).

Technical Report DIKU-TR-97/17.

[24] 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** (1996), 589–612.

[25] D.E. Knuth. *The art of computer programming*, vol.3 (sorting and searching). Reading, MA, Addison-Wesley, 1973 and 1998.

[26] M.J. van Kreveld and M.H. Overmars, *Divided k-d trees.* Algorithmica **6** (1991), 840–858.

[27] M.J. van Kreveld and M.H. Overmars. *Concatenable structures for decomposable problems.* Information and Computation **110** (1994), 130–148.

[28] D.B. Lomet and B. Salzberg. *The hB-tree: A multiattribute indexing method with good guaranteed performance.* ACM Transactions on Databases Systems **15** (1990), 625–658.

[29] K. Mehlhorn. *Multi-dimensional searching and computational geometry.* EATCS Monographs on Theoretical Computer Science, vol. 3, Springer–Verlag, Berlin/New York, 1984.

[30] J. Nievergelt, H. Hinterberger and K.C. Sevcik. *The grid file: An adaptable, symmetric multikey file structure.* ACM Transactions on Databases Systems **9** (1984), 38–71.

[31] M.H. Nodine and J.S. Vitter. *Greed Sort: Optimal deterministic sorting on parallel disks.* Journal of the ACM **42** (1995) 919–933.

[32] J.A. Orenstein. *Spatial query processing in an object-oriented database sytem.* Proc. ACM SIGMOD International Conference on the Management of Data (1986) pp.326–336.

[33] M.H. Overmars. *The design of dynamic data structures.* Lecture Notes in Computer Science 156, Springer–Verlag, Berlin/New York, 1983.

[34] M.H. Overmars and J. van Leeuwen. *Worst-case optimal insertion and deletion methods for decomposable searching problems.* Information Processing Letters **12** (1981), 168–173.

[35] M.H. Overmars and J. van Leeuwen. *Dynamic multi-dimensional data structures based on quad– and k–d trees.* Acta Informatica **17** (1982), 267–285.

[36] M.H. Overmars, M. Smid, M.T. de berg, and M.J. van Kreveld. *Maintaining range trees in secondary storage. Part I: Partitions.* Acta Informatica **27** (1990), 423–452.

[37] N. P. Patt (ed.). *The I/O subsystem: A candidate for improvement*, special issue on I/O subsystems. IEEE Computer **27** (1994), 15–16.

[38] S. Ramaswamy and S. Subramanian. *Path caching: A technique for optimal external searching.* Proc. ACM Symp. on Principles of Database Systems (1994), 25–35.

[39] K.V. Ravi Kanth and A.K. Singh. “Optimal dynamic range searching in non-replicating index structures”. Technical Report TRCS97-13, University of California at Santa Barbara, Dpetartment of Computer Science, July 1997.

[40] H. Samet. *Bibliography on quad–trees and related hierarchical data structures.* In Data Structures for Raster Graphics, L. Kessenaar, F. Peters, and M. van Lierop eds., Springer–Verlag, Berlin, (1986), pp.181–201.

[41] D. D. Sleator and R. E. Tarjan. *A data structure for dynamic trees.* Journal of Computer and System Sciences **24** (1983), 362–381.

[42] S. Subramanian and S. Ramaswamy. *The P-range tree: A new data structure for range searching in secondary memory.* Proc. 5th ACM–SIAM Annual Symp. on Discrete Algorithms (1995), pp.378–387.

[43] D.E. Vengroff and J.S. Vitter. *Efficient 3-D range searching in external memory.* Proc. 28th ACM Symp. on Theory of Computing (1996) pp.192–201.

[44] J.S. Vitter and E.A.M. Shriver. *Algorithms for parallel memory, I: Two–level memories.* Algorithmica **12** (1994), 110–147. Also Proc. 22nd ACM Symp. on Theory of Computing (1990) pp.159–169.