DIMACS Series in Discrete Mathematics
and Theoretical Computer Science

External Memory Techniques for Isosurface Extraction in Scientific Visualization

Yi-Jen Chiang and Cláudio T. Silva

Abstract. Isosurface extraction is one of the most effective and powerful techniques for the investigation of volume datasets in scientific visualization. Previous isosurface techniques are all main-memory algorithms, often not applicable to large scientific visualization applications. In this paper we survey our recent work that gives the first external memory techniques for isosurface extraction. The first technique, *I/O-filter*, uses the existing I/O-optimal interval tree as the indexing data structure (where the corner structure is not implemented), together with the isosurface engine of Vtk (one of the currently best visualization packages). The second technique improves the first version of *I/O-filter* by replacing the I/O interval tree with the metablock tree (whose corner structure is not implemented). The third method further improves the first two, by using a *two-level indexing* scheme, together with a new *meta-cell* technique and a new I/O-optimal indexing data structure (the *binary-blocked I/O interval tree*) that is simpler and more space-efficient in practice (whose corner structure is not implemented). The experiments show that the first two methods perform isosurface queries faster than Vtk by a factor of two orders of magnitude for datasets larger than main memory. The third method further reduces the disk space requirement from 7.2–7.7 times the original dataset size to 1.1–1.5 times, at the cost of slightly increasing the query time; this method also exhibits a smooth trade-off between disk space and query time.

1. Introduction

The field of computer graphics can be roughly classified into two subfields: *surface graphics*, in which objects are defined by surfaces, and *volume graphics* [**17, 18**], in which objects are given by datasets consisting of 3D sample points over their volume. In volume graphics, objects are usually modeled as *fuzzy* entities. This representation leads to greater freedom, and also makes it possible to visualize the *interior* of an object. Notice that this is almost impossible for traditional surface-graphics objects. The ability to visualize the interior of an object is particularly

1991 *Mathematics Subject Classification.* 65Y25, 68U05, 68P05, 68Q25.

Key words and phrases. Computer Graphics, Scientific Visualization, External Memory, Design and Analysis of Algorithms and Data Structures, Experimentation.

The first author was supported in part by NSF Grant DMS-9312098 and by Sandia National Labs.

The second author was partially supported by Sandia National Labs and the Dept. of Energy Mathematics, Information, and Computer Science Office, and by NSF Grant CDA-9626370.

important in *scientific visualization*. For example, we might want to visualize the internal structure of a patient's brain from a dataset collected from a computed tomography (CT) scanner, or we might want to visualize the distribution of the density of the mass of an object, and so on. Therefore volume graphics is used in virtually all scientific visualization applications. Since the dataset consists of points sampling the entire volume rather than just vertices defining the surfaces, typical volume datasets are huge. This makes volume visualization an ideal application domain for I/O techniques.

Input/Output (I/O) communication between fast internal memory and slower external memory is the major bottleneck in many large-scale applications. Algorithms specifically designed to reduce the I/O bottleneck are called *external-memory* algorithms. In this paper, we survey our recent work that gives the first external memory techniques for one of the most important problems in volume graphics: *isosurface extraction* in scientific visualization.

1.1. Isosurface Extraction. Isosurface extraction represents one of the most effective and powerful techniques for the investigation of volume datasets. It has been used extensively, particularly in visualization [**20, 22**], simplification [**14**], and implicit modeling [**23**]. Isosurfaces also play an important role in other areas of science such as biology, medicine, chemistry, computational fluid dynamics, and so on. Its widespread use makes efficient isosurface extraction a very important problem.

The problem of isosurface extraction can be stated as follows. The input dataset is a *scalar volume dataset* containing a list of tuples $(\mathbf{x}, \mathcal{F}(\mathbf{x}))$, where $\mathbf{x}$ is a 3D sample point and $\mathcal{F}$ is a scalar function defined over 3D points. The scalar function $\mathcal{F}$ is an unknown function; we only know the *sample value* $\mathcal{F}(\mathbf{x})$ at each sample point $\mathbf{x}$. The function $\mathcal{F}$ may denote temperature, density of the mass, or intensity of an electronic field, etc., depending on the applications. The input dataset also has a list of *cells* that are cubes or tetrahedra or of some other geometric type. Each cell is defined by its vertices, where each vertex is a 3D sample point $\mathbf{x}$ given in the list of tuples $(\mathbf{x}, \mathcal{F}(\mathbf{x}))$. Given an isovalue (a scalar value) q, to extract the isosurface of q is to compute and display the isosurface $C(q) = \{\mathbf{p} | \mathcal{F}(\mathbf{p}) = q\}$. Note that the isosurface point $\mathbf{p}$ may not be a sample point $\mathbf{x}$ in the input dataset: if there are two sample points with their scalar values smaller and larger than q, respectively, then the isosurface $C(q)$ will go between these two sample points via linear interpolation. Some examples of isosurfaces (generated from our experiments) are shown in Fig. 1, where the Blunt Fin dataset shows an airflow through a flat plate with a blunt fin, and the Combustion Chamber dataset comes from a combustion simulation. Typical use of isosurface is as follows. A user may ask: "display all areas with temperature equal to 25 degrees." After seeing that isosurface, the user may continue to ask: "display all areas with temperature equal to 10 degrees." By repeating this process interactively, the user can study and perform detailed measurements of the properties of the datasets. Obviously, to use isosurface extraction effectively, it is crucial to achieve fast interactivity, which requires efficient computation of isosurface extraction.

The computational process of isosurface extraction can be viewed as consisting of two phases (see Fig. 2). First, in the *search phase*, one finds all *active* cells of the dataset that are intersected by the isosurface. Next, in the *generation phase*, depending on the type of cells, one can apply an algorithm to actually generate the

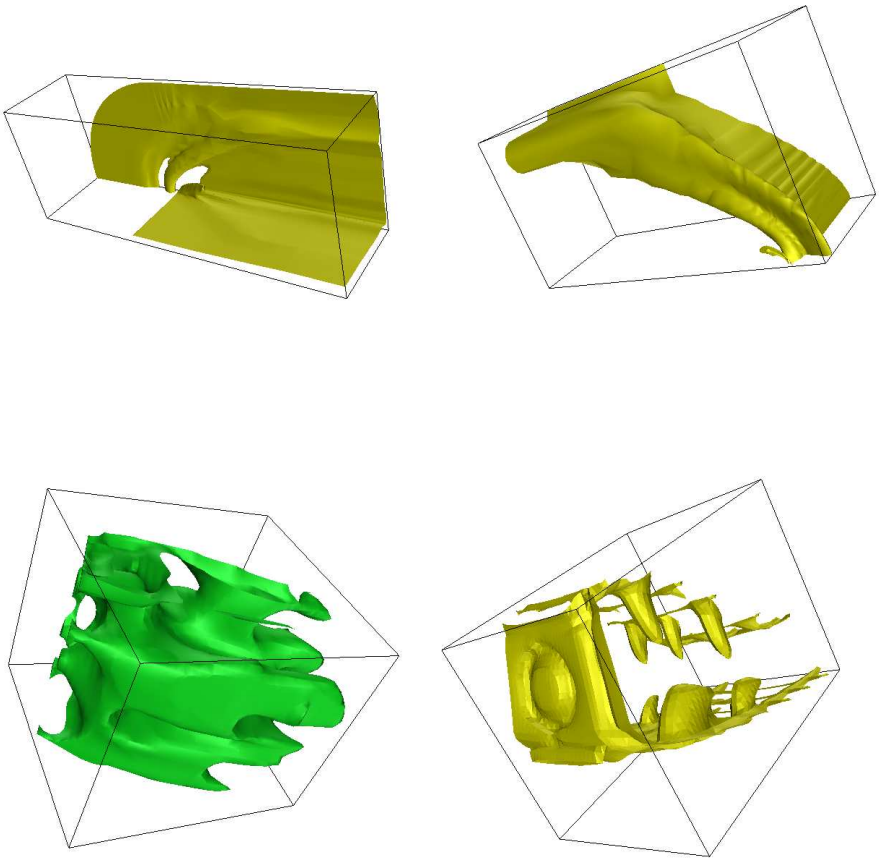

FIGURE 1. Typical isosurfaces are shown. The upper two are for the Blunt Fin dataset. The ones in the bottom are for the Combustion Chamber dataset.

isosurface from those active cells (Marching Cubes [**20**] is one such algorithm for hexahedral cells). Notice that the search phase is usually the bottleneck of the entire process, since it searches the 3D dataset and produces 2D data. In fact, letting N be the total number of cells in the dataset and K the number of active cells, it is estimated that the typical value of K is $O(N^{2/3})$ [**15**]. Therefore an exhaustive scanning of all cells in the search phase is inefficient, and a lot of research efforts have thus focused on developing *output-sensitive* algorithms to speed up the search phase.

In the rest of the paper we use N and K to denote the total number of cells in the dataset and the number of active cells, respectively, and M and B to respectively denote the numbers of cells fitting in main memory and in a disk block. Each I/O operation reads or writes one disk block.

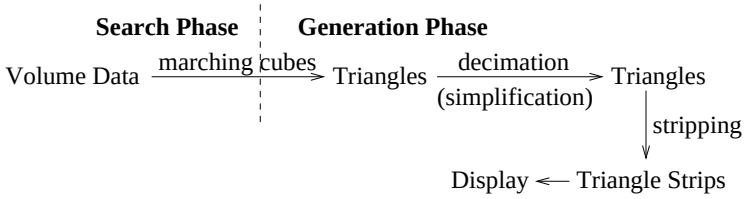

FIGURE 2. A pipeline of the isosurface extraction process.

1.2. Overview of Main Memory Isosurface Techniques. There is a very rich literature for isosurface extraction. Here we only briefly review the results that focus on speeding up the search phase. For an excellent and thorough review, see [**19**].

In Marching Cubes [**20**], all cells in the volume dataset are searched for isosurface intersection, and thus $O(N)$ time is needed. Concerning the main memory issue, this technique does not require the entire dataset to fit into main memory, but $\lceil N/B \rceil$ disk reads are necessary. Wilhems and Van Gelder [**30**] propose a method of using an octree to optimize isosurface extraction. This algorithm has worst-case time of $O(K + K \log(N/K))$ (this analysis is presented by Livnat *et al.* [**19**]) for isosurface queries, once the octree has been built.

Itoh and Kayamada [**15**] propose a method based on identifying a collection of *seed cells* from which isosurfaces can be propagated by performing local search. Basically, once the seed cells have been identified, they claim to have a nearly $O(N^{2/3})$ expected performance. (Livnat *et al.* [**19**] estimate the worst-case running time to be $O(N)$, with a high memory overhead.) More recently, Bajaj *et al.* [**3**] propose another contour propagation scheme, with expected performance of $O(K)$. Livnat *et al.* [**19**] propose NOISE, an $O(\sqrt{N} + K)$-time algorithm. Shen *et al.* [**27, 28**] also propose nearly optimal isosurface extraction methods.

The first *optimal* isosurface extraction algorithm was given by Cignoni *et al.* [**11**], based on the following two ideas. First, for each cell, they produce an interval $I = [\min, \max]$ where min and max are the minimum and maximum of the scalar values in the cell vertices. Then the active cells are exactly those cells whose intervals contain q. Searching active cells then amounts to performing the following *stabbing queries*: Given a set of 1D intervals, report all intervals (and the associated cells) containing the given query point q. Secondly, the stabbing queries are solved by using an internal-memory interval tree [**12**]. After an $O(N \log N)$-time preprocessing, active cells can be found in optimal $O(\log N + K)$ time.

All the isosurface techniques mentioned above are main-memory algorithms. Except for the inefficient exhaustive scanning method of Marching Cubes, all of them require the time and main memory space to read and keep the entire dataset in main memory, plus additional preprocessing time and main memory space to build and keep the search structure. Unfortunately, for (usually) very large volume datasets, these methods often suffer the problem of not having enough main memory, which can cause a major slow-down of the algorithms due to a large number of page faults. Another issue is that the methods need to load the dataset into main memory and build the search structure each time we start the running process. This start-up cost can be very expensive since loading a large volume dataset from disk is very time-consuming.

1.3. Summary of External Memory Isosurface Techniques. In [**8**] we give *I/O-filter*, the first I/O-optimal technique for isosurface extraction. We follow the ideas of Cignoni *et al.* [**11**], but use the I/O-optimal interval tree of Arge and Vitter [**2**] as an indexing structure to solve the stabbing queries. This enables us to find the active cells in optimal $O(\log_B N + K/B)$ I/O's. We give the first implementation of the I/O interval tree (where the *corner structure* is not implemented, which may result in non-optimal disk space and non-optimal query I/O cost in the worst case), and also implement our method as an *I/O filter* for the isosurface extraction routine of Vtk [**24, 25**] (which is one of the currently best visualization packages). The experiments show that the isosurface queries are faster than Vtk by a factor of two orders of magnitude for datasets larger than main memory. In fact, the search phase is no longer a bottleneck, and the performance is *independent* of the main memory available. Also, the preprocessing is performed only *once* to build an indexing structure in disk, and later on there is no start-up cost for running the query process. The major drawback is the overhead in disk scratch space and the preprocessing time necessary to build the search structure, and of the disk space needed to hold the data structure.

In [**9**], we give the second version of *I/O-filter*, by replacing the I/O interval tree [**2**] with the metablock tree of Kanellakis *et al.* [**16**]. We give the first implementation of the metablock tree (where the corner structure is not implemented to reduce the disk space; this may result in non-optimal query I/O cost in the worst case). While keeping the query time the same as in [**8**], the tree construction time, the disk space and the disk scratch space are all improved.

In [**10**], at the cost of slightly increasing the query time, we greatly improve all the other cost measures. In the previous methods [**8, 9**], the *direct vertex information* is duplicated many times; in [**10**], we avoid such duplications by employing a *two-level indexing* scheme. We use a new *meta-cell* technique and a new I/O-optimal indexing data structure (the *binary-blocked I/O interval tree*) that is simpler and more space-efficient in practice (where the corner structure is not implemented, which may result in non-optimal I/O cost for the stabbing queries in the worst case). Rather than fetching only the active cells into main memory as in *I/O-filter* [**8, 9**], this method fetches the set of *active meta-cells*, which is a *superset* of all active cells. While the query time is still at least one order of magnitude faster than Vtk, the disk space is reduced from 7.2–7.7 times the original dataset size to 1.1–1.5 times, and the disk scratch space is reduced from 10–16 times to less than 2 times. Also, instead of being a single-cost indexing approach, the method exhibits a smooth trade-off between disk space and query time.

1.4. Organization of the Paper. The rest of the paper is organized as follows. In Section 2, we review the I/O-optimal data structures for stabbing queries, namely the metablock tree [**16**] and the I/O interval tree [**2**] that are used in the two versions of our I/O-filter technique, and the binary-blocked I/O-interval tree [**10**] that is used in our two-level indexing scheme. The preprocessing algorithms and the implementation issues, together with the dynamization of the binary-blocked I/O interval tree (which is not given in [**10**] and may be of independent interest; see Section 2.3.3) are also discussed. We describe the I/O-filter technique [**8, 9**] and summarize the experimental results of both versions in Section 3. In Section 4 we survey the two-level indexing scheme [**10**] together with the experimental results. Finally we conclude the paper in Section 5.

2. I/O Optimal Data Structures for Stabbing Queries

In this section we review the metablock tree [**16**], the I/O interval tree [**2**], and the binary-blocked I/O interval tree [**10**]. The metablock tree is an external-memory version of the priority search tree [**21**]. The static version of the metablock tree is the first I/O-optimal data structure for *static* stabbing queries (where the set of intervals is fixed); its *dynamic* version only supports insertions of intervals and the update I/O cost is not optimal. The I/O interval tree is an external-memory version of the main-memory interval tree [**12**]. In addition to being I/O-optimal for the static version, the dynamic version of the I/O interval tree is the first I/O-optimal *fully dynamic* data structure that also supports insertions and deletions of intervals with optimal I/O cost. Both the metablock tree and the I/O interval tree have three kinds of secondary lists and each interval is stored up to three times in practice. Motivated by the practical concern on the disk space and the simplicity of coding, the binary-blocked I/O interval tree is an alternative external-memory version of the main-memory interval tree [**12**] with only two kinds of secondary lists. In practice, each interval is stored twice and hence the tree is more space-efficient and simpler to implement. We remark that the powerful dynamization techniques of [**2**] for dynamizing the I/O interval tree can also be applied to the binary-blocked I/O interval tree to support I/O-optimal updates (amortized rather than worst-case bounds as in the I/O interval tree — although we believe the techniques of [**2**] can further turn the bounds into worst-case, we did not verify the details. See Section 2.3.3). For our application of isosurface extraction, however, we only need the *static* version, and thus all three trees are I/O-optimal. We only describe the static version of the trees (but in addition we discuss the dynamization of the binary-blocked I/O interval tree in Section 2.3.3). We also describe the preprocessing algorithms that we used in [**8, 9, 10**] to build these static trees, and discuss their implementation issues.

Ignoring the cell information associated with the intervals, we now use M and B to respectively denote the numbers of intervals that fit in main memory and in a disk block. Recall that N is the total number of intervals, and K is the number of intervals reported from a query. We use Bf to denote the branching factor of a tree.

2.1. Metablock Tree.

2.1.1. *Data Structure.* We briefly review the metablock tree data structure [**16**], which is an external-memory version of the priority search tree [**21**]. The stabbing query problem is solved in the *dual space*, where each interval $[left, right]$ is mapped to a dual point (x, y) with $x = left$ and $y = right$. Then the query "find intervals $[x, y]$ with $x \le q \le y$" amounts to the following *two-sided orthogonal range query* in the dual space: report all dual points (x, y) lying in the intersection of the half planes $x \le q$ and $y \ge q$. Observe that all intervals $[left, right]$ have $left \le right$, and thus all dual points lie in the half plane $x \le y$. Also, the "corner" induced by the two sides of the query is the dual point (q, q), so all query corners lie on the line $x = y$.

The metablock tree stores dual points in the same spirit as a priority search tree, but increases the branching factor Bf from 2 to $\Theta(B)$ (so that the tree height is reduced from $O(\log_2 N)$ to $O(\log_B N)$), and also stores $Bf \cdot B$ points in each tree node. The main structure of a metablock tree is defined recursively as follows (see Fig. 3(a)): if there are no more than $Bf \cdot B$ points, then all of them are assigned to

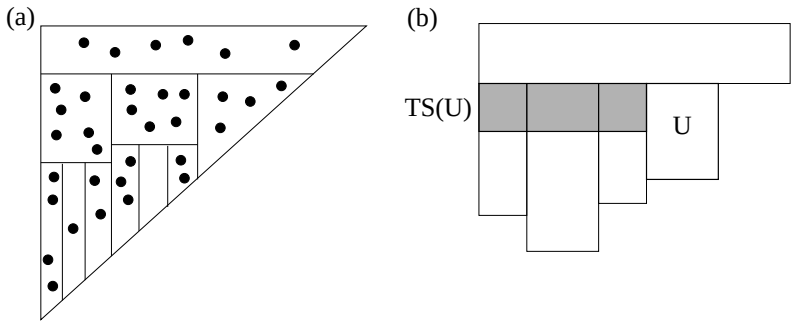

FIGURE 3. A schematic example of metablock tree: (a) the main structure; (b) the TS list. In (a), $Bf = 3$ and $B = 2$, so each node has up to 6 points assigned to it. We relax the requirement that each vertical slab have the same number of points.

the current node, which is a leaf; otherwise, the topmost $Bf \cdot B$ points are assigned to the current node, and the remaining points are distributed by their x-coordinates into Bf vertical slabs, each containing the same number of points. Now the Bf subtrees of the current node are just the metablock trees defined on the Bf vertical slabs. The $Bf-1$ slab boundaries are stored in the current node as keys for deciding which child to go during a search. Notice that each internal node has no more than Bf children, and there are Bf blocks of points assigned to it. For each node, the points assigned to it are stored twice, respectively in two lists in disk of the same size: the *horizontal* list, where the points are horizontally blocked and stored sorted by decreasing y-coordinates, and the *vertical* list, where the points are vertically blocked and stored sorted by increasing x-coordinates. We use unique dual point ID's to break a tie. Each node has two pointers to its *horizontal* and *vertical* lists. Also, the "bottom" (*i.e.*, the y-value of the bottommost point) of the *horizontal* list is stored in the node.

The second piece of organization is the TS list maintained in disk for each node U (see Fig. 3(b)): the list $TS(U)$ has at most Bf blocks, storing the topmost Bf blocks of points from *all left siblings* of U (if there are fewer than $Bf \cdot B$ points then all of them are stored in $TS(U)$). The points in the TS list are horizontally blocked, stored sorted by decreasing y-coordinates. Again each node has a pointer to its TS list, and also stores the "bottom" of the TS list.

The final piece of organization is the *corner structure*. A corner structure can store $t = O(B^2)$ points in optimal $O(t/B)$ disk blocks, so that a two-sided orthogonal range query can be answered in optimal $O(k/B + 1)$ I/O's, where k is the number of points reported. Assuming all t points can fit in main memory during preprocessing, a corner structure can be built in optimal $O(t/B)$ I/O's. We refer to [**16**] for more details. In a metablock tree, for each node U where a query corner can possibly lie, a corner structure is built for the ($\leq Bf \cdot B = O(B^2)$) points assigned to U (recall that $Bf = \Theta(B)$). Since any query corner must lie on the line $x = y$, each of the following nodes needs a corner structure: (1) the leaves, and (2) the nodes in the rightmost root-to-leaf path, including the root (see Fig. 3(a)). It is easy to see that the entire metablock tree has height $O(\log_{Bf}(N/B)) = O(\log_B N)$ and uses optimal $O(N/B)$ blocks of disk space [**16**]. Also, it can be seen that the

corner structures are *additional* structures to the metablock tree; we can save some storage space by not implementing the corner structures (at the cost of increasing the worst-case query bound; see Section 2.1.2).

As we shall see in Section 2.1.3, we will slightly modify the definition of the metablock tree to ease the task of preprocessing, while keeping the bounds of tree height and tree storage space the same.

2.1.2. *Query Algorithm.* Now we review the query algorithm given in [**16**]. Given a query value q, we perform the following recursive procedure starting with *meta-query* (q, the root of the metablock tree). Recall that we want to report all dual points lying in $x \le q$ and $y \ge q$. We maintain the invariant that the current node U being visited always has its x-range containing the vertical line $x = q$.

Procedure *meta-query* (query q, node U)

1. If U contains the corner of q, *i.e.*, the bottom of the *horizontal* list of U is lower than the horizontal line $y = q$, then use the corner structure of U to answer the query and stop.
2. Otherwise ($y(bottom(U)) \ge q$), all points of U are above or on the horizontal line $y = q$. Report all points of U that are on or to the left of the vertical line $x = q$, using the *vertical* list of U.
3. Find the child U_c (of U) whose x-range contains the vertical line $x = q$. The node U_c will be the next node to be recursively visited by *meta-query*.
4. Before recursively visiting U_c, take care of the left-sibling subtrees of U_c first (points in all these subtrees are on or to the left of the vertical line $x = q$, and thus it suffices to just check their heights):
   (a) If the bottom of $TS(U_c)$ is lower than the horizontal line $y = q$, then report the points in $TS(U_c)$ that lie inside the query range. Go to step 5.
   (b) Else, for each left sibling W of U_c, repeatedly call procedure *H-report* (query q, node W). (*H-report* is another recursive procedure given below.)
5. Recursively call *meta-query* (query q, node U_c).

H-report is another recursive procedure for which we maintain the invariant that the current node W being visited have all its points lying on or to the left of the vertical line $x = q$, and thus we only need to consider the condition $y \ge q$.

Procedure *H-report* (query q, node W)

1. Use the *horizontal* list of W to report all points of W lying on or above the horizontal line $y = q$.
2. If the bottom of W is lower than the line $y = q$ then stop.
   Otherwise, for each child V of W, repeatedly call *H-report* (query q, node V) recursively.

It can be shown that the queries are performed in optimal $O(\log_B N + \frac{K}{B})$ I/O's [**16**]. We remark that only one node in the search path would possibly use its corner structure to report its points lying in the query range since there is at most one node containing the query corner (q, q). If we do not implement the corner structure, then step 1 of Procedure *meta-query* can still be performed by checking the *vertical* list of U up to the point where the current point lies to the right of the vertical line $x = q$ and reporting all points thus checked with $y \ge q$. This might perform extra Bf I/O's to examine the entire *vertical* list without reporting any point, and hence is not optimal. However, if $K \ge \alpha \cdot (Bf \cdot B)$ for some constant

$\alpha < 1$ then this is still worst-case I/O-optimal since we need to perform $\Omega(Bf)$ I/O's to just report the answer.

2.1.3. *Preprocessing Algorithms.* Now we describe a preprocessing algorithm proposed in [**9**] to build the metablock tree. It is based on a paradigm we call *scan and distribute*, inspired by the *distribution sweep* I/O technique [**6, 13**]. The algorithm relies on a slight modification of the definition of the tree.

In the original definition of the metablock tree, the vertical slabs for the subtrees of the current node are defined by dividing the *remaining* points not assigned to the current node into *Bf* groups. This makes the distribution of the points into the slabs more difficult, since in order to assign the topmost *Bf* blocks to the current node we have to sort the points by y-values, and yet the slab boundaries (x-values) from the remaining points cannot be directly decided. There is a simple way around it: we first sort all N points by increasing x-values into a fixed set X. Now X is used to decide the slab boundaries: the root corresponds to the entire x-range of X, and each child of the root corresponds to an x-range spanned by consecutive $|X|/Bf$ points in X, and so on. In this way, the slab boundaries of the entire metablock tree is *pre-fixed*, and the tree height is still $O(\log_B N)$.

With this modification, it is easy to apply the *scan and distribute* paradigm. In the first phase, we sort all points into the set X as above and also sort all points by decreasing y-values into a set Y. Now the second phase is a recursive procedure. We assign the first *Bf* blocks in the set Y to the root (and build its *horizontal* and *vertical* lists), and scan the remaining points to distribute them to the vertical slabs of the root. For each vertical slab we maintain a temporary list, which keeps one block in main memory as a *buffer* and the remaining blocks in disk. Each time a point is distributed to a slab, we put that point into the corresponding buffer; when the buffer is full, it is written to the corresponding list in disk. When all points are scanned and distributed, each temporary list has all its points, automatically sorted by decreasing y. Now we build the TS lists for child nodes U_0, $U_1, \cdots$ numbered left to right. Starting from U_1, $TS(U_i)$ is computed by merging two sorted lists in decreasing y and taking the first *Bf* blocks, where the two lists are $TS(U_{i-1})$ and the temporary list for slab $i-1$, both sorted in decreasing y. Note that for the initial condition $TS(U_0) = \emptyset$. (It suffices to consider $TS(U_{i-1})$ to take care of all points in slabs $0, 1, \cdots, i-2$ that can possibly enter $TS(U_i)$, since each TS list contains up to *Bf* blocks of points.) After this, we apply the procedure recursively to each slab. When the current slab contains no more than *Bf* blocks of points, the current node is a leaf and we stop. The corner structures can be built for appropriate nodes as the recursive procedure goes. It is easy to see that the entire process uses $O(\frac{N}{B}\log_B N)$ I/O's. Using the same technique that turns the nearly-optimal $O(\frac{N}{B}\log_B N)$ bound to optimal in building the static I/O interval tree [**1**], we can turn this nearly-optimal bound to optimal $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$. (For the metablock tree, this technique basically builds a $\Theta(M/B)$-fan-out tree and converts it into a $\Theta(B)$-fan-out tree during the tree construction; we omit the details here.) Another I/O-optimal preprocessing algorithm is described in [**9**].

2.2. I/O Interval Tree. In this section we describe the I/O interval tree [**2**]. Since the I/O interval tree and the binary-blocked I/O interval tree [**10**] (see Section 2.3) are both external-memory versions of the (main memory) binary interval tree [**12**], we first review the binary interval tree T.

Given a set of N intervals, such interval tree T is defined recursively as follows. If there is only one interval, then the current node r is a leaf containing that interval. Otherwise, r stores as a key the median value m that partitions the interval endpoints into two slabs, each having the same number of endpoints that are smaller (resp. larger) than m. The intervals that contain m are assigned to the node r. The intervals with both endpoints smaller than m are assigned to the left slab; similarly, the intervals with both endpoints larger than m are assigned to the right slab. The left and right subtrees of r are recursively defined as the interval trees on the intervals in the left and right slabs, respectively. In addition, each internal node u of T has two secondary lists: the *left list*, which stores the intervals assigned to u, sorted in *increasing left endpoint values*, and the *right list*, which stores the same set of intervals, sorted in *decreasing right endpoint values*. It is easy to see that the tree height is $O(\log_2 N)$. Also, each interval is assigned to exactly one node, and is stored either twice (when assigned to an internal node) or once (when assigned to a leaf), and thus the overall space is $O(N)$.

To perform a query for a query point q, we apply the following recursive process starting from the root of T. For the current node u, if q lies in the left slab of u, we check the left list of u, reporting the intervals sequentially from the list until the first interval is reached whose left endpoint value is larger than q. At this point we stop checking the left list since the remaining intervals are all to the right of q and cannot contain q. We then visit the left child of u and perform the same process recursively. If q lies in the right slab of u then we check the right list in a similar way and then visit the right child of u recursively. It is easy to see that the query time is optimal $O(\log_2 N + K)$.

2.2.1. *Data Structure.* Now we review the I/O interval tree data structure [**2**]. Each node of the tree is one block in disk, capable of holding $\Theta(B)$ items. The main goal is to increase the tree fan-out Bf so that the tree height is $O(\log_B N)$ rather than $O(\log_2 N)$. In addition to having left and right lists, a new kind of secondary lists, the *multi* lists, is introduced, to store the intervals assigned to an internal node u that *completely span* one or more vertical slabs associated with u. Notice that when $Bf = 2$ (*i.e.*, in the binary interval tree) there are only two vertical slabs associated with u and thus no slab is completely spanned by any interval. As we shall see below, there are $\Theta(Bf^2)$ *multi* lists associated with u, requiring $\Theta(Bf^2)$ pointers from u to the secondary lists, therefore Bf is taken to be $\Theta(\sqrt{B})$.

We describe the I/O interval tree in more details. Let E be the set of $2N$ endpoints of all N intervals, sorted from left to right in increasing values; E is *pre-fixed* and will be used to define the *slab boundaries* for each internal node of the tree. Let S be the set of all N intervals. The I/O interval tree on S and E is defined recursively as follows. The root u is associated with the entire range of E and with all intervals in S. If S has no more than B intervals, then u is a leaf storing all intervals of S. Otherwise u is an internal node. We evenly divide E into Bf *slabs* $E_0, E_1, \cdots, E_{Bf-1}$, each containing the same number of endpoints. The $Bf-1$ *slab boundaries* are the first endpoints of slabs $E_1, \cdots, E_{Bf-1}$; we store these slab boundaries in u as keys. Now consider each interval I in S (see Fig. 4). If I crosses one or more slab boundaries of u, then I is assigned to u and is stored in the secondary lists of u. Otherwise I completely lies inside some slab E_i and is assigned to the subset S_i of S. We associate each child u_i of u with the slab E_i and with the set S_i of intervals. The subtree rooted at u_i is recursively defined as the I/O interval tree on E_i and S_i.

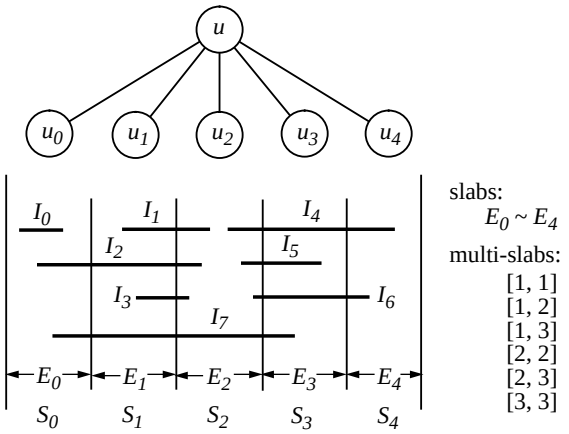

FIGURE 4. A schematic example of the I/O interval tree for the branching factor $Bf = 5$. Note that this is not a complete example and some intervals are not shown. Consider only the intervals shown here and node u. The interval sets for its children are: $S_0 = \{I_0\}$, and $S_1 = S_2 = S_3 = S_4 = \emptyset$. Its *left* lists are: $left(0) = (I_2, I_7)$, $left(1) = (I_1, I_3)$, $left(2) = (I_4, I_5, I_6)$, and $left(3) = left(4) = \emptyset$ (each list is an *ordered* list as shown). Its *right* lists are: $right(0) = right(1) = \emptyset$, $right(2) = (I_1, I_2, I_3)$, $right(3) = (I_5, I_7)$, and $right(4) = (I_4, I_6)$ (again each list is an *ordered* list as shown). Its *multi* lists are: $multi([1,1]) = \{I_2\}$, $multi([1,2]) = \{I_7\}$, $multi([1,3]) = multi([2,2]) = multi([2,3]) = \emptyset$, and $multi([3,3]) = \{I_4, I_6\}$.

For each internal node u, we use three kinds of secondary lists for storing the intervals assigned to u: the *left, right* and *multi* lists, described as follows.

For each of the Bf slabs associated with u, there is a *left* list and a *right* list; the *left* list stores all intervals belonging to u whose left endpoints lie in that slab, sorted in *increasing left endpoint values*. The *right* list is symmetric, storing all intervals belonging to u whose right endpoints lie in that slab, sorted in *decreasing right endpoint values* (see Fig. 4).

Now we describe the third kind of secondary lists, the *multi* lists. There are $(Bf-1)(Bf-2)/2$ *multi* lists for u, each corresponding to a *multi-slab* of u. A *multi-slab* $[i,j]$, $0 \le i \le j \le Bf-1$, is defined to be the union of slabs $E_i, \cdots, E_j$. The *multi* list for the multi-slab $[i,j]$ stores all intervals of u that *completely span* $E_i \cup \cdots \cup E_j$, *i.e.*, all intervals of u whose left endpoints lie in E_{i-1} and whose right endpoints lie in E_{j+1}. Since the *multi* lists $[0,k]$ for any k and the *multi* lists $[\ell, Bf-1]$ for any ℓ are always empty by the definition, we only care about multi-slabs $[1,1], \cdots, [1,Bf-2], [2,2], \cdots, [2,Bf-2], \cdots, [i,i], \cdots, [i,Bf-2], \cdots, [Bf-2,Bf-2]$. Thus there are $(Bf-1)(Bf-2)/2$ such multi-slabs and the associated *multi* lists (see Fig. 4).

For each *left*, *right*, or *multi* list, we store the entire list in consecutive blocks in disk, and in the node u (occupying one disk block) we store a pointer to the starting position of the list in disk. Since in u there are $O(Bf^2) = O(B)$ such pointers, they can all fit into one disk block, as desired.

It is easy to see that the tree height is $O(\log_{Bf}(N/B)) = O(\log_B N)$. Also, each interval I belongs to exactly one node, and is stored at most three times. If I belongs to a leaf node, then it is stored only once; if it belongs to an internal node, then it is stored once in some *left* list, once in some *right* list, and possibly one more time in some *multi* list. Therefore we need *roughly* $O(N/B)$ disk blocks to store the entire data structure.

Theoretically, however, we may need more disk blocks. The problem is because of the *multi* lists. In the worst case, a *multi* list may have only very few ($<< B$) intervals in it, but still requires one disk block for storage. The same situation may occur also for the *left* and *right* lists, but since each internal node has Bf *left*/*right* lists and the same number of children, these *underflow* blocks can be charged to the child nodes. But since there are $O(Bf^2)$ *multi* lists for an internal node, this charging argument does not work for the *multi* lists. In [**2**], the problem is solved by using the *corner structure* of [**16**] that we mentioned at the end of Section 2.1.1.

The usage of the corner structure in the I/O interval tree is as follows. For each of the $O(Bf^2)$ *multi* lists of an internal node, if there are at least $B/2$ intervals, we directly store the list in disk as before; otherwise, the ($< B/2$) intervals are maintained in a corner structure associated with the internal node. Observe that there are $O(B^2)$ intervals maintained in a corner structure. Assuming that all such $O(B^2)$ intervals can fit in main memory, the corner structure can be built with optimal I/O's (see Section 2.1.1). In summary, the height of the I/O interval tree is $O(\log_B N)$, and using the corner structure, the space needed is $O(N/B)$ blocks in disk, which is worst-case optimal.

2.2.2. *Query Algorithm.* We now review the query algorithm given in [**2**], which is very simple. Given a query value q, we perform the following recursive process starting from the root of the interval tree. For the current node u that we want to visit, we read it from disk. If u is a leaf, we just check the $O(B)$ intervals stored in u, report those intervals containing q and stop. If u is an internal node, we perform a binary search for q on the keys (slab boundaries) stored in u to identify the slab E_i containing q. Now we want to report all intervals belonging to u that contain q. We check the *left* list associated with E_i, report the intervals sequentially until we reach some interval whose *left* endpoint value is *larger* than q. Recall that each *left* list is sorted by increasing left endpoint values. Similarly, we check the *right* list associated with E_i. This takes care of all intervals belonging to u whose *endpoints* lie in E_i. Now we also need to report all intervals that *completely span* E_i. We carry out this task by reporting the intervals in the *multi* lists $multi([\ell, r])$, where $1 \le \ell \le i$ and $i \le r \le Bf-2$. Finally, we visit the i-th child u_i of u, and recursively perform the same process on u_i.

Since the height of the tree is $O(\log_B N)$, we only visit $O(\log_B N)$ nodes of the tree. We also visit the *left, right,* and *multi* lists for reporting intervals. Let us discuss the theoretically worst-case situations about the underflow blocks in the lists. An underflow block in the *left* or *right* list is fine. Since we only visit one *left* list and one *right* list per internal node, we can charge this $O(1)$ I/O cost to that internal node. But this charging argument does not work for the *multi* lists, since we may visit $\Theta(B)$ *multi* lists for an internal node. This problem is again solved in [**2**] by using the corner structure of [**16**] as mentioned at the end of Section 2.2.1. The underflow *multi* lists of an internal node u are not accessed, but are collectively taken care of by performing a query on the corner structure of u. Thus the query algorithm achieves $O(\log_B N + K/B)$ I/O operations, which is worst-case optimal.

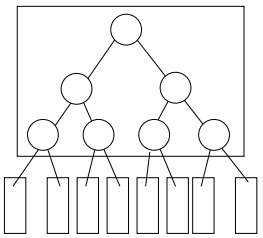

FIGURE 5. Intuition of a binary-blocked I/O interval tree $\mathcal{T}$: each circle is a node in the binary interval tree T, and each rectangle, which blocks a subtree of T, is a node of $\mathcal{T}$.

2.2.3. *Preprocessing Algorithms.* In [**8**] we used the *scan and distribute* paradigm to build the I/O interval tree. Since the algorithmic and implementation issues of the method are similar to those of the preprocessing method for the binary-blocked I/O interval tree [**10**], we omit the description here and refer to Section 2.3.4. Other I/O-optimal preprocessing algorithms for the I/O interval tree are described in [**8**].

2.3. Binary-Blocked I/O Interval Tree. Now we present our binary-blocked I/O interval tree [**10**]. It is an external-memory extension of the (main memory) binary interval tree [**12**]. Similar to the I/O interval tree [**2**] (see Section 2.2), the tree height is reduced from $O(\log_2 N)$ to $O(\log_B N)$, but the branching factor Bf is $\Theta(B)$ rather than $\Theta(\sqrt{B})$. Also, the tree does not introduce any *multi* lists, so it is simpler to implement and also is more space-efficient (by a factor of 2/3) in practice.

2.3.1. *Data Structure.* Recall the binary interval tree [**12**], denoted by T, described at the beginning of Section 2.2. We use $\mathcal{T}$ to denote our binary-blocked I/O interval tree. Each node in $\mathcal{T}$ is one disk block, capable of holding B items. We want to increase the branching factor Bf so that the tree height is $O(\log_B N)$. The intuition of our method is extremely simple — we *block* a subtree of the binary interval tree T into one node of $\mathcal{T}$ (see Fig. 5). In the following, we refer to the nodes of T as *small nodes*. We take Bf to be $\Theta(B)$. Then in an internal node of $\mathcal{T}$, there are $Bf-1$ small nodes, each having a key, a pointer to its left list and a pointer to its right list, where all left and right lists are stored in disk.

Now we give a more formal definition of the tree $\mathcal{T}$. First, we sort all *left* endpoints of the N intervals in increasing order from left to right, into a set E. We use interval ID's to break ties. The set E is used to define the keys in small nodes. Then $\mathcal{T}$ is recursively defined as follows. If there are no more than B intervals, then the current node u is a leaf node storing all intervals. Otherwise, u is an internal node. We take $Bf-1$ median values from E, which partition E into Bf slabs, each with the same number of endpoints. We store sorted, in non-decreasing order, these $Bf-1$ median values in the node u, which serve as the keys of the $Bf-1$ small nodes in u. We *implicitly* build a subtree of T on these $Bf-1$ small nodes, by a *binary-search scheme* as follows. The root key is the median of the $Bf-1$ sorted keys, the key of the left child of the root is the median of the lower half keys, and the right-child key is the median of the upper half keys, and so on. Now consider the intervals. The intervals that contain one or more keys of u are assigned to u. In fact, each such interval I is assigned to the *highest* small node (in the subtree in

u) whose key is contained in I; we store I in the corresponding left and right lists of that small node in u. For the remaining intervals, each has both endpoints in the same slab and is assigned to that slab. We recursively define the Bf subtrees of the node u as the binary-blocked I/O interval trees on the intervals in the Bf slabs.

Notice that with the above binary-search scheme for implicitly building a (sub)tree on the keys stored in an internal node u, Bf does not need to be a power of 2 — we can make Bf as large as possible, as long as the $Bf-1$ keys, the $2(Bf-1)$ pointers to the left and right lists, and the Bf pointers to the children, etc., can all fit into one disk block.

It is easy to see that $\mathcal{T}$ has height $O(\log_B N)$: $\mathcal{T}$ is defined on the set E with N left endpoints, and is perfectly balanced with $Bf = \Theta(B)$. To analyze the space complexity, observe that there are no more than N/B leaves and thus $O(N/B)$ disk blocks for the tree nodes of $\mathcal{T}$. For the secondary lists, as in the binary interval tree T, each interval is stored either once or twice. The only issue is that a left (right) list may have very few ($<< B$) intervals but still needs one disk block for storage. We observe that an internal node u has $2(Bf-1)$ left plus right lists, *i.e.*, at most $O(Bf)$ such *underfull* blocks. But u also has Bf children, and thus we can charge the underfull blocks to the child blocks. Therefore the overall space complexity is optimal $O(N/B)$ disk blocks.

As we shall see in Section 2.3.2, the above data structure supports queries in non-optimal $O(\log_2 \frac{N}{B} + K/B)$ I/O's, and we can use the *corner structures* to achieve optimal $O(\log_B N + K/B)$ I/O's while keeping the space complexity optimal.

2.3.2. *Query Algorithm.* The query algorithm for the binary-blocked I/O interval tree $\mathcal{T}$ is very simple and mimics the query algorithm for the binary interval tree T. Given a query point q, we perform the following recursive process starting from the root of $\mathcal{T}$. For the current node u, we read u from disk. Now consider the subtree T_u implicitly built on the small nodes in u by the binary-search scheme. Using the same binary-search scheme, we follow a root-to-leaf path in T_u. Let r be the current small node of T_u being visited, with key value m. If $q = m$, then we report all intervals in the left (or equivalently, right) list of r and stop. (We can stop here for the following reasons. (1) Even some descendent of r has the same key value m, such descendent must have empty left and right lists, since if there are intervals containing m, they must be assigned to r (or some small node higher than r) before being assigned to that descendent. (2) For any non-empty descendent of r, the stored intervals are either entirely to the left or entirely to the right of $m = q$, and thus cannot contain q.) If $q < m$, we scan and report the intervals in the left list of r, until the first interval with the left endpoint larger than q is encountered. Recall that the left lists are sorted by increasing left endpoint values. After that, we proceed to the left child of r in T_u. Similarly, if $q > m$, we scan and report the intervals in the right list of r, until the first interval with the right endpoint smaller than q is encountered. Then we proceed to the right child of r in T_u. At the end, if q is not equal to any key in T_u, the binary search on the $Bf-1$ keys locates q in one of the Bf slabs. We then visit the child node of u in $\mathcal{T}$ which corresponds to that slab, and apply the same process recursively. Finally, when we reach a leaf node of $\mathcal{T}$, we check the $O(B)$ intervals stored to report those that contain q, and stop.

Since the height of the tree $\mathcal{T}$ is $O(\log_B N)$, we only visit $O(\log_B N)$ nodes of $\mathcal{T}$. We also visit the left and right lists for reporting intervals. Since we always report the intervals in an *output-sensitive* way, this reporting cost is roughly $O(K/B)$.

However, it is possible that we spend one I/O to read the first block of a left/right list but only very few ($<< B$) intervals are reported. In the worst case, all left/right lists visited result in such *underfull reported blocks* and this I/O cost is $O(\log_2 \frac{N}{B})$, because we visit one left or right list per small node and the total number of small nodes visited is $O(\log_2 \frac{N}{B})$ (this is the height of the balanced binary interval tree T obtained by "concatenating" the small-node subtrees T_u's in all internal nodes u's of $\mathcal{T}$). Therefore the overall worst-case I/O cost is $O(\log_2 \frac{N}{B} + K/B)$.

We can improve the worst-case I/O query bound by using the *corner structure* [**16**] mentioned at the end of Section 2.1.1. The idea is to check a left/right list from disk *only when* at least *one full block* is reported; the underfull reported blocks are collectively taken care of by the corner structure.

For each internal node u of $\mathcal{T}$, we remove the first block from each left and right lists of each small node in u, and collect all these removed intervals (with duplications eliminated) into a single corner structure associated with u (if a left/right list has no more than B intervals then the list becomes empty). We also store in u a "guarding value" for each left/right list of u. For a left list, this guarding value is the smallest left endpoint value among the *remaining* intervals still kept in the left list (*i.e.*, the $(B+1)$-st smallest left endpoint value in the *original* left list); for a right list, this value is the largest right endpoint value among the remaining intervals kept (*i.e.*, the $(B+1)$-st largest right endpoint value in the *original* right list). Recall that each left list is sorted by increasing left endpoint values and symmetrically for each right list. Observe that u has $2(Bf-1)$ left and right lists and $Bf = \Theta(B)$, so the corner structure of u has $O(B^2)$ intervals, satisfying the restriction for the corner structure (see Section 2.1.1). Also, the overall space needed is still optimal $O(N/B)$ disk blocks.

The query algorithm is basically the same as before, with the following modification. If the current node u of $\mathcal{T}$ is an internal node, then we first query the corner structure of u. A left list of u is checked from disk only when the query value q is larger than or equal to the guarding value of that list; similarly for the right list. In this way, although a left/right list might be checked using one I/O to report very few ($<< B$) intervals, it is ensured that in this case the *original first block* of that list is also reported, from the corner structure of u. Therefore we can charge this one underflow I/O cost to the one I/O cost needed to report such first full block. This means that the overall underflow I/O cost can be charged to the K/B term of the reporting cost, so that the overall query I/O cost is optimal $O(\log_B N + K/B)$.

2.3.3. *Dynamization.* As a detour from our application of isosurface extraction which only needs the static version of the tree, we now discuss the ideas on how to apply the dynamization techniques of [**2**] to the binary-blocked I/O interval tree $\mathcal{T}$ so that it also supports insertions and deletions of intervals each in optimal $O(\log_B N)$ I/O's amortized (we believe that the bounds can further be turned into worst-case, as discussed in [**2**], but we did not verify the details).

The first step is to assume that all intervals have *left* endpoints in a fixed set E of N points. (This assumption will be removed later.) Each left/right list is now stored in a B-tree so that each insertion/deletion of an interval on a secondary list can be done in $O(\log_B N)$ I/O's. We slightly modify the way we use the guarding values and the corner structure of an internal node u of $\mathcal{T}$ mentioned at the end of Section 2.3.2: instead of putting the first B intervals of each left/right list into the corner structure, we put between $B/4$ and B intervals. For each left/right list L, we keep track of how many of its intervals are actually stored in the corner

structure; we denote this number by $C(L)$. When an interval I is to be inserted to a node u of $\mathcal{T}$, we insert I to its destination left and right lists in u, by checking the corresponding guarding values to actually insert I to the corner structure or to the list(s). Some care must be taken to make sure that no interval is inserted twice to the corner structure. Notice that each insertion/deletion on the corner structure needs amortized $O(1)$ I/O's [**16**]. When a left/right list L has B intervals stored in the corner structure (i.e., $C(L) = B$), we update the corner structure, the list L and its guarding value so that only the first $B/2$ intervals of L are actually placed in the corner structure. The update to the corner structure is performed by rebuilding it, in $O(B)$ I/O's; the update to the list L is done by inserting the extra $B/2$ intervals (coming from the corner structure) to L, in $O(B \log_B N)$ I/O's. We perform deletions in a similar way, where an adjustment of the guarding value of L occurs when L has only $B/4$ intervals in the corner structure (i.e., $C(L) = B/4$), in which case we delete the first $B/4$ intervals from L and insert them to the corner structure to make $C(L) = B/2$ ($C(L) < B/2$ if L stores less than $B/4$ intervals before the adjustment), using $O(B \log_B N)$ I/O's. Observe that between two $O(B \log_B N)$-cost updates there must be already $\Omega(B)$ updates, so each update needs amortized $O(\log_B N)$ I/O's.

Now we show how to remove the assumption that all intervals have *left* endpoints in a fixed set E, by using the *weight-balanced B-tree* developed in [**2**]. This is basically the same as the dynamization step of the I/O interval tree [**2**]; only the details for the rebalancing operations differ.

A weight-balanced B-tree has a *branching parameter* a and a *leaf parameter* k, such that all leaves have the same depth and have weight $\Theta(k)$, and that each internal node on level l (leaves are on level 0) has weight $\Theta(a^l k)$, where the weight of a leaf is the number of items stored in the leaf and the weight of an internal node u is the sum of the weights of the leaves in the subtree rooted at u (items defining the weights are stored only in the leaves). For our application on the binary-blocked I/O interval tree, we choose the maximum number of children of an internal node, $4a$, to be Bf ($= \Theta(B)$), and the maximum number of items stored in a leaf, $2k$, to be B. The leaves collectively store all N *left* endpoints of the intervals to define the weight of each node, and each node is one disk block. With these parameter values, the tree uses $O(N/B)$ disk blocks, and supports searches and insertions each in $O(\log_B N)$ worst-case I/O's [**2**]. As used in [**2**], this weight-balanced B-tree serves as the base tree of our binary-blocked I/O interval tree. Rebalancing of the base tree during an insertion is carried out by *splitting* nodes, where each split takes $O(1)$ I/O's and there are $O(\log_B N)$ splits (on the nodes along a leaf-to-root path). A key property is that after a node of weight w splits, it will split again only after another $\Omega(w)$ insertions [**2**]. Therefore, our goal is to split a node of weight w, including updating the secondary lists involved, in $O(w)$ I/O's, so that each split uses amortized $O(1)$ I/O's and thus the overall insertion cost is $O(\log_B N)$ amortized I/O's.

Suppose we want to split a node u on level l. Its weight $w = \Theta(a^l k) = \Theta(B^{l+1})$. As considered in [**2**], we split u into two new nodes u' and u'' along a slab boundary b of u, such that all children of u to the left of b belong to u' and the remaining children of u, which are to the right of b, belong to u''. The node u is replaced by u' and u'', and b becomes a *new* slab boundary in $parent(u)$, separating the two new adjacent slabs corresponding to u' and u''. To update the corresponding secondary lists, recall that the intervals belonging to u are distributed to the left/right lists of

the small nodes of u by a binary-search scheme, where the keys of the small nodes are the slab boundaries of u, and the small-node tree T_u of u is implicitly defined by the binary-search scheme. Here we slightly modify this, by just specifying the small-node tree T_u, which then guides the search scheme (so if T_u is not perfectly balanced, the search on the keys is not a usual binary search). Now, we have to re-distribute the intervals of u, so that those containing b are first put to $parent(u)$, and those to the left of b are put to u' and the rest put to u''. To do so, we first merge all left lists of u to get all intervals of u sorted by the left endpoints. Actually, we need one more list to participate in the merging, namely the list of those intervals stored in the corner structure of u, sorted by the left endpoints. This corner-structure list is easily produced by reading the intervals of the corner structure into main memory and performing an internal sorting. After the merging is done, we scan through the sorted list and re-distribute the intervals of u by using a *new* small-node tree T_u of u where b is the root key and the left and right subtrees of b are the small-node trees $T_{u'}$ and $T_{u''}$ inside the nodes u' and u''. This takes care of all left lists of u' and u'', each automatically sorted by the left endpoints, and also decides the set S_b of intervals that have to be moved to $parent(u)$. We construct the right lists in u' and u'' in a similar way. We build the corner structures of u' and u'' appropriately, by putting the first $B/2$ intervals of each left/right list (or the entire list if its size is less than $B/2$) to the related corner structure. Since each interval belonging to u has its left endpoint inside the slab associated with u, the total number of such intervals is $O(w)$, and thus we perform $O(w/B)$ I/O's in this step. Note that during the merging, we may have to spend one I/O to read an underfull left/right list (i.e., a list storing $<< B$ intervals), resulting in a total of $O(B)$ I/O's for the underfull lists. But for an internal node u, its weight $w = \Theta(B^{l+1})$ with $l \geq 1$, so the $O(B)$ term is dominated by the $O(w/B)$ term. (If u is a leaf then there are no left/right lists.) This completes the operations on the nodes u' and u''.

We also need to update $parent(u)$. First, the intervals in S_b have to be moved to $parent(u)$. Also, b becomes a new slab boundary in $parent(u)$, and we have to re-distribute the intervals of $parent(u)$, including those in S_b, against a *new* small-node tree $T_{parent(u)}$ in which b is the key of some small node. We again consider the left lists and the right lists separately as before. Notice that we get two lists for S_b from the previous step, sorted respectively by the left and the right endpoints. Since $parent(u)$ has weight $\Theta(a^{l+1}k) = \Theta(Bw)$, updating $parent(u)$ takes $O(w)$ I/O's. The overall update cost for splitting u is thus $O(w)$ I/O's, as desired.

We remark that in the case of the I/O interval tree [**2**], updating $parent(u)$ only needs $O(w/B)$ I/O's, which is better than needed. Alternatively, we can simplify our task of updating $parent(u)$ by always putting the new slab boundary b as a *leaf* of the small-node tree $T_{parent(u)}$. In this way, the original intervals of $parent(u)$ do not need to be re-distributed, and we only need to attach the left and right lists of S_b into $parent(u)$. Again, we need to insert the first $B/2$ intervals of the two lists (removing duplications) to the corner structure of $parent(u)$, using $O(B)$ I/O's by simply rebuilding the corner structure. For u being a leaf, the overall I/O cost for splitting u is $O(1+B) = O(w)$ (recall that $w = \Theta(B^{l+1})$), and for u being an internal node, such I/O cost is $O(w/B+B) = O(w/B)$. We remark that although the small-node trees (e.g., $T_{parent(u)}$) may become very unbalanced, our optimal query I/O bound and all other performance bounds are not affected, since they do not depend on the small-node trees being balanced. Finally, the deletions are performed by

lazy deletions, with the rebalancing carried out by a global rebuilding, as described in [**2**]. The same amortized bound carries over. In summary, our binary-blocked I/O interval tree can support insertions and deletions of intervals each in optimal $O(\log_B N)$ I/O's amortized, with all the other performance bounds unchanged.

2.3.4. *Preprocessing Algorithm.* Now we return to the static version of the tree $\mathcal{T}$. In [**10**] we again use the *scan and distribute* preprocessing algorithm to build the tree. The algorithm follows the definition of $\mathcal{T}$ given in Section 2.3.1.

In the first phase, we sort (using external sorting) all N input intervals in increasing *left* endpoint values from left to right, into a set S. We use interval ID's to break a tie. We also copy the *left* endpoints, in the same sorted order, from S to a set E. The set E is used to define the median values to partition E into slabs throughout the process.

The second phase is a recursive process. If there are no more than B intervals, then we make the current node u a leaf, store all intervals in u and stop. Otherwise, u is an internal node. We first take the $Bf-1$ median values from E that partition E into Bf slabs, each containing the same number of endpoints. We store sorted in u, in non-decreasing order from left to right, these median values as the keys in the small nodes of u. We now scan all intervals (from S) to distribute them to the node u or to one of the Bf slabs. We maintain a temporary list for u, and also a temporary list for each of the Bf slabs. For each temporary list, we keep one block in main memory as a *buffer*, and keep the remaining blocks in disk. Each time an interval is distribute to the node u or to a slab, we put that interval to the corresponding buffer; when a buffer is full, it is written to the corresponding list in disk. The distribution of each interval I is carried out by the *binary-search scheme* described in Section 2.3.1, which implicitly defines a balanced binary tree T_u on the $Bf-1$ keys and the corresponding small nodes in u. We perform this binary search on these keys to find the highest small node r whose key is contained in I, in which case we assign I to small node r (and also to the current node u), by appending the small node ID of r to I and putting it to the temporary list for the node u, or to find that no such small node exists and both endpoints of I lie in the same slab, in which case we distribute I to that slab by putting I to the corresponding temporary list. When all intervals in S are scanned and distributed, each temporary list has all its intervals, automatically sorted in increasing left endpoint values. Now we sort the intervals belonging to the node u by the small node ID as the first key and the left endpoint value as the second key, in increasing order, so that intervals assigned to the same small node are put together, sorted in increasing left endpoint values. We read off these intervals to set up the left lists of all small nodes in u. Then we copy each such left list to its corresponding right list, and sort the right list by decreasing right endpoint values. The corner structure of u, if we want to construct, can be built at this point. This completes the construction of u. Finally, we perform the process recursively on each of the Bf slabs, using the intervals in the corresponding temporary list as input, to build each subtree of the node u.

We remark that in the above *scan and distribute* process, instead of keeping all intervals assigned to the current node u in *one* temporary list, we could maintain $Bf-1$ temporary lists for the $Bf-1$ small nodes of u. This would eliminate the subsequent sorting by the small node ID's, which is used to *re-distribute* the intervals of u into individual small nodes. But as we shall see in Section 2.4, our method is used to address the system issue that a process cannot open too many files simultaneously, while avoiding a blow-up in disk scratch space.

It is easy to see that the entire process uses $O(\frac{N}{B}\log_B N)$ I/O's, which is nearly optimal. To make a theoretical improvement, we can view the above algorithm as processing $\Theta(\log_2 B)$ levels of the binary interval tree T at a time. By processing $\Theta(\log_2 \frac{M}{B})$ levels at a time, we achieve the theoretically optimal I/O bound $O(\frac{N}{B}\log_{\frac{M}{B}} \frac{N}{B})$. This is similar to the tree-height conversion method of [**1**] that turns the nearly-optimal I/O bound to optimal for the *scan and distribute* algorithm that builds the I/O interval tree of [**2**].

2.4. Implementing the Scan and Distribute Preprocessing Algorithms. There is an interesting issue associated with the implementation of the *scan and distribute* preprocessing algorithms. We use the binary-blocked I/O interval tree as an illustrating example. Recall from Section 2.3.4 that during one pass of *scan and distribute*, there are Bf temporary lists for the Bf child slabs and one additional temporary list for the current node u; all these lists grow simultaneously while we distribute intervals to them. If we use one file for each temporary list, then all these $Bf+1$ files (where $Bf = 170$ in our implementation) have to be open at the same time. Unfortunately, there is a hard limit, imposed by the OS, on the number of files a process can open at the same time. This number is given by the system parameter `OPEN_MAX`, which in older versions of UNIX was 20 and in many systems was increased to 64. Certainly we can not simultaneously open a file for each temporary list.

We solve this problem by using a single scratch file `dataset.intvl_temp` to collect all child-slab temporary lists, and a file `dataset.current` for the temporary list of the current node u. We use a file `dataset.intvl` for the set S of input intervals. Recall that we have a set E of the *left* endpoints of all intervals; this set is used to define the $Bf-1$ median values that partition E into Bf slabs of $\lceil n/Bf \rceil$ blocks each, where n is the size of E in terms of integral blocks. We use an interval to represent its left endpoint, so that E is of the same size as S. Since an interval belongs to a slab if both endpoints lie in that slab, the size of each child-slab temporary list is no more than $\lceil n/Bf \rceil$ blocks. Therefore we let the i-th such list start from the block $i \cdot \lceil n/Bf \rceil$ in the file `dataset.intvl_temp`, for $i = 0, \cdots, Bf-1$. After the construction of all such lists is over, we copy them to the corresponding positions in the file `dataset.intvl`, and the scratch file `dataset.intvl_temp` is available for use again. Note that this scratch file is of size n blocks. Recall from Section 2.3.4 that the temporary list for the current node u is used to keep the intervals assigned to u. Thus the size of the scratch file `dataset.current` is also no more than n blocks. After constructing the left and right lists of u, this scratch file is again available for use. To recursively perform the process for each child slab i, we use the portion of the file `dataset.intvl` starting from the block $i \cdot \lceil n/Bf \rceil$ with no more than $\lceil n/Bf \rceil$ blocks as the new set S of input intervals.

As mentioned in Section 2.3.4, our algorithm of collecting all intervals assigned to the current node u in a *single* temporary list is crucial for keeping the disk scratch space small. If we were to maintain $Bf-1$ temporary lists for the individual small nodes in u and collect all such lists into a scratch file as we do above for child slabs, then we would have a disk space blow-up. Observe that the *potential* size of each small-node list can be much larger than the actual size; in fact, *all* input intervals can belong to the topmost small node, or to the two small nodes one level below, or to any of the $\lceil \log_2 Bf \rceil$ levels in the subtree on the small nodes of u. Therefore to reserve enough space for each small-node list, the disk scratch space would blow up

by a factor of $\lceil \log_2 Bf \rceil$. In addition, the varying sizes of the temporary lists inside the scratch file would complicate the coding. (Using a uniform list size instead in this method, then, would increase the blow-up factor to $Bf - 1$!) On the contrary, our method is both simple and efficient in solving the problem.

3. The I/O-Filter Technique

In this section we describe our I/O-filter technique [**8, 9**]. As mentioned in Section 1.3, we first produce an interval $I = [\min, \max]$ for each cell of the dataset so that searching active cells amounts to performing stabbing queries. We then use the I/O interval tree or the metablock tree as the indexing structure to solve the stabbing queries, together with the isosurface engine of Vtk [**25**].

It is well known that random accesses in disk following pointer references are very inefficient. If we keep the dataset and build a *separate* indexing structure where each interval has a pointer to its corresponding cell record in disk, then during queries we have to perform pointer references in disk to obtain the cell information, possibly reading one disk block per active cell, *i.e.*, the reporting I/O cost becomes $O(K)$ rather than $O(K/B)$, which is certainly undesirable. In the I/O-filter technique [**8, 9**], we store the cell information *together with* its interval in the indexing data structure, so that this kind of pointer references are avoided. Also, the cell information we store is the *direct cell information*, *i.e.*, the x-, y-, z- and the scalar values of the vertices of the cell, rather than pointers to the vertices in the vertex information list. (In addition to the direct cell information, we also store the cell ID and the left and right endpoint values for each interval.) In this way, the dataset is *combined* with the indexing structure, and the original dataset can be thrown away. Inefficient pointer references in disk are completely avoided, at the cost of increasing the disk space needed to hold the combined indexing structure. We will address this issue in Section 4 when we describe our two-level indexing scheme [**10**].

3.1. Normalization. If the input dataset is given in a format that provides direct cell information, then we can build the interval/metablock tree directly. Unfortunately, the datasets are often given in a format that contains indices to vertices[1]. In the I/O filter technique, we first de-reference the indices before actually building the interval tree or the metablock tree. We call this de-referencing process *normalization*.

Using the technique of [**5, 7**], we can efficiently perform normalization as follows. We make one file (the *vertex file*) containing the direct information of the vertices (the 3D coordinates and the scalar values), and another file (the *cell file*) of the cell records with the vertex indices. In the first pass, we externally sort the cell file by the indices (pointers) to the first vertex, so that the first group in the file contains the cells whose first vertices are vertex 1, the second group contains the cells whose first vertices are vertex 2, and so on. Then by scanning through the vertex file and the cell file simultaneously, we fill in the direct information of the first vertex of each cell in the cell file. In the next pass, we sort the cell file by the indices to the second vertices, and fill in the direct information of the second vertex of each cell in the same way. Actually, each pass is a *joint* operation

[1]The input is usually a Toff file, which is analogous to the Geomview "off" file. It has the number of vertices and tetrahedra, followed by a list of the vertices and a list of the tetrahedra, each of which is specified using the vertex location in the file as an index. See [**29**].

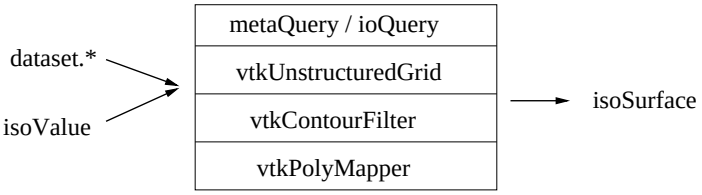

FIGURE 6. Isosurface extraction phase. Given the data structure files of the metablock/interval tree and an isovalue, `metaQuery`/`ioQuery` filters the dataset and passes to Vtk only those active cells of the isosurface. Several Vtk routines are used to generate the isosurface.

(commonly used in database), using the vertex ID's (the vertex indices) as the key on both the cell file and the vertex file. By repeating the joint process for each vertex of the cells, we obtain the direct information for each cell; this completes the normalization process.

3.2. Interfacing with Vtk. A full isosurface extraction pipeline should include several steps in addition to finding active cells (see Fig. 2). In particular, (1) the intersection points and triangles have to be computed; (2) the triangles need to be decimated [**26**]; and (3) the triangle strips have to be generated. Steps (1)–(3) can be carried out by the existing code in Vtk [**25**].

Our two pieces of isosurface querying code, `metaQuery` (for querying the metablock tree) and `ioQuery` (for querying the I/O interval tree), are implemented by linking the respective I/O querying code with Vtk's isosurface generation code, as shown in Fig. 6. Given an isovalue q, we use `metaQuery` or `ioQuery` to query the indexing structure in disk, and bring *only the active cells* to main memory; this much smaller set of active cells is treated as an input to Vtk, whose usual routines are then used to generate the isosurface. Thus we *filter out* those portions of the dataset that are not needed by Vtk. More specifically, given an isovalue, (1) all active cells are collected from disk; (2) a vtkUnstructuredGrid object is generated; (3) the isosurface is extracted with vtkContourFilter; and (4) the isosurface is saved in a file with vtkPolyMapper. At this point, memory is deallocated. If multiple isosurfaces are needed, this process is repeated. Note that this approach requires double buffering of the active cells during the creation of the vtkUnstructuredGrid data structure. A more sophisticated implementation would be to incorporate the functionality of `metaQuery` (resp. `ioQuery`) inside the Vtk data structures and make the methods I/O aware. This should be possible due to Vtk's pipeline evaluation scheme (see Chapter 4 of [**25**]).

3.3. Experimental Results. Now we present some experimental results of running the two implementations of *I/O-filter* and also Vtk's native isosurface implementation on real datasets. We have run our experiments on four different datasets. All of these are tetrahedralized versions of well-known datasets. The Blunt Fin, the Liquid Oxygen Post, and the Delta Wing datasets are courtesy of NASA. The Combustion Chamber dataset is from Vtk [**25**]. Some representative isosurfaces generated from our experiments are shown in Fig. 1.

Our benchmark machine was an off-the-shelf PC: a Pentium Pro, 200MHz with 128M of RAM, and two EIDE Western Digital 2.5Gb hard disk (5400 RPM, 128Kb

	Blunt	Chamber	Post	Delta
`metaQuery` – 128M	9s	17s	19s	26s
`ioQuery` – 128M	7s	16s	18s	31s
`vtkiso` – 128M	15s	22s	44s	182s
`vtkiso` I/O – 128M	3s	2s	12s	40s
`metaQuery` – 32M	9s	19s	21s	31s
`ioQuery` – 32M	10s	19s	22s	32s
`vtkiso` – 32M	21s	54s	1563s	3188s
`vtkiso` I/O – 32M	8s	28s	123s	249s

TABLE 1. Overall running times for the extraction of the 10 isosurfaces using `metaQuery`, `ioQuery`, and `vtkiso` with different amounts of main memory. These include the time to read the datasets and write the isosurfaces to files. `vtkiso` I/O is the fractional amount of time of `vtkiso` for reading the dataset and generating a vtkUnstructuredGrid object.

cache, 12ms seek time). Each disk block size is 4,096 bytes. We ran Linux (kernels 2.0.27, and 2.0.30) on this machine. One interesting property of Linux is that it allows during booting the specification of the exact amount of main memory to use. This allows us to *fake* for the isosurface code a given amount of main memory to use (after this memory is completely used, the system will start to use disk swap space and have page faults). This has the exact same effect as removing physical main memory from the machine.

In the following we use `metaQuery` and `ioQuery` to denote the entire isosurface extraction codes shown in Fig. 6, and `vtkiso` to denote the Vtk-only isosurface code. There are two batteries of tests, each with different amount of main memory (128M and 32M). Each test consists of calculating 10 isosurfaces with isovalues in the range of the scalar values of each dataset, by using `metaQuery`, `ioQuery`, and `vtkiso`. We did not run X-windows during the isosurface extraction time, and the output of vtkPolyMapper was saved in a file instead.

We summarize in Table 1 the *total* running times for the extraction of the 10 isosurfaces using `metaQuery`, `ioQuery`, and `vtkiso` with different amounts of main memory. Observe that both `metaQuery` and `ioQuery` have significant advantages over `vtkiso`, especially for large datasets and/or small main memory. In particular, from the Delta entries in 32M, we see that both `metaQuery` and `ioQuery` are about 100 times faster than `vtkiso`!

In Fig. 7, we show representative benchmarks of calculating 10 isosurfaces from the Delta dataset with 32M of main memory, using `ioQuery` (left column) and `vtkiso` (right column). For each isosurface calculated using `ioQuery`, we break the running time into four categories. In particular, the bottommost part is the time to perform I/O search in the I/O interval tree to find the active cells (the search phase), and the third part from the bottom is the time for Vtk to actually generate the isosurface from the active cells (the generation phase). It can be seen that the search phase always takes *less* time than the generation phase, *i.e.*, the search phase is no longer a bottleneck. Moreover, the cost of the search phase can be hidden by

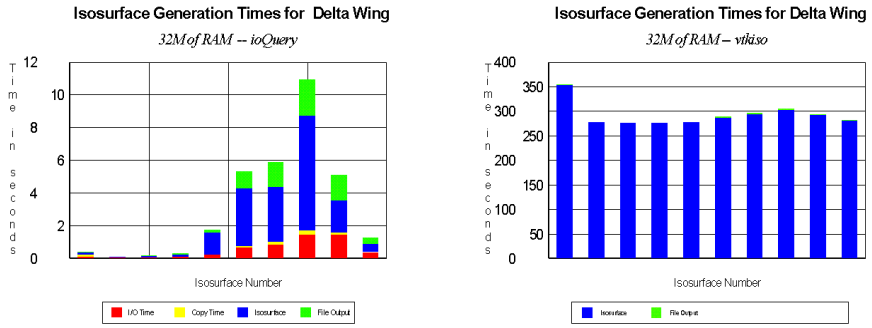

FIGURE 7. Running times for extracting isosurfaces from the Delta dataset, using `ioQuery` (left column) and `vtkiso` (right column) with 32M of main memory. Note that for `vtkiso` the cost of reading the entire dataset into main memory is not shown.

overlapping this I/O time with the CPU (generation phase) time. The isosurface query performance of `metaQuery` is similar to that of `ioQuery`.

In Table 2, we show the performance of querying the metablock tree and the I/O interval tree on the Delta dataset. In general, the query times of the two trees are comparable. We are surprised to see that sometimes the metablock tree performs much more disk reads but the running time is faster. This can be explained by a better locality of disk accesses of the metablock tree. In the metablock tree, the *horizontal*, *vertical*, and TS lists are always read sequentially during queries, but in the I/O interval tree, although the *left* and *right* lists are always read sequentially, the *multi* lists reported may cause scattered disk accesses. Recall from Section 2.2.2 that for a query value q lying in the slab i, all *multi* lists of the multi-slabs spanning the slab i are reported; these include the multi-slabs $[1,i],[1,i+1],\cdots,[1,Bf-2]$, $[2,i],[2,i+1],\cdots,[2,Bf-2],\cdots,[i,i],[i,i+1],\cdots,[i,Bf-2]$. While $[\ell,\cdot]$'s are in consecutive places of a file and can be sequentially accessed, changing from $[\ell,Bf-2]$ to $[\ell+1,i]$ causes non-sequential disk reads (since $[\ell+1,\ell+1],\cdots,[\ell+1,i-1]$ are skipped) — we store the *multi* lists in a "row-wise" manner in our implementation, but a "column-wise" implementation would also encounter a similar situation. This

Delta (1,005K cells)											
Isosurface ID		1	2	3	4	5	6	7	8	9	10
Active Cells		32	296	1150	1932	5238	24788	36738	55205	32677	8902
`metaQuery`	Page Ins	3	8	506	503	471	617	705	1270	1088	440
	Time (sec)	0.05	0	0.31	0.02	0.02	0.89	0.59	1.24	1.88	0.29
`ioQuery`	Page Ins	6	31	35	46	158	578	888	1171	765	271
	Time (sec)	0.1	0.05	0.05	0.09	0.2	0.67	0.85	1.44	1.43	0.35

TABLE 2. Searching active cells on the metablock tree (using `metaQuery`) and on the I/O interval tree (using `ioQuery`) in a machine with 32M of main memory. This shows the performance of the query operations of the two trees. (A "0" entry means "less than 0.01 before rounding".)

leads us to believe that in order to correctly model I/O algorithms, some cost should be associated with the *disk-head movements*, since this is one of the major costs involved in disk accesses.

Finally, without showing the detailed tables, we remark that our metablock-tree implementation of I/O-filter improves the practical performance of our I/O-interval-tree implementation of I/O-filter as follows: the tree construction time is twice as fast, the average disk space is reduced from 7.7 times the original dataset size to 7.2 times, and the disk scratch space needed during preprocessing is reduced from 16 times the original dataset size to 10 times. In each implementation of I/O-filter, the running time of preprocessing is the same as running external sorting a few times, and is linear in the dataset size even when it exceeds the main memory size, showing the *scalability* of the preprocessing algorithms.

We should point out that our I/O interval tree implementation did not implement the corner structure for the sake of simplicity of the coding. This may result in non-optimal disk space and non-optimal I/O query cost in the worst case. Our metablock tree implementation did not implement its corner structure either; this is to reduce the disk space overhead (unlike the I/O interval tree, the corner structure is not needed to achieve the worst-case optimal space bound; implementing the corner structure in the metablock tree would only increase the space overhead by some constant factor), but may also result in non-optimal query I/O cost in the worst case. Since the I/O query time already took less time than the CPU generation time, our main interest for implementing the metablock tree was in reducing the disk space overhead and hence there was no need to implement the corner structure. However, from the view point of data-structure experimentation, it is an interesting open question to investigate the effects of the corner structure into the practical performance measures of the two trees.

4. The Two-Level Indexing Scheme

In this section we survey our two-level indexing scheme proposed in [**10**]. The goal is to reduce the large disk space overhead of I/O-filter, at the cost of possibly increasing the query time. In particular, we would like to have a flexible scheme that can exhibit a smooth trade-off between disk space and query time.

We observe that in the I/O-filter technique [**8, 9**], to avoid inefficient *pointer references* in disk, the *direct cell information* is stored with its interval, in the indexing data structure (see Section 3). This is very inefficient in disk space, since the vertex information (*i.e.*, the x-, y-, $z-$ and the scalar values of the vertex) is duplicated many times, once for each cell sharing the vertex. Moreover, in the indexing structures [**2, 16**] used, each interval is stored three times in practice, increasing the duplications of vertex information by another factor of three. To eliminate this inefficiency, the new indexing scheme uses a two-level structure. First, we partition the original dataset into clusters of cells, called *meta-cells*. Secondly, we produce *meta-intervals* associated with the meta-cells, and build our indexing data structure on the meta-intervals. We *separate* the cell information, kept only in meta-cells in disk, from the indexing structure, which is also in disk and only contains pointers to meta-cells. Isosurface queries are performed by first querying the indexing structure, then using the reported meta-cell pointers to read from disk the *active* meta-cells intersected by the isosurface, which can then be generated from active meta-cells.

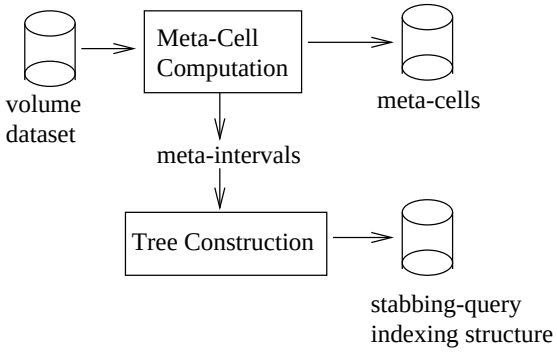

FIGURE 8. The preprocessing pipeline of the two-level indexing scheme isosurface technique.

While we need to perform *pointer references* in disk from the indexing structure to meta-cells, the *spatial coherences* of isosurfaces and of our meta-cells ensure that each meta-cell being read contains *many* active cells, so such pointer references are efficient. Also, a meta-cell is always read as a whole, hence we can use pointers *within* a meta-cell to store each meta-cell compactly. In this way, we obtain efficiencies in *both* query time and disk space.

4.1. Main Techniques. We show the preprocessing pipeline of the overall algorithm in Fig. 8. The main tasks are as follows.

1. Group spatially neighboring cells into *meta-cells*. The total number of vertices in each meta-cell is roughly the same, so that during queries each meta-cell can be retrieved from disk with approximately the same I/O cost. Each cell is assigned to exactly one meta-cell.
2. Compute and store in disk the meta-cell information for each meta-cell.
3. Compute *meta-intervals* associated with each meta-cell. Each meta-interval is an interval $[\min, \max]$, to be defined later.
4. Build in disk a stabbing-query indexing structure on meta-intervals. For each meta-interval, only its min and max values and the meta-cell ID are stored in the indexing structure, where the meta-cell ID is a pointer to the corresponding meta-cell record in disk.

We describe the representation of meta-cells. Each meta-cell has a list of vertices, where each vertex entry contains its x-, y-, z- and scalar values, and a list of cells, where each cell entry contains pointers to its vertices in the vertex list. In this way, a vertex shared by many cells in the same meta-cell is stored just *once* in that meta-cell. The only duplications of vertex information occur when a vertex belongs to two cells in *different* meta-cells; in this case we let both meta-cells include that vertex in their vertex lists, so that each meta-cell has *self-contained* vertex and cell lists. We store the meta-cells, one after another, in disk.

The purpose of the meta-intervals for a meta-cell is analogous to that of an interval for a cell, *i.e.*, a meta-cell is *active* (intersected by the isosurface of q) if and only if one of its meta-intervals contains q. Intuitively, we could just take the minimum and maximum scalar values among the vertices to define the meta-interval (as cell intervals), but such big range would contain "gaps" in which no

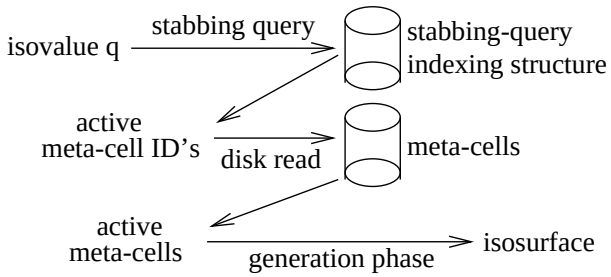

FIGURE 9. The query pipeline of the two-level indexing scheme isosurface technique.

cell interval lies. Therefore, we break such big range into pieces, each a meta-interval, by the gaps. Formally, we define the *meta-intervals* of a meta-cell as the *connected components* among the intervals of the cells in that meta-cell. With this definition, searching active meta-cells amounts to performing stabbing queries on the meta-intervals.

The query pipeline of our overall algorithm is shown in Fig. 9. We have the following steps.

1. Find all meta-intervals (and the corresponding meta-cell ID's) containing q, by querying the stabbing-query indexing structure in disk.
2. (Internally) sort the reported meta-cell ID's. This makes the subsequent disk reads for active meta-cells *sequential* (except for skipping inactive meta-cells), and minimizes the disk-head movements.
3. For each active meta-cell, read it from disk to main memory, identify active cells and compute isosurface triangles, throw away the current meta-cell from main memory and repeat the process for the next active meta-cell. At the end, patch the generated triangles and perform the remaining operations in the generation phase to generate and display the isosurface.

Now we argue that in step 3 the pointer references in disk to read meta-cells are efficient. Intuitively, by the way we construct the meta-cells, we can think of each meta-cell as a cube, with roughly the same number of cells in each dimension. Also, by the *spatial coherence* of an isosurface, the isosurface usually cannot cut too many meta-cells *through corners only*. Thus by a dimension argument, if an active meta-cell has C cells, for most times the isosurface cuts through $C^{2/3}$ cells. This is similar to the argument that usually there are $\Theta(N^{2/3})$ active cells in an N-cell volume dataset. This means that we read C cells (a whole meta-cell) for every $C^{2/3}$ active cells, *i.e.*, we traverse a "thickness" of $C^{1/3}$ layers of cells, for one layer of isosurface. Therefore we read $C^{1/3} \cdot (K/B)$ disk blocks for K active cells, which is a factor of $C^{1/3}$ from optimal. Notice that when the size of the meta-cells is increased, the number of duplicated vertices is decreased (less vertices in meta-cell boundaries), and the number of meta-intervals is also decreased (less meta-cells), while the number C is increased. Hence we have a *trade-off* between space and query time, by varying the meta-cell size. Since the major cost in disk reads is in *disk-head movements* (*e.g.*, reading two disk blocks takes approximately the same time as reading one block, after moving the disk head), we can increase meta-cell sizes while keeping the effect of the factor $C^{1/3}$ negligible.

4.1.1. *Stabbing Query Indexing Structure.* In the two-level indexing scheme, any stabbing-query indexing structure can be used. For the sake of simplicity of coding and being more space-efficient, we implemented the binary-blocked I/O interval tree [**10**] (see Section 2.3) as the indexing structure. Since our goal is to reduce the disk space overhead, we do not implement the corner structure (whose only purpose is to improve the query I/O cost to optimal; see Section 2.3.2).

4.1.2. *Meta-Cell Computation.* The efficient subdivision of the dataset into meta-cells lies at the heart of the overall isosurface algorithm. The computation is similar to the partition induced by a k-d-tree [**4**], but we do not need to compute the multiple levels. The computation is essentially carried out by performing external sorting a few times.

We assume that the input dataset is in a general "index cell set" (ICS) format, *i.e.*, there is a list of vertices, each containing its x-, y-, z- and scalar values, and a list of cells, each containing pointers to its vertices in the vertex list. We want to partition the dataset into H^3 meta-cells, where H is a parameter that we can adjust to vary the meta-cell sizes, which are usually several disk blocks. The final output of the meta-cell computation is a single file that contains all meta-cells, one after another, each an *independent* ICS file (*i.e.*, the pointer references from the cells of a meta-cell are *within* the meta-cell). We also produce meta-intervals for each meta-cell. The meta-cell computation consists of the following steps.

1. Partition vertices into clusters of equal size. This is the *key* step in constructing meta-cells. We use each resulting cluster to define a meta-cell, whose vertices are those in the cluster, plus some *duplicated* vertices to be constructed later. Observe that meta-cells may differ dramatically in their volumes, but their numbers of vertices are roughly the same. The partitioning method is very simple. We first externally sort all vertices by the x-values, and partition them into H consecutive chunks. Then, for each such chunk, we externally sort its vertices by the y-values, and partition them into H chunks. Finally, we repeat the process for each refined chunk, except that we externally sort the vertices by the z-values. We take the final chunks as clusters. Clearly, each cluster has spatially neighboring vertices. The computing cost is bounded by three passes of external sorting. This step actually *assigns* vertices to meta-cells. We produce a *vertex-assignment* list with entries (v_{id}, m_{id}), indicating that the vertex v_{id} is assigned to the meta-cell m_{id}.

2. Assign cells to meta-cells and duplicate vertices. Our assignment of cells to meta-cells attempts to minimize the wasted space. The basic coverage criterion is to see how a cell's vertices have been mapped to meta-cells. A cell whose vertices all belong to the same meta-cell is assigned to that meta-cell. Otherwise, the cell is in the boundary, and a simple voting scheme is used: the meta-cell that contains the "most" vertices owns that cell, and the "missing" vertices of the cell have to be duplicated and inserted to this meta-cell. We break ties arbitrarily. In order to determine this assignment, we need to obtain for each cell, the destination meta-cells of its vertices. This pointer de-referencing in disk is carried out by performing the *joint* operation a few times as described in the *normalization* process in Section 3.1. At the end, we have a list for assigning cells to meta-cells, and also a *vertex-duplication* list with entries (v_{id}, m_{id}), indicating that the vertex v_{id} has to be duplicated and inserted to the meta-cell m_{id}.

3. Compute the vertex and cell lists for each meta-cell. To actually duplicate vertices and insert them to appropriate meta-cells, we first need to de-reference the

vertex ID's (to obtain the *complete* vertex information) from the vertex-duplication list. We can do this by using one join operation, using vertex ID as the key, on the original input vertex list and the vertex-duplication list. Now the vertex-duplication list contains for each entry the complete vertex information, together with the ID of the meta-cell to which the vertex must be inserted. We also have a list for assigning cells to meta-cells. To finish the generation of meta-cells, we use a main join operation on these lists, using the meta-cell ID's as the main key. To avoid possible replications of the same vertex inside a meta-cell, we use vertex ID's as the secondary key during the sorting for the join operation. Finally, we update the vertex pointers for the cells *within* each meta-cell. This can be easily done since each meta-cell can be kept in main memory.

4. Compute meta-intervals for each meta-cell. Since each meta-cell can fit in main memory, this step only involves in-core computation. First, we compute the interval for each cell in the meta-cell. Then we sort all interval endpoints. We scan through the endpoints, with a counter initialized to 0. A left endpoint encountered increases the counter by 1, and a right endpoint decreases the counter by 1. A "$0 \rightarrow 1$" transition gives the beginning of a new meta-interval, and a "$1 \rightarrow 0$" transition gives the end of the current meta-interval. We can easily see that the computation is correct, and the computing time is bounded by that of internal sorting.

4.2. Experimental Results. The experimental set up is similar to the one described in Section 3.3. We considered five datasets in the experimental study; four of them were used in [**8, 9**] as shown in Section 3.3. A new, larger dataset, Cyl3 with about 5.8 million cells has been added to our test set. This new dataset was originally a smaller dataset (the Cylinder dataset) from Vtk [**25**] but was subdivided to higher resolution by breaking the tetrahedra into smaller ones.

Tables 3 and 4 summarize some important statistics about the performance of the preprocessing. In Table 3, a global view of the performance of the technique can be seen on four different datasets. Recall that the indexing structure used is the binary-blocked I/O interval tree [**10**] (see Section 2.3), abbreviated the BBIO tree here. It is interesting to note that the size of the BBIO tree is quite small. This is because we separate the indexing structure from the dataset. Also, previously we have one interval stored for each cell, but now we have only a few (usually no more than 3) meta-intervals for a meta-cell, which typically contains more than 200 cells. We remark that if we replace the I/O interval tree or the metablock tree with the BBIO tree in the I/O-filter technique (described in Section 3), then the average tree size (combining the direct cell information together) is about 5 times the original dataset size, as opposed to 7.2–7.7 times, showing that the BBIO tree *along* can improve the disk space by a factor of 2/3 (and much more improvements are obtained by employing the two-level indexing scheme and the meta-cell technique).

In Table 4 we vary the number of meta-cells used for the Delta dataset. This table shows that our algorithm scales well with increasing meta-cell sizes. The most important feature is the linear dependency of the querying accuracy versus the disk overhead. That is, using 146 meta-cells (at 7% disk space overhead), for a given isosurface, we needed 3.34s to find the active cells. When using 30,628 meta-cells (at 63% disk space overhead), we only need 1.18s to find the active cells. Basically, the more meta-cells, the more accurate our active cell searchers, and less data we need to fetch from disk. An interesting point is that the more data fetched, the

	Blunt	Chamber	Post	Cyl3
# of meta-cells	737	1009	1870	27896
Const. Time	50s	60s	154.8s	3652s
Original Size	3.65M	4.19M	10M	152M
Meta-Cell Size	4.39M	5M	12.2M	271M
Size Increase	20%	21%	22%	78%
Avg # of Cells	254.2	213.1	274.5	208
BBIO_Tree (size)	29K	28K	84K	1.7M
BBIO_Tree (time)	0.35s	0.67s	1.23s	43s

TABLE 3. Statistics for the preprocessing on different datasets. First, we show the number of meta-cells used for partitioning the dataset, followed by the total time for the meta-cell computation. Secondly, the original dataset size and the size of the meta-cell file are shown. We also show the overall increase in storage, and the average number of cells in a meta-cell. Next, we show the size (in bytes) of the indexing structure, the binary-blocked I/O interval tree (the BBIO tree) and its construction time.

# of meta-cells	146	361	1100	2364	3600	8400	30628
Size Increase	7%	10%	16%	22%	26.9%	37.9%	63%
Act. Meta-Cells	59%	52%	37%	31%	30%	23%	16%
Query Time	3.34s	2.76s	2.09s	1.82s	1.73s	1.5s	1.18s

TABLE 4. Statistics for preprocessing and querying isosurfaces on the Delta dataset (original size 19.4M). We show the size increase in the disk space overhead. Also, we show the performance of a representative isosurface query: percentage of the meta-cells that are active (intersected by the isosurface), and the time for finding the active cells (the time for actual isosurface generation is not included here).

more work (and more main memory usage due to larger individual meta-cell sizes) that the isosurface generation engine has to do. By paying 63% disk overhead, we only need to fetch 16% of the dataset into main memory, which is clearly a substantial saving.

In summary, we see that the disk space is reduced from 7.2–7.7 times the original dataset size in the I/O-filter technique to 1.1–1.5 times, and the disk scratch space is reduced from 10–16 times to less than 2 times. The query time is still at least one order of magnitude faster than Vtk. Also, rather than being a single-cost indexing approach, the method exhibits a smooth trade-off between disk space and query time.

5. Conclusions and Future Directions

In this paper we survey our recent work on external memory techniques for isosurface extraction in scientific visualization. Our techniques provide cost-effective

methods to speed up isosurface extraction from volume data. The actual code can be made much faster by fine tuning the disk I/O. This is an interesting but hard and time-consuming task, and might often be non-portable across platforms, since the interplay among the operating system, the algorithms, and the disk is non-trivial to optimize. We believe that a substantial speed-up can be achieved by optimizing the external sorting and the file copying primitives.

The two-level indexing scheme is also suitable for dealing with *time-varying* datasets, in which there are several scalar values at each sample point, one value for each time step. By separating the indexing data structure from the meta-cells, one can keep a single copy of the geometric data (in the meta-cells), and have multiple indexing structures for indexing different time steps.

The technique we use to compute the meta-cells has a wider applicability in the preprocessing of general cell structures larger than main memory. For example, one could use our technique to break polyhedral surfaces larger than main memory into spatially coherent sections for simplification, or to break large volumetric grids into smaller ones for rendering purposes.

We believe the two-level indexing scheme [**10**] brings efficient external-memory isosurface techniques closer to practicality. One remaining challenge is to improve the preprocessing times for large datasets, which, even though is much lower than the ones presented in [**8, 9**], is still fairly costly.

Acknowledgements

We thank William Schroeder, who is our co-author in the two-level indexing scheme paper [**10**] and also one of the authors of Vtk [**24, 25**]. We thank James Abello, Jeff Vitter and Lars Arge for useful comments.

References

[1] L. Arge. Personal communication, April, 1997.

[2] L. Arge and J. S. Vitter. Optimal interval management in external memory. In *Proc. IEEE Foundations of Comp. Sci.*, pages 560–569, 1996.

[3] C. L. Bajaj, V. Pascucci, and D. R. Schikore. Fast isocontouring for improved interactivity. In *1996 Volume Visualization Symposium*, pages 39–46, October 1996.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[5] Y.-J. Chiang. Dynamic and I/O-efficient algorithms for computational geometry and graph problems: theoretical and experimental results. Ph.D. Thesis, Technical Report CS-95-27, Dept. Computer Science, Brown University, 1995.

[6] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. *Computational Geometry: Theory and Applications*, 9(4):211–236, 1998.

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

[8] Y.-J. Chiang and C. T. Silva. I/O optimal isosurface extraction. In *Proc. IEEE Visualization*, pages 293–300, 1997.

[9] Y.-J. Chiang and C. T. Silva. Isosurface extraction in large scientific visualization applications using the I/O-filter technique. Submitted for a journal publication. Preprint: Technical Report, University at Stony Brook, 1997.

[10] Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. In *Proc. IEEE Visualization*, 1998 (to appear).

[11] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Optimal isosurface extraction from irregular volume data. In *1996 Volume Visualization Symposium*, pages 31–38, October 1996.

[12] H. Edelsbrunner. A new approach to rectangle intersections, Part I. *Internat. J. Comput. Math.*, 13:209–219, 1983.
[13] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *IEEE Foundations of Comp. Sci.*, pages 714–723, 1993.
[14] T. He, L. Hong, A. Varshney, and S. Wang. Controlled topology simplification. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):171–184, June 1996.
[15] T. Itoh and K. Koyamada. Automatic isosurface propagation using an extrema graph and sorted boundary cell lists. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):319–327, December 1995.
[16] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. In *Proc. ACM Symp. on Principles of Database Sys.*, pages 233–243, 1993.
[17] A. Kaufman. Volume visualization. In J. G. Webster, editor, *Encyclopedia of Electrical and Electronics Engineering*, pages 163–169. Wiley Publishing, 1997.
[18] A. Kaufman, D. Cohen, and R. Yagel. Volume graphics. *IEEE Computer*, 26:51–64, July 1993.
[19] Y. Livnat, H.-W. Shen, and C.R. Johnson. A near optimal isosurface extraction algorithm using span space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, March 1996.
[20] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 163–169, July 1987.
[21] E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14:257–276, 1985.
[22] G. M. Nielson and B. Hamann. The asymptotic decider: Removing the ambiguity in marching cubes. In *Visualization '91*, pages 83–91, 1991.
[23] W. Schroeder, W. Lorensen, and C. Linthicum. Implicit modeling of swept surfaces and volumes. In *IEEE Visualization '94*, pages 40–45, October 1994.
[24] W. Schroeder, K. Martin, and W. Lorensen. The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In *IEEE Visualization '96*, October 1996.
[25] W. Schroeder, K. Martin, and W. Lorensen. *The Visualization Toolkit*. Prentice-Hall, 1996.
[26] W. Schroeder, J. Zarge, and W. Lorensen. Decimation of triangle meshes. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 65–70, July 1992.
[27] H.-W. Shen, C. D. Hansen, Y. Livnat, and C. R. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In *IEEE Visualization '96*, October 1996.
[28] H.-W. Shen and C.R. Johnson. Sweeping simplices: A fast iso-surface extraction algorithm for unstructured grids. In *IEEE Visualization '95*, pages 143–150, October 1995.
[29] C. T. Silva, J. S. B. Mitchell, and A. E. Kaufman. Fast rendering of irregular grids. In *1996 Volume Visualization Symposium*, pages 15–22, October 1996.
[30] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, volume 24, pages 57–62, November 1990.

Department of Computer and Information Science, Polytechnic University, 6 MetroTech Center, Brooklyn, NY 11201
E-mail address: yjc@poly.edu

P.O. Box 704, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598
E-mail address: csilva@watson.ibm.com