A Generic Approach to Bulk Loading Multidimensional Index Structures

Jochen van den Bercken[1] **Bernhard Seeger**[1] **Peter Widmayer**[2]

[1]Fachgebiet Informatik, Universität Marburg
Hans-Meerwein-Str., D-35032 Marburg, Germany
e-mail: {*lastname*}@informatik.uni-marburg.de

[2]Institut für Theoretische Informatik
ETH-Zentrum, CH-8092 Zürich, Switzerland
e-mail: widmayer@inf.ethz.ch

Abstract: Recently there has been an increasing interest in supporting bulk operations on multidimensional index structures. Bulk loading refers to the process of creating an initial index structure for a presumably very large data set. In this paper, we present a generic algorithm for bulk loading which is applicable to a broad class of index structures. Our approach differs completely from previous ones for the following reasons. First, sorting multidimensional data according to a predefined global ordering is completely avoided. Instead, our approach is based on the standard routines for splitting and merging pages which are already fully implemented in the corresponding index structure. Second, in contrast to inserting records one by one, our approach is based on the idea of inserting multiple records simultaneously. As an example we demonstrate in this paper how to apply our technique to the R-tree family. For R-trees we show that the I/O performance of our generic algorithm meets the lower bound of external sorting. Empirical results demonstrate that performance improvements are also achieved in practice without sacrificing query performance.

1 Introduction

Among the most important non-standard databases are historical, spatial, image and text databases. For temporal and spatial databases, the underlying multidimensional objects are embedded typically in a two- or three-dimensional space. Objects are not limited to points, but may possess a spatial extent. For image and text databases, a multidimensional object represents an array of features taken from a picture and document, respectively. Typically, the dimension of the objects is extremely high. In text databases, for example, vectors have usually more than 1000 components [Hen 96]. Common to all these applications is that one-dimensional index structures (e.g., B+-trees) are not appropriate indexing techniques.

In order to support queries on multidimensional data sets efficiently, a huge number of multidimensional index structures have been proposed during the last two decades. Multidimensional index structures support insertions, deletions and updates, as well as proximity queries (e.g., window queries, nearest-neighbor queries). In addition to these traditional operations, there is currently an increasing interest in supporting bulk operations. A bulk operation is a collection of individual operations that are executed in consecution, without being interrupted by other requests. The most common bulk operation is to create a multidimensional index structure for a given set of records from scratch. This operation has also been termed bulk loading [DKL+ 94]. Bulk loading is for example important for processing spatial joins [LR 94] when (spatial) indices do not exist on the participating relations. In this paper, we address the problem of supporting bulk loading of multidimensional tree-based index structures.

So far, there has been very little work on bulk loading of multidimensional index structures. There have been several proposals for bulk loading R-trees [RL 85], [KF 93], [LEL 95]. All of them are based on sorting data according to a global one-dimensional criterion (e.g., attribute of the data object, space-filling curve). After sorting, they follow the standard approach of clustering the data according to the linear order (known from B+-trees), i.e., they simply build up the R-tree index from bottom to top. In [DKL 94] it has been recognized that this approach generally leads to poor query performance. The performance of R-trees produced by these bulk loading methods seems to be acceptable when both of the following assumptions are fulfilled: the dimension of the objects is low and the objects have small spatial extent (in parameter space). For the applications mentioned above, both assumptions generally do not hold, and also R-trees may not always be the prime choice for the problem at hand. For structures other than R-trees and those multidimensional index structures that are based on the combination of B+-trees and space-filling curves, the problem of bulk loading has not been addressed.

We propose an approach to bulk loading that is completely different from the previous ones used on R-trees. The reason is that sorting data objects according to a global one-dimensional criterion imposes too strong a limitation on the way spatial clustering can be achieved. Instead of sorting, the split and merge routines of the particular index structure

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

that is desired for the given setting are exploited for building an extremely efficient temporary data structure. The latter one is derived from the so-called buffer-tree [Arg 96] and therefore we use the term buffer-tree throughout the paper. The buffer-tree differs from the target index structure mainly in the following two points. First, each internal node of the buffer-tree has an additional (external) buffer where records are (temporarily) stored. Second, multiple insertions are processed simultaneously in the buffer-tree, in the following sense. A node in the tree defers an arriving insertion process and stores the corresponding record in its buffer. When the number of records in the buffer exceeds a predefined threshold, the insertion processes of all records in the buffer advance to the next level of the tree. In order to determine the subtrees (to which the insertions are directed), it is sufficient to read the node (with the required routing information) into main memory only once. This may give a first intuitive hint that the amortized insertion cost of the buffer-tree is much lower than the insertion cost of the original index structure. In spite of these differences, the buffer-tree allows to build up the desired index structure incrementally bottom-up, one level at a time. Our approach can also be considered a generalization of the bulk loading method based on seeded trees [LR 95]. In contrast to our approach, seeded trees are restricted to create a forest of R-trees. In particular, the resulting index structure does not always fulfill the properties of an R-tree. Moreover, seeded trees gives no performance improvement in the worst-case in comparison to the original insertion algorithm of an R-tree.

One of the most important advantages of our approach is its applicability to a broad class of tree-based index structures. Among them are B+-trees, R-trees [Gut 84], TV-trees [LJF 94], GIST [HNP 95], buddy-trees [SK 90], LSD-trees [HSW 89], and multiversion B-trees (MVBTs) [BGO+ 96] and their relatives [LS 89]. Therefore, we describe algorithms in a generic fashion without making specific assumptions about the underlying index structure. However, in order to make our approach more concrete, we consider the R-tree as our prime example in this paper. In particular, we show for the R-tree that the I/O performance of our approach is asymptotically equal to external sorting [AV 88].

The remaining paper is structured as follows. The next section introduces the underlying I/O model and our terminology. Section 3 of the paper introduces the buffer-tree. An example illustrates the basic ideas of our bulk loading algorithms. Section 4 describes the generic algorithms in detail. In Section 5, we consider the worst-case performance of these algorithms when they are applied to R-trees. Section 6 presents results obtained from an empirical performance evaluation. The conclusions are in Section 7. In an appendix, we summarize the symbols used throughout the paper.

2 Preliminaries

In this section we introduce our most important notions. Moreover, we discuss first the underlying I/O model and then the requirements on the underlying index structure.

2.1 The I/O model

Our I/O model corresponds to a disk. We assume that a disk is partitioned into pages of fixed size, with random access to each page at unit cost. At most B records (data objects) can be stored in a data page. Each access to disk transfers one page; we denote this as one I/O. The performance of our algorithms is measured in the number of I/Os needed for performing a sequence of N insertions. In particular, we do not focus on the I/O-cost of a single insertion. We make use of all available main memory (a supposedly large amount) for the operations of the bulk loading algorithm; let M denote the maximum number of records the available main memory can hold. The I/O cost of the algorithms is expressed in terms of N, M and B, i.e., none of these three parameters is viewed as a constant. We will abbreviate N/B and M/B by n and m, respectively.

Assuming this I/O model it was shown [AV 88] that external sorting requires $\Theta(n \log_m n)$ I/Os in the worst-case. The I/O cost of bulk loading a one-dimensional index structure that preserves the ordering of data (e.g., B+-tree) is therefore asymptotically optimal in the worst-case, if it meets the lower bound of external sorting. Therefore, our goal is to achieve this bound for bulk loading multidimensional index structures, without sacrificing search performance.

2.2 External tree-based index structures

In the following, we assume an index structure to be a tree. Each node of the tree corresponds to a page on disk. Routing information is stored in *index nodes*, whereas data records are stored in the *data nodes*. The index nodes together with the data nodes form a tree; the data nodes are the leaves of the tree. We call the part of the tree that consists of the index nodes the *index tree*; the entries of its *leaf nodes* contain references to data nodes, while the entries of its *internal nodes* contain references to index nodes. A node also corresponds to a d-dimensional *(node) region* of the dataspace; the records/entries of a node have to lie in the corresponding region. The region of a node will be stored with the reference to the node as an entry in an index node on the next higher level. For an R-tree, the region of an index node is the minimum bounding rectangle of the regions in its child nodes.

Most (multidimensional) index structures show great similarities in their internal interface. Similar to [HNP 95], we assume that the index structure provides the following operations:

- *InsertIntoNode*: insert a record into a data node or an entry into an index node.
- *Split*: split an overfull node into two. Note that there are some index structures which guarantee that storage in nodes is used at least to a certain degree. For example, nodes of a B+-tree will be used at least to a degree of 50%.
- *ChooseSubtree*: for a data record and an index node, choose a subtree for inserting the record. We assume

that *ChooseSubtree* also accepts a node region as parameter (instead of a data record).

Note that these methods will generally be available in multidimensional index structures. The only special requirement about these operations is that *ChooseSubtree* must be able to choose a subtree for both record and *(node) region*. This, however, is not a problem for index structures like the R-tree, where records and regions are rectangles. Other index structures like the B+-tree or the multiversion B-tree can easily be extended to support such an operation. The details for the MVBT are given in [BSW 96].

3 The Basic Idea of Bulk Loading

In this section, we first describe the abstract data structure of the buffer-tree. Thereafter, the basic idea of bulk loading will be illustrated by an example where the buffer-tree is made concrete for bulk loading an R-tree.

3.1 Review of the Buffer-Tree

Arge [Arg 96] proposed the buffer-tree for making the plane-sweep paradigm applicable to the case when the data set is too large for being kept resident in main memory. The plane-sweep paradigm is a key technique in the area of computational geometry [PS 85]. In order to be general enough, we modify his definition of the buffer-tree slightly.

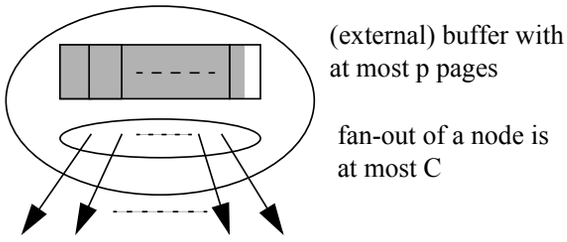


Figure 1: An index node of the buffer-tree

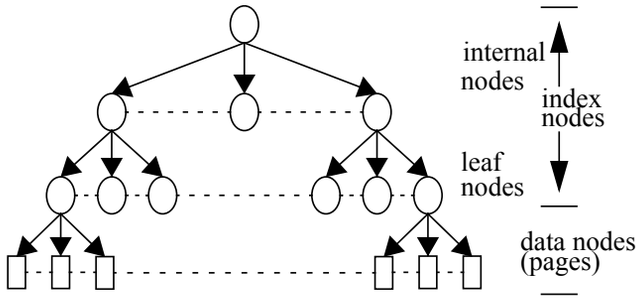


Figure 2: Structure of a buffer-tree

Definition 1: A tree-based index structure is a buffer-tree of order C and maximum buffer capacity p, if the following properties are fulfilled:

(i) each index node contains at most C entries; these entries refer to subtrees;

(ii) each index node contains a buffer of at most p occupied pages;

(iii) except for the last page, the occupied pages of the buffer are guaranteed to be full.

The structure of the buffer-tree is illustrated in Fig. 2. We distinguish between three types of nodes: internal nodes, leaf nodes and data nodes. Internal nodes and leaf nodes are index nodes whose structure is depicted in Fig. 1. Each of these nodes consists of a buffer with at most p pages and a routing table with at most C references to subtrees. Note that in our terminology a buffer of a node can change its size dynamically. We say that a *buffer is full*, if it contains p full pages (i.e., B*p records). A *buffer is empty* if it contains no pages (records). In an intermediate state of an operation, a buffer can contain more than p pages. We say then that there is an *overflow of the buffer*. Parameter C is also called the *branching factor*; it only depends *on the available main memory* and not as usually on the physical page size. That is, the size of an index node in general is larger than the physical page size. The size of a data node however is equal to the size of a physical page.

In contrast to an index structure designed for individual queries, multiple insertions are processed in the buffer-tree simultaneously. An insertion process is in one of the three states *active*, *blocked* and *terminated*. Only one of the insertion processes is active at a time, whereas the others are blocked or terminated. We say that an insertion process is *terminated* if its record is in a data page. An active insertion process becomes *blocked* when it arrives at an index node. The corresponding record is then inserted into the buffer of the index node. At a later point in time, a blocked insertion process will be reactivated.

3.2 Example

Before going into more details, we first give an example to illustrate how buffer-trees are used for bulk loading R-trees. The R-tree [Gut 84] is a height-balanced tree suitable for organizing a collection of multidimensional (rectilinear) rectangles in a dynamic setting. The distinctive property of R-trees is to represent each of the data rectangles only once in the data structure. This, however, leads to the undesirable property that there is overlap between the rectangles of index entries which are stored on the same level of the tree. An insertion of a new rectangle can increase this overlap and hence decrease geometric selectivity; therefore, researchers have studied the problem of improving the insertion algorithm [BKSS 90]. More precisely, new algorithms have been developed for splitting nodes and for choosing a subtree where the insertion process will continue. All of these algorithms were designed such that an insertion requires the retrieval of nodes from a single path of the tree.

In our example of bulk loading an R-tree, we consider a set of 25 rectangles $\{r_1,\ldots,r_{25}\}$, see Fig. 3. In order to keep our example manageable, we assume the following setting of

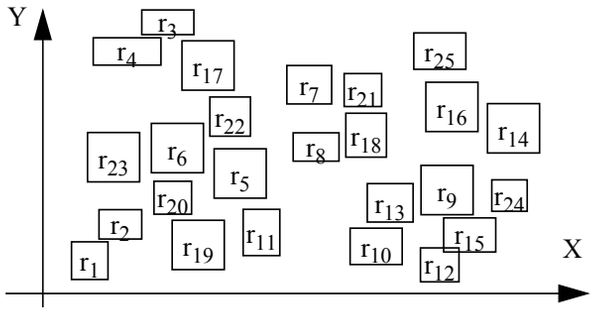


Figure 3: Sample set of 25 rectangles

the parameters: B (capacity of a data page) = 3, C (branching factor) = 4, p (number of pages in a buffer) = 2.

In Fig. 4 we depicted the buffer-tree after having started the first 23 insertion processes, where rectangles are inserted in the order of their indices into an initially empty tree. The insertion proceeds as in an R-tree. The buffer-tree consists of three index nodes N_1, N_2, N_3 and five data pages $P_1,\ldots,P_5$. Twelve of the 23 insertion processes have reached the leaf level, and therefore, these processes have been terminated. The other insertion processes are still alive, but blocked. For example, the insertion process of rectangle r_{17} is waiting at node N_2. Whenever a node blocks an insertion process the corresponding rectangle is temporarily stored in its buffer. Note that these buffers are generally not resident in main memory. As required in condition (iii) of Definition 1, the buffer of each of the index nodes contains at most p (= 2) pages.

An insertion into the buffer-tree is processed very similarly to an insertion into an R-tree, except that an index node keeps and blocks arriving insertions and reactivates them at a later point in time. Let us consider the insertion process of rectangle r_{24}. Since the buffer of the root page is not full, the insertion process is deactivated at the root and r_{24} is inserted into the root buffer. In order to support the insertion into the root buffer efficiently, we keep the last (occupied)

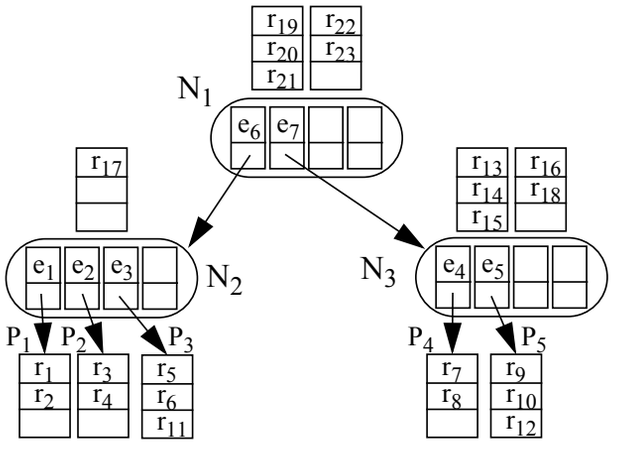


Figure 4: Example buffer-tree after having started 23 insertion processes

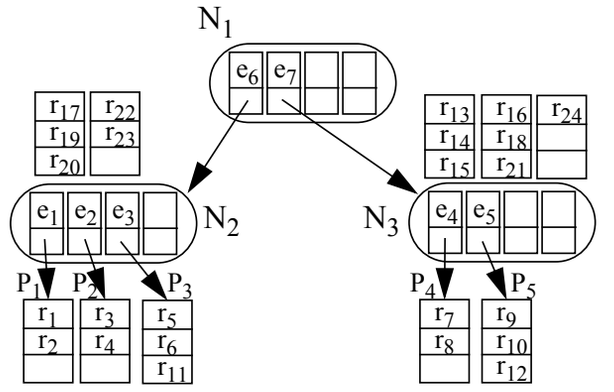


Figure 5: Example buffer-tree after clearing the root buffer

page of the root buffer in main memory, as long as no structural change is performed. Thereafter, the insertion process of rectangle r_{25} starts. Since the root buffer is already full, r_{25} cannot be inserted there. Therefore, a structural change of the buffer-tree is performed first. The structural change consists of reactivating all processes (one by one) which are blocked at the root page. Each of these processes takes its rectangle from the buffer and advances to the next level of the buffer-tree (by calling *ChooseSubtree*). We also say that the buffer (of the root page) is *cleared*. The situation after having cleared the buffer is illustrated in Fig. 5. As depicted, the buffer of N_1 has no page, r_{19}, r_{20} r_{22}, r_{23} are now in the buffer of N_2 and r_{21}, r_{24} are in the buffer of N_3.

After clearing of a buffer it may happen that some of the buffers at the next level contain more than p (= 2) pages, see

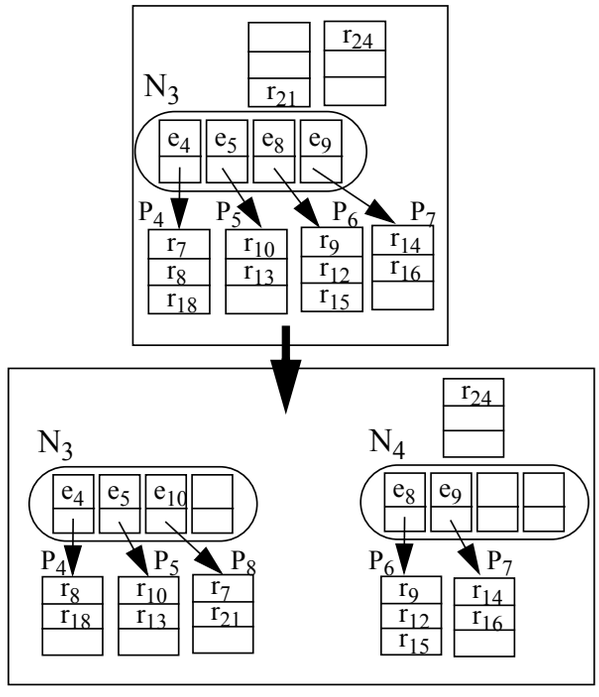


Figure 6: Splitting of an index node N_3 (at the top) into node N_3 and N_4 (at the bottom)

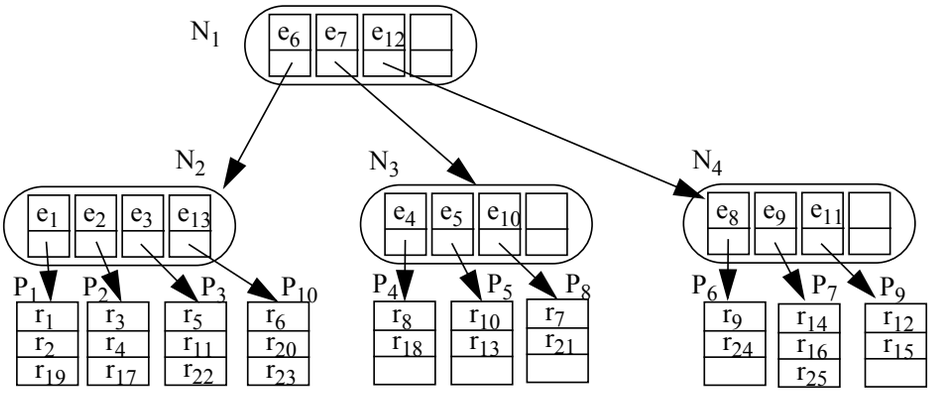


Figure 7: The example buffer-tree after having terminated all insertion processes

for example the buffer of node N_3 in Fig. 5. These overflows are then eliminated again by clearing those buffers one by one. In our example, therefore, the insertion processes of rectangles r_{13}, …, r_{16}, r_{18}, r_{21}, r_{24} continue to the leaf level of the buffer-tree. Note that an overflow in a data page is treated in the same fashion as known from the R-tree: a page is first split into two and then the corresponding entries are posted to the parent node.

At the top of Fig. 6, the subtree rooted at node N_3 is depicted after having cleared the buffer of N_3 partially, with r_{21} and r_{24} still remaining. Let us assume that rectangle r_{21} should be inserted in page P_4. Since P_4 is already full, an overflow occurs and therefore, a new page P_8 and a new entry e_{10} is created. The insertion of e_{10} in node N_3 again produces an overflow of the routing table. Therefore, a new node N_4 (with its own buffer) is created, and some of the entries of N_3 are moved to N_4, see the bottom of Fig. 6. For each record from the buffer of N_3, *ChooseSubtree* is called to determine whether the record remains in the buffer or moves to the buffer of N_4. In our example, the buffer of N_4 contains rectangle r_{24}, whereas the buffer of N_3 is empty (quite a special situation). Only the buffer with the highest number of entries is then totally cleared, whereas the other buffer remains unchanged. In our example, the buffer of N_4 is cleared (i.e., the insertion process of r_{24} is activated). The entries of the nodes are then posted to the parent node N_1. Thereafter, the structural change is finished and rectangle r_{25} is eventually inserted into the root buffer.

After having started all insertion processes, some of them did not arrive at a data node, but are still blocked at an index node. Therefore, bulk loading is not terminated yet. What remains to be done is to empty all non-empty buffers of the buffer-tree in a depth-first order, starting from the root. Since we already have illustrated this operation, we will not go into further details here. In Fig. 7, the final buffer-tree is depicted. Pages P_1,…,P_{10} now represent the data nodes of the target R-tree.

At first glance, it may seem that the index nodes of the buffer-tree already correspond to the index nodes of the target R-tree. This is however not the case in general, since we took advantage in choosing the size of an index node different from the physical page size. Now, in order to arrive at the target R-tree, we create a second buffer-tree by inserting the entries e_1, …, e_5, e_8, e_9, e_{10}, e_{11}, e_{13} which are in the index leaf nodes of the first buffer-tree. The second buffer-tree will produce all the index pages of the target R-tree that are above the leaves. In this way, each level of the target R-tree is built by creating a further buffer-tree.

4 The generic algorithm in details

In this section, we give a detailed description of the bulk loading algorithms without making special assumptions about the underlying index structure. Let us recall that the index structure provides an interface with the following three operations: *InsertIntoNode*, *Split* and *ChooseSubtree*. These operations will be frequently used in the following algorithms.

An obvious implementation of the buffer-tree would be to use the process facilities of the operating system for creating a process for each insertion. The overhead for managing all these processes (in time and storage cost) would however be very high, and therefore we decided against using multiple processes. In our approach, only one process is used for simulating all the different “processes”.

The basic structure of our algorithm for bulk loading is somehow similar to the algorithm of an (ordinary) insertion in an index structure such as an R-tree. In the buffer-tree, an insertion operation of a record first traverses the tree from the root to a leaf, whereas a restructuring operation can then traverse from the leaf backwards to the root. The unique feature of the buffer-tree is that multiple insertion operations and multiple restructuring operations are processed simultaneously in the tree. In the example of the previous section we have already discussed how insertions are processed. A restructuring operation is triggered by an overflow in a data node or by an overflow in a routing table of an internal node. A restructuring operation consists of a split of the overflowing node and an insertion of the new entry in its parent index node. Similar to an insertion process of a record, an internal index node also defers an incoming insertion operation of an entry. The index node temporarily stores the entry in a list. When all subtrees of the node have finished their restructuring operations (i.e., each of these subtrees fulfills the properties of a buffer-tree), all the entries of the list are inserted in the rout-

ing table of the node. This can again produce overflows and further restructuring operations.

The main algorithm of bulk loading is *StartInsert* which is given below. We assume that the buffer-tree initially consists of an index page with one reference to an empty data page. *StartInsert* initiates a new insertion process at the root of the buffer-tree (*Root*). First, the number of records in the buffer of *Root* is computed by calling *BufferLoad.* Usually, the load is lower than B*p (i.e., the condition of the if-statement is not fulfilled) and then, only *InsertIntoBuffer* is called. This routine simply blocks the insertion process and inserts the new record into the root buffer. In order to save storage space, the buffer consists of the minimum number of pages necessary to hold the records. Hence, the insertion goes into the currently last page of the buffer. If this page is already full, a new page will be added to the buffer. After a sequence of B*p calls of *StartInsert*, the buffer is full and the condition of the if-statement is fulfilled for the next insertion process. Then, the buffer of *Root* is cleared, i.e., insertion processes are reactivated. As the result of a call of *ClearBuffer,* we obtain a list of entries (*new_children*) which refer to new child nodes of *Root*. Next, *InsertChildren* inserts these entries of *new_children* into *Root*. Moreover, a new sibling is created when an overflow occurs in *Root* or in one of the previously created siblings. Eventually, *InsertChildren* returns a list of the new siblings to *StartInsert*. If this list is not empty, a new root of the buffer-tree is created from this list.

```
ALGORITHM StartInsert (Root, R)
  (* An insertion process is started for record R in a
     buffer-tree with rootnode Root. *)
  IF BufferLoad(Root) = B*p
    new_children := ClearBuffer(Root);
    new_siblings := InsertChildren(Root, new_children);
    IF new_siblings is not empty
      create a new root from new_siblings;
      (* update Root *)
  InsertIntoBuffer(Root, R);
END StartInsert.
```

In the following, we give a detailed description of *ClearBuffer* and *InsertChildren*. Let us first consider the latter one (see below). Algorithm *InsertChildren* inserts entries of a list (*new_children*) into a node (*Node*). First, entries are inserted only into *Node* until an overflow occurs. The routing table of *Node* is then split into two and a new entry (referring to the new node) is inserted into a list (*new_siblings*). At that time it might be that some of the entries from *new_children* are still unprocessed. In the next pass through the for-loop, the entry has to be assigned to one of the two nodes. *ChooseSubtree* is therefore called. After the for-statement is processed, the records from the buffer of *Node* are redistributed among *Node* and its siblings. Finally, *new_siblings* is returned.

```
ALGORITHM InsertChildren(Node, new_children)
(* The entries from children are inserted in Node. The algorithm returns a list of nodes (new_siblings) which are derived from Node through split operations. *)
  new_siblings := {};
  FOREACH entry E from new_children
    all_siblings := new_siblings ∪ {entry of Node};
    ParentNode := ChooseSubtree(all_siblings, E);
    InsertIntoNode(ParentNode, E);
    IF overflow in ParentNode
      Split(routing table of ParentNode);
      (* split routine of the index structure *)
      insert the new entry into new_siblings;
  IF new_siblings is not empty
    (* split the buffer of Node *)
    (* all_siblings = new_siblings ∪ {entry of Node} *)
    FOREACH record R from the buffer of Node
      Target := ChooseSubtree(all_siblings, R);
      move R from buffer of Node to the one of Target;
  RETURN new_siblings;
END InsertChildren.
```

Most important to the buffer-tree is the algorithm *ClearBuffer* for clearing an overflowing buffer of a node. This corresponds to activating all blocked processes at a node. *ClearBuffer* distinguishes between internal index nodes and leaf nodes of the index and calls algorithm *ClearInternalBuffer* and *ClearLeafBuffer*, respectively.

For the case of an internal index node, the idea of the algorithm is to read the first B*p records from the buffer and determine the nodes to which the records should be assigned. For a given record *R* and index node *Child*, the routine *InsertIntoBuffer* puts *R* into the last page of the buffer of *Child*. When an overflow occurs in the buffer (i.e., *BufferLoad*(*Child*) is greater than B*p), we provisionally insert the corresponding node into a list (*overflow_list*). Note however that the buffers of the nodes in *overflow_list* may still receive records from the buffer of *Node*. After the buffer is cleared of B*p records, we traverse through *overflow_list* and apply *ClearBuffer* to each of the nodes in a recursive fashion. The (indirect) recursive calls return lists of entries which refer to new child nodes. After that, *InsertChildren* inserts the entries of these lists into *Node*.

An important feature of *ClearInternalBuffer* is that a buffer is only partially cleared from the first B*p records. The reason is that then the number of overflow records in a buffer is at most B*p (because a node can receive at most B*p records from its parent). To put in other words, a buffer contains in all situations at most 2*B*p records. After emptying a buffer partially, it is therefore guaranteed that a buffer contains at most B*p records.

```
ALGORITHM ClearInternalBuffer(Node)
(* Clears the buffer of Node whose entries refer to an index
   node. *)
   overflow_list := {};
   FOREACH R of the first B*p records in buffer of Node
      (* reactivate the insertion process of R *)
      Child := ChooseSubtree(Node, R);
      InsertIntoBuffer(Child, R);
      IF BufferLoad(Child) > B*p
         insert Child into overflow_list;
   new_children := {};
   FOREACH Child from overflow_list
      add ClearBuffer(Child) to new_children;
   new_siblings := InsertChildren(Node, new_children);
   RETURN new_siblings;
END ClearInternalBuffer.
```

The second case (algorithm *ClearLeafBuffer)* is different from the first one. Consider a leaf node, say *Node*, whose buffer should be cleared. For each record in the buffer, *ChooseSubtree* delivers the data node where the record is then inserted. Since the size of data nodes is limited by the physical page size, such nodes have to be split occasionally, and corresponding entries then have to be inserted into the routing table of *Node*. This produces an overflow if the routing table of *Node* contains already C references to data nodes. The overflow is eliminated by a split, i.e., both routing table and buffer are split into two. The index entries which point to new nodes are inserted into *new_siblings*. Thereafter, the clearing algorithm continues with the node whose buffer contains most of the data.

```
ALGORITHM ClearLeafBuffer(Node)
(* Clear buffer of Node which is an internal leaf node. Re-
   turns a list of entries of new siblings of Node. *)
   new_siblings := {};
   FOREACH record R in buffer of Node
   (* reactivate the insertion process of R *)
      DataNode := ChooseSubtree(Node, R);
      InsertIntoNode(DataNode, R);
      (* insertion process is terminated *)
      IF overflow in DataNode
         Split(DataNode);
         (* split routine of the index structure *)
         apply corresponding entries to Node;
         (* immediately *)
         IF overflow in Node
            split Node into two (let N be the new node);
            (* split the routing table and the buffer *)
            insert the entry of N into new_siblings;
            update entry of Node;
            IF BufferLoad(N) > BufferLoad(Node)
               add ClearLeafBuffer(N) to new_siblings;
            ELSE
               add ClearLeafBuffer(Node) to new_siblings;
            RETURN new_siblings; (* exit of algorithm *)
   RETURN new_siblings; (* is empty *)
END ClearLeafBuffer.
```

After *StartInsert* has been called for all records, the buffers of the nodes are cleared in a depth-first order. Thereafter, all records reside in the data pages. Moreover, the data pages already represent the lowest level (the data level) of our multidimensional index structure. Therefore, this completes the algorithm for placing the data into the pages of the desired structure.

The question now arises how the index pages of the structure can be created. Note that the index nodes of the buffer-tree cannot be used directly, because the size of the nodes of the buffer-tree will generally not equal the size of nodes in the index structure. This leads to a more general problem: How can we efficiently transform a tree with fan-out x into a tree with fan-out y?

Our approach to the problem of creating the index levels of the index structure is very similar to the one described above for inserting data records. We start to build up a new buffer-tree. Instead of inserting records, we use the index entries of the nodes from the index leaf level (immediately above the data nodes) as input to the new buffer-tree. In order to insert entries (i.e., node regions), we follow the procedure for inserting records. Let us recall that *ChooseSubtree* also accepts node regions as input. As a result, we obtain a buffer-tree whose "data" pages contain all index entries which refer to data pages of the target R-tree. We repeat this process until we arrive at the root of the desired index structure.

The performance of bulk loading an index structure with the approach described in this section depends on the available main memory and on the performance of the required operations of the index structure. We therefore analyze in the next section, the performance of our method when it is applied to the R-tree.

5 Performance analysis: bulk loading R-trees

The members of the R-tree family are ideal candidates for exploiting our approach to bulk loading. They obviously fulfill all our requirements and provide the desired functionality. Note that *ChooseSubtree* is implemented in R-trees for both data rectangles and index entries. Moreover, based on the fact that an insertion requires at most $O(\log_B n)$ I/Os, we will show now that our approach of bulk loading requires $O(n \log_m n)$ I/Os for creating an R-tree.

In order to achieve the desired result, an adequate value for C (the branching factor of the buffer-tree) is important. Our goal is that the cost for clearing a buffer of an internal node should be small. Therefore, the routing table of the node, a page of its buffer and one page for each of the child nodes should be kept in main memory. As a consequence, C must fulfill the following inequality:

$$\textbf{(1)} \qquad m \geq \left\lceil \frac{C}{B} \right\rceil + C + 1$$

For sake of efficiency we choose C as the largest integer that fulfills the inequality above. Then, $C = \Theta(m)$ holds. For the sake of simplicity, we assume in the following that $2*p = C$.

Theorem 1: The total I/O cost for inserting N rectangles into an initially empty R-buffer-tree (this is a buffer-tree built according to the rules of the R-tree) is $O(n \log_m n)$.

Proof:

The proof is structured similarly to the one given in [Arg 96]. We distinguish between three components of the total cost: the I/O cost for clearing the buffers of internal index nodes, the I/O cost for clearing the buffers of index leaf nodes, and the I/O-cost for splitting the nodes of the buffer-tree.

Let us first discuss the I/O-cost for clearing the buffers of the internal index nodes. A clearing of a full buffer, say X, requires O(m) I/Os, because X consists of at most C (= O(m)) pages and we can accommodate one page for each of the buffers of the child nodes in main memory. Therefore, it suffices for each of the data rectangles in the buffer X to pay for O(1/B) I/Os. Since a data rectangle will be in at most $O(\log_m n)$ full buffers on its insertion path, a data rectangle has to pay for $O(1/B * \log_m n)$ I/Os. Therefore, N rectangles have to pay a total of $O((N/B) \log_m n) = O(n \log_m n)$ I/Os.

Second, let us consider the cost for clearing a buffer X of an index leaf node. The I/O cost of this operation increases monotonically with the number of data nodes. There will be, however, not more than (2+(B-1)/b)*C data nodes (after having cleared the buffer), as the following thoughts show. Recall that b denotes the minimum number of entries in a node of the R-tree. The factor B/b is a constant for the R-tree, with B/b > 2; it is 3 in many R-tree implementations.

The worst-case scenario that produces a maximum number of data nodes is the following: The buffer consists of B*C (= 2*B*p) data rectangles. These data rectangles are inserted into the data nodes referenced by the index leaf node one by one. There are C data nodes, each of them is full (i.e., the data node contains B records). Now, each of the first C insertions of rectangles splits a data node into two, in such a way that one of the data nodes obtains b rectangles, and the other gets B-b+1. After that, a new data node can be generated only after having inserted a sequence of b rectangles. Thus, this process will result in generating at most C+(B-1)*C/b data nodes per buffer. Together with the C data nodes that are present in the beginning, we get the claimed bound.

The generation of data nodes is, of course, not the only effect of clearing an index leaf buffer. In addition, in the index leaf node, we have to insert (1+(B-1)/b)*C new index entries. This again triggers 1 + B/b(1+(B-1)/b) = O(1) split operations in the worst case. The cost for splitting internal index nodes will be discussed in the following paragraph. It is important for clearing the original buffer, however, that the splitting process creates only a constant number of new nodes. Each of these nodes requires at most O(C) I/Os for clearing its buffer (which represents a subset of the original buffer). Thus, clearing any original buffer requires at most O(m) (= O(C)) I/Os in total.

Third, let us consider the cost of splitting internal index nodes of the tree. Recall that a node that has been split into two will have no overflow for the next C*(b/B) insert operations, where an insert refers to inserting a new data page into the tree. According to [MS 81], there will be at most O(n/C) split operations of internal nodes. A split of an internal node can also split a buffer into two and therefore, a split of an internal node requires O(C) I/Os. Overall, the total cost for splitting nodes will therefore be O(n).

Our desired result directly follows from the discussion of the three cases above. ❐

Theorem 2: The I/O cost for clearing all buffers of a R-buffer-tree (this is a buffer-tree built according to the rules of the R-tree) is O(n).

Proof:

The total number of buffers is O(n/m). Each clearing of a buffer requires O(m) I/Os. From the proof of Theorem 1 it follows that the corresponding split operations require O(n) I/Os. This proves the theorem. ❐

Theorems 1 and 2 show that $O(N/B \log_m N/B)$ I/Os are sufficient for generating a R-buffer-tree from a set of *N* rectangles such that all rectangles reside in data pages. These data pages already represent the leaf level of the R-tree. The other levels are generated in a recursive fashion. In the next step of the recursion, we consider the O(N/B) index entries which are referring to the data pages. These entries are processed in the same way as the data rectangles: they are inserted into an initially empty R-buffer-tree. The I/O cost for building up the R-buffer-tree is $O(N/B^2 \log_m (N/B^2))$. In general, the I/O cost for generating the (h-i)-th level of the R-tree is $O(N/B^{i+1} \log_m (N/B^{i+1}))$. The total cost of bulk loading an entire R-tree is therefore

$$\sum_{i=0}^{h} O\left(\frac{n}{B^i} \log_m \frac{n}{B^i}\right) \le O\left(\sum_{i \ge 0} \frac{n}{B^i} \log_m \frac{n}{B^i}\right)$$

$$= O\left(\sum_{i \ge 0} \frac{n}{B^i} \log_m n\right) = O\left((n \log_m n)\left(\sum_{i \ge 0} \frac{1}{B^i}\right)\right)$$

$$= O(n \log_m n)$$

This proves the following theorem.

Theorem 3: The I/O cost for bulk loading an R-tree is $O(n \log_m n)$.

The result of Theorem 3 is asymptotically optimal for the following reasons: First, it meets the lower bound of external sorting. Second, a one-dimensional R-tree can obviously be used for sorting.

In contrast to previous methods for bulk loading R-trees, our approach gives a different method for each member of the R-tree family. The bulk loading methods differ in the routines *ChooseSubtree* and *Split* that they inherit from the different R-trees.

Data Page Capacity (B)	I/O Data	I/O Directory	I/O Buffer
10	41.9%	0.4%	57.7%
20	44.9%	0.4%	54.7%
30	47.1%	0.4%	52.5%
40	48.0%	0.4%	51.6%
50	48.5%	0.4%	51.1%

Table 1: Relationship between different access types

6 Empirical Performance Evaluation

In this section, we report the results of a preliminary set of experiments for bulk loading applied to a structure different from R-trees, namely the multiversion B-tree (MVBT) [BGO+ 96]. All the results are obtained from MVBTs; similar results are expected when our bulk loading method will be applied to members of the R-tree family as well as to other index structures. The main objective of our set of experiments is to show that our approach of bulk loading is not only asymptotically efficient in the worst-case, but also gives excellent performance in the average-case. To this end, the I/O cost of bulk loading an MVBT is compared with the I/O cost of *tuple loading*, i.e., an MVBT is created by inserting records one by one. Notice that when the same set of data records is used, the MVBT created by bulk loading is identical to the one created by tuple loading. Thus, the search performance of MVBTs is not affected by bulk loading.

In our experiments, the I/O cost only refers to the cost of creating the leaf level of the MVBT. This is justified because the I/O cost for building the other levels will be considerably lower. The leaf level consists of data pages whose size is equal to the physical page size. For bulk loading we count all the I/Os required for building the leaf level, whereas for tuple loading we count the I/Os for reading and writing data pages only.

For bulk loading, we were primarily interested in the impact of the following parameters on the I/O cost: m (number of pages in main memory) and B (data page capacity). In our experiments, the routing table of an index node can be kept in one physical page. We therefore count for each access to the routing table one I/O. Parameter C (branching factor of the routing table) is chosen according to the rule from Section 5 (inequality 1). Therefore, C = m - 2 holds. The parameter p (capacity of a buffer of an index node) was always set to C in our experiments. Both data pages and buffer pages of the MVBT correspond to physical pages.

For tuple loading, we used a buffer of m pages in main memory. Therefore, both methods of index creation use the same amount of main memory (namely m pages). The buffer was organized according to the LRU replacement strategy. The I/O cost only refers to the number of disk accesses.

The set of experiments were performed in the following way. In each experiment, the same level of two MVBTs

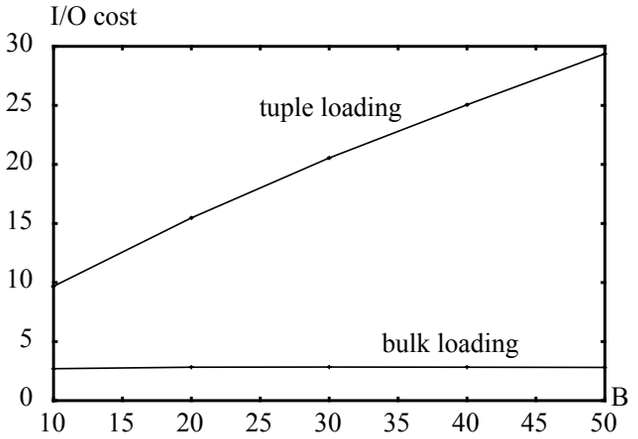


Figure 8: The I/O cost per data page as a function of B (data page capacity) for main memory of 200 pages.

was created by inserting 100,000 records. When an insertion was performed, the corresponding record was generated at random. The process of creation differed only in whether tuple loading or bulk loading was used. For bulk loading, we distinguished between accesses to data pages, buffer pages and routing tables (of the index nodes).

Let us first discuss the results of bulk loading only. In Table 1 we present the relationship among the different types of accesses (data page, buffer page, routing table) as a function of B (data page capacity). The available main memory m was set to 200 pages in these experiments. The results show that most accesses are spent for buffer pages. For an increasing page capacity, the number of buffer page accesses approaches the number of data page accesses. Accesses to routing tables have only a minor impact on the total I/O cost.

Next let us compare the cost of the two methods for creating the MVBTs. We varied the parameter m= 25, 50, …, 200, and the data page capacity, B = 10, 20, …,50. All experiments gave similar results to those plotted in Fig. 8, where m= 200. The curves show the number of I/Os divided by the number of data pages, as a function of B. As expected from our worst-case analysis, the cost of tuple loading increases almost linearly in B, whereas the cost per data page using bulk loading is independent from B. Bulk loading requires roughly three I/Os per data page which is comparable to the I/O cost of external sorting.

7 Conclusions

In this paper we address the problem of bulk loading a set of records into an initially empty tree-based index structure. From B+-trees it is well known that bulk loading is considerably more efficient than inserting records one by one. We present a generic approach to bulk loading multidimensional index structures that avoids sorting the data beforehand according to a global ordering. Instead of sorting, the split and merge routines of the target index structure are exploited for building an extremely efficient temporary data structure. From the temporary structure, the de-

sired index structure is built up incrementally, one level at a time.

In contrast to previous approaches, our generic algorithms are not restricted to R-trees, but can also be applied to a broad class of tree-based index structures. One member of this class is the multiversion B-tree (MVBT) which served as an example in a set of preliminary experiments. For bulk loading R-trees, we showed that our approach requires $O(n \log_m n)$ disk accesses in the worst-case where n and m denote the number of data pages and the available main memory (in pages), respectively. This result is equal to the lower bound of external sorting. The same result also holds for the MVBT [BSW 96]. Results obtained from experiments with an implementation of the MVBT confirmed that bulk loading is considerably more efficient than inserting records one by one.

In our future and current work, we are interested in the following issues. First, we are currently investigating improvements of our algorithms with respect to CPU-cost (which has not been considered in this paper). This is important for index structures like R-trees, which are known to consume much CPU-time while inserting records. Moreover, we are interested in algorithms for bulk loading under a different I/O model which takes into account that sequential I/Os are more efficient than random I/Os.

References

[Arg 96] L. Arge, "Efficient External-Memory Data Structures and Applications", BRICS Dissertation Series, DS-96-3, University of Aarhus, 1996.

[AV 88] A. Aggarwal and J. S. Vitter. "The Input/Output Complexity of Sorting and Related Problems", Communications of the ACM, 31(9), September 1988, 1116-1127.

[BKSS 90] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger: *"The R*-tree: An Efficient and Robust Access Method for Points and Rectangles",* Proc. ACM SIGMOD, 1990, pp. 322-331.

[BSW 96] J. van den Bercken, B. Seeger, P. Widmayer: "A Generic Approach to Bulk Loading Multidimensional Index Structures". Technical Report 13, University of Marburg, Computer Science Department, 1996.

[DKL+ 94] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, J.-B. Yu: "Client-Server Paradise", Proc. VLDB 1994: 558-569

[BGO+ 96] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer, "An asymptotically optimal multiversion B-tree", VLDB Journal 5(4), 1996: 264-275.

[Gut 84] A. Guttman, "R-trees: a dynamic index structure for spatial searching", Proc. ACM SIGMOD, 1984, 47-57.

[Hen 96] A. Henrich: "Adapting a Spatial Access Structure for Document Representations in Vector Space", Proc. 5th Int. Conf. on Information and Knowledge Management (CIKM), 1996.

[HNP 95] J. Hellerstein, J. Naughton A. Pfeffer, "Generalized Search Trees for Database Systems", Proc. VLDB, 1995, 562-573.

[HSW 89] A. Henrich, H.-W. Six, P. Widmayer: "The LSD tree: Spatial Access to Multidimensional Point and Nonpoint Objects". Proc. VLDB 1989: 45-53.

[KF 93] I. Kamel, C. Faloutsos: "On Packing R-trees", Proc. 2nd Int. Conf. on Information and Knowledge Management (CIKM), 1993, 490-499.

[LEL 95] S. T. Leutenegger, J. Edgington, M. A. Lopez: "Efficient Bulk Loading of R-trees", Univ. of Denver, Technical Report 95-1.

[LJF 94] K.-I. Lin, H. V. Jagadish, C. Faloutsos: "The TV-Tree: An Index Structure for High-Dimensional Data". VLDB Journal 3(4): 517-542 (1994)

[LR 94] M.-L. Lo, C. V. Ravishankar: "Spatial Joins Using Seeded Trees", Proc. ACM SIGMOD, 1994, 209-220.

[LR 95] M.-L. Lo, C. V. Ravishankar: "Generating Seeded Trees from Data Sets". Symp. on Large Spatial Databases, in Lecture Notes in Computer Science, Vol. 951, 1995, 328-347.

[LS 89] D. Lomet, B. Salzberg, "Access Methods for Multiversion Data", Proc. ACM SIGMOD, 1989, pp. 315-324.

[MS 81] D. Maier, S. Salveter, "Hysterical B-trees", Information Processing Letters, Vol. 12(4), 1981, pp. 199-202.

[PS 85] F. P. Preparata, M. I. Shamos, "Computational Geometry: An Introduction", Springer-Verlag, 1985.

[RL 85] N. Roussopoulos, D. Leifker: "Direct Spatial Search on Pictorial Databases Using Packed R-Trees". Proc. SIGMOD 1985: 17-31.

[SK 90] B. Seeger, H.-P. Kriegel, "The Buddy-Tree: An Efficient and Robust Access Method for Spatial Data Base Systems". Proc. VLDB 1990: 590-601.

Appendix: List of symbols

b	minimum number of records (entries) in a node
m	M/B
n	N/B
p	capacity of the buffers of the buffer-tree (in pages)
B	capacity of a data page (in records)
C	branching factor of the routing table
M	maximum number of records in main memory
N	number of insertions

바로는 data에는 historical, spatial, image, text

database가 존재.

시간 및 공간적 data를 대부분은 2, 3차원 공간.

대부분이 객체들이 차원도 매우 높음.

∴ 대부분 index구조(ex. B+ tree)는 적절한 indexing

서능이 아님.

저장된 레코드 집합에 대한 여러번 index 구조를

처음부터 만드는 것: 일반적인 해상작업

bulk operation = bulk loading

기존의 R-tree나 B+ tree 및 Space-filling curves.

기반 다차원 index 구조는 bulk loading 문제가 해결이 안됨.

global one-dimensional criterion은.

→ 객체 공간 시 공간 clustering 특성이 강력한 제한을 부과하므로

공간 데이터 처리 분야에 폭넓은 응용 index 구조의 설계 및 평가

주된 활용하여 효율적인 외부 data 구조 구축

= Buffer tree : Target index structure의 기술

① Buffer Tree의 각 내부 node에는 레코드가 일정량 저장되는 추가 버퍼가 존재

② Buffer Tree에서 multiple insertion이 훨씬 빨라짐

개념

I/O model : Disk.

Disk : 고정된 크기의 page로 분할.

단위 비용으로 각 page의 random access.

Data page에는 최대 B개의 Record 저장 가능.

(Data Objects)

Disk의 한 번 access : 1 page 전송. ⇒ 이를 하나의 I/O라고 함

알고리즘의 정능 : N개의 insertion sequence 수행시 필요한 I/O 수

Bulk loading 알고리즘 작업을 위해 가용한 main-memory 사용.

M : 사용가능한 main memory가 저장할 수 있는 최대 Record 수.

알고리즘의 I/O 비용 : N, M, B로 표현

$N/B = n$. $M/B = m$.

external sorting: $\Theta(n \log_m n)$ I/Os 최악.

우리의 목표: 검색 성능 저하시키지 않고 multidimensional index 구조를 bulk loading 하는 것

작업공간상태일 때: buffer: p 페이지 이상도 가능. 그 다음에 해당 상태를 overflow라고 함.

buffer: 크기 동적으로 변경 가능.

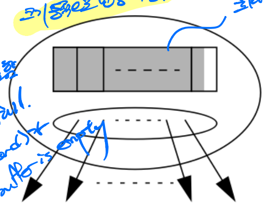
최대 p개의 page가 있는 buffer.

page는 Bm record 포함 = B x p Record

buffer가 p개의 page 포함 → buffer is full.

buffer에 page(record)가 없는 경우 → buffer is empty

parameter C: branching factor. 실제 page 크기보다 사용성을 main memory에 맞게.



subtree에 대한 최대 C개의 reference 가능 routing table로 구성

Figure 1: An index node of the buffer-tree

Buffer tree는 data node, index node로 구성

internal node의 entry: index node에 대한 참조 포함.

Index node가 tree 구조.

tree의 각 node: disk의 page

Routing 정보 → index node 저장

Data Record → data node 저장. (leaf node)

index tree.

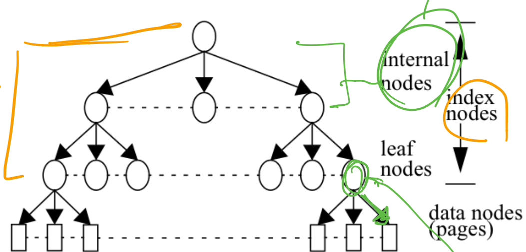


Figure 2: Structure of a buffer-tree

leaf node의 entry: data node에 대한 참조 포함.

→ 일반적으로 index node크기 > 실제 page 크기. 하지만 data node의 크기 = 실제 page 크기.

Node: dataspace의 d 차원 region에 해당함.

Node의 entry / record 모두 그 영역에 있어야 함.

Node의 region은 Node에 대한 참조와 함께 next higher level의 index node entry로 저장된다.

Operations.

- InsertIntoNode : Data node에 record insert.
  Index node에 entry

- Split : overflow된 node 분할

- ChooseSubtree : data record 및 index node에 대해,
  record 삽입을 위한 subtree 선택.
  (Data record 삽입시 Node organization parameter를 사용함)

Buffer-Tree

① 각 index node는 최대 M개의 entry contain
각 entry는 subtree 참조

② 각 index node는 최대 p개의 사용중인 page
같는 buffer contain

③ last page를 제외하고, buffer의 사용된 page는
가득 차게 보장된다.

Buffer Tree에서는 multiple insertion이 동시에 처리됨.

insertion process → active
- blocked
- terminated.

한 번에 하나의 insertion process만 active이고,
나머지는 전부 blocked or terminated.

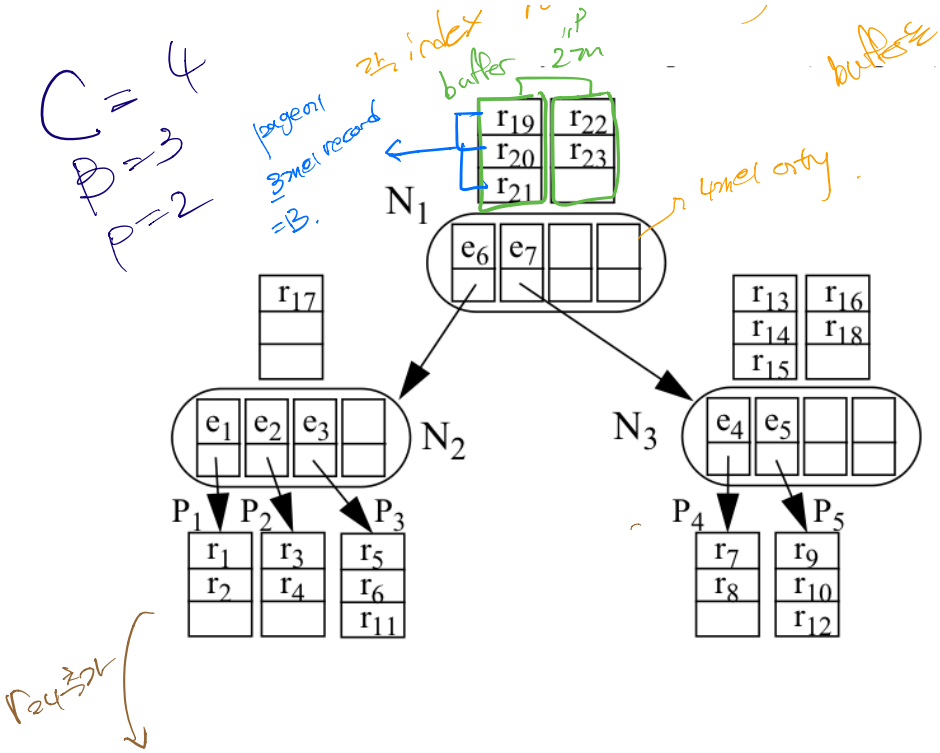
- 많은 record가 data page에 있는 경우. insertion process = terminated.
- active insertion process가 index node 도달 → blocked.
  그리고 해당 node가 index node의 buffer에 삽입..
  나중에 block된 process가 다시 reactivate됨

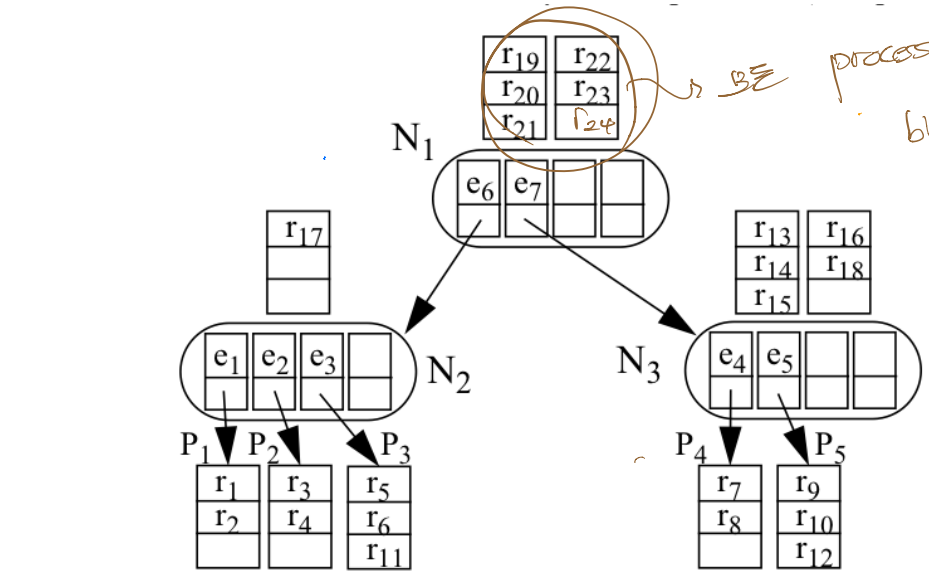
R tree : height balanced tree.
(multidimensional (rectilinear) rectangles의 collection을 organize 하는데 적합

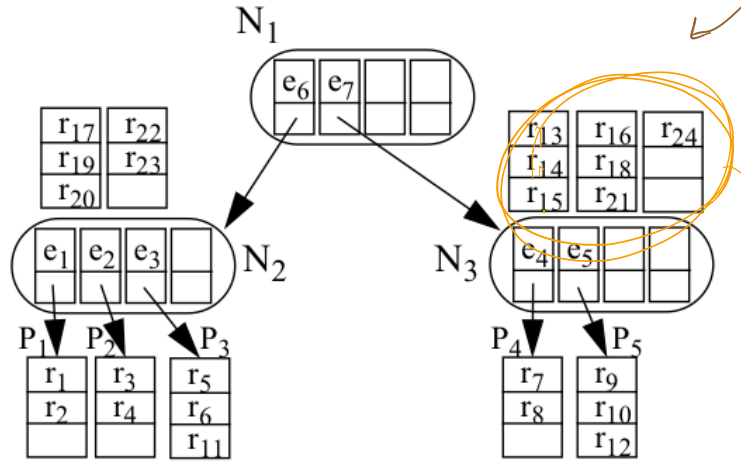
R tree의 특징은 data structure에서 각 data 사각형은 한 번만 나타남

⇒ same level에 저장하는. index entry 의 사각형들 간에 overlap 발생.

node 최대 C개의
only contain
, n pl (=2) 되어지


r_{19} r_{20} r_{21} r_{22} r_{23}
N_1
e_6 e_7
r_{17}
e_1 e_2 e_3
N_2
N_3
r_{13} r_{14} r_{15} r_{16} r_{18}
e_4 e_5
P_1 P_2 P_3 P_4 P_5
r_1 r_2 r_3 r_4 r_5 r_6 r_{11}
r_7 r_8 r_9 r_{10} r_{12}


r_{19} r_{20} r_{21} r_{22} r_{23} r_{24}
N_1
e_6 e_7
r_{17}
e_1 e_2 e_3
N_2
N_3
r_{13} r_{14} r_{15} r_{16} r_{18}
e_4 e_5
P_1 P_2 P_3 P_4 P_5
r_1 r_2 r_3 r_4 r_5 r_6 r_{11}
r_7 r_8 r_9 r_{10} r_{12}



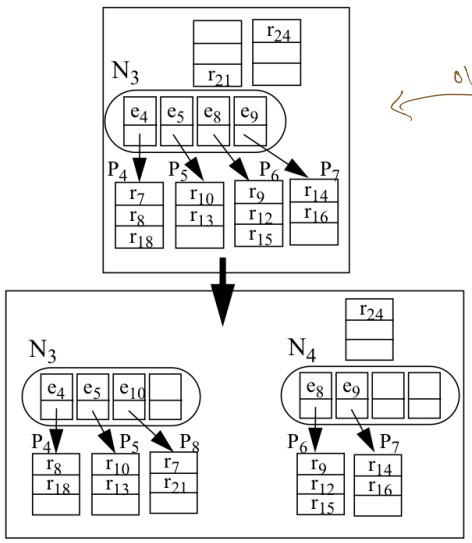


Figure 6: Splitting of an index node N_3 (at the top) into node N_3 and N_4 (at the bottom)

... 새로운 Node N4 생성, ...
(자체 buffer 포함).

N3 버퍼의 각 record에 대해 ChooseSubtree가 호출되어,
record가 buffer에 남아있는지 아니면 N4의 buffer로 이동하는지 확인

항목 수가 가장 많은 버퍼만 완전히 지워지고 다른 buffer는
변경되지 X. 여기서 N4 buffer가 지워지므로 r24 insert
process reactivate

그 다음 Node의 entry가 N1에 게시.

r25 root buffer 삽입.

모든 rectangle에 대해 insert process 모두 완료했으나
전부 data node에 도착하지 않았으면 (= bulk loading 종료된 것 아님)
root에서 시작해 트리 위로 순서로 비어 있지 않은
buffer를 모두 비움.

// 최종 buffer tree

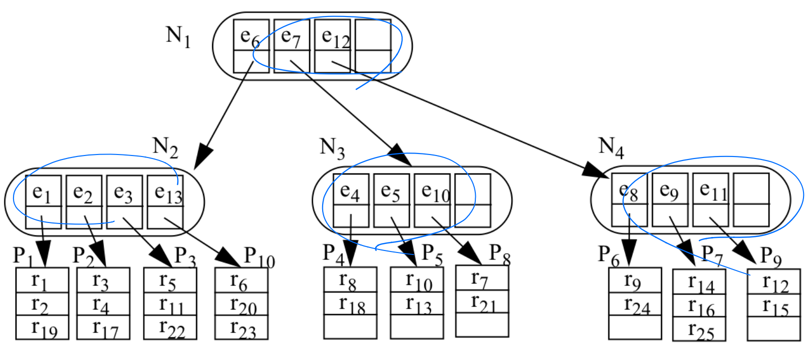


Figure 7: The example buffer-tree after having terminated all insertion processes

P1 ~ P10 = target R-tree의 data node

target R-tree의 index leaf node인 e1 ~ e5, e8, e9, e10, e11, e13을
삽입하여 2번째 buffer tree 생성.

= buffer tree의 leaf 위에 있는 target R-tree의
모든 index page 생성.

⇒ 이런 방식으로 target R-tree의 각 level을 추가 buffer tree로 구축됨

Bulk Loading Algorithm.

개념 : R-tree나 같은 index 종류에 insert algorithm 없이

insert : root → leaf 로 삽입

reconstruct : leaf → root 로 삽입

buffer tree의 unique 특징 : multiple insertion/reconstructing operation이 동시에 진행.

(Reconstructing : Data node의 overflow
Internal node의 routing table의 overflow) 이러한 trigger.

Reconstructing — overflow된 Node의 split.
— insertion of the new entry to the parent index node

record insertion process 진행.

internal index node로 들어온 entry insertion 작업을 먼저 한다.

Bulk Loading의 주요 algorithm : StartInsert.

Buffer tree는 처음에 빈 데이터 page에 대한 하나의 참조가 있는 index page로 구성되어 있다고 가정

StartInsert : Buffer tree의 root부터 new insert process 시작.

ALGORITHM *StartInsert* (*Root*, *R*)

(* An insertion process is started for record *R* in a buffer-tree with rootnode *Root*. *)

IF *BufferLoad*(*Root*) = B*p

new_children := *ClearBuffer*(*Root*);
new_siblings := *InsertChildren*(*Root*, *new_children*);
IF *new_siblings* is not empty

create a new root from *new_siblings*;

(* update *Root* *)

InsertIntoBuffer(*Root*, *R*);

END *StartInsert*.

① BufferLoad 호출 : Root buffer에 있는 record 수 계산.

같은 충족 X

→ InsertIntoBuffer 호출

이러한 routine : 일반 삽입 process 차단 & new record를 root buffer에 삽입.

저장공간 절약 위해. buffer : record를 보관하는데 필요한 최소 page 크기

- insert : buffer의 현재 마지막 page에 삽입.

이 페이지가 가득 차면 새 page가 buffer에 추가

② 같은 충족시. → Root buffer가 지워짐. reactivate insertion process

ClearBuffer 호출 결과로 Root의 새 자식 node를 참조하는 entry 목록인

new_children 얻음

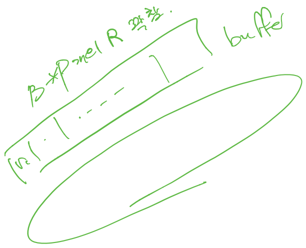
Insert Children : new_children 항목을 Root에 삽입.

↳ return 새 형제 목록.

empty가 아니면 new_siblings으로부터 new root 생성

```
ALGORITHM InsertChildren(Node, new_children)
(* The entries from children are inserted in Node. The algo-
   rithm returns a list of nodes (new_siblings) which are de-
   rived from Node through split operations. *)
  new_siblings := {};
  FOREACH entry E from new_children
    all_siblings := new_siblings ∪ {entry of Node};
    ParentNode := ChooseSubtree(all_siblings, E);
    InsertIntoNode(ParentNode, E);
    IF overflow in ParentNode
      Split(routing table of ParentNode);
      (* split routine of the index structure *)
      insert the new entry into new_siblings;
  IF new_siblings is not empty
    (* split the buffer of Node *)
    (* all_siblings = new_siblings ∪ {entry of Node} *)
    FOREACH record R from the buffer of Node
      Target := ChooseSubtree(all_siblings, R);
      move R from buffer of Node to the one of Target;
  RETURN new_siblings;
END InsertChildren.
```

Clearbuffer : Node의 overflow buffer를 지우는 buffer tree에서 가장 중요한 알고리즘.



internal index node에선 → ClearInternalBuffer

index의 leaf node 쪽은 → ClearLeafBuffer.

```
ALGORITHM ClearInternalBuffer(Node)
(* Clears the buffer of Node whose entries refer to an index
   node. *)
   overflow_list := {};
   FOREACH R of the first B*p records in buffer of Node
      (* reactivate the insertion process of R *)
      Child := ChooseSubtree(Node, R);
      InsertIntoBuffer(Child, R);
      IF BufferLoad(Child) > B*p
         insert Child into overflow_list;
   new_children := {};
   FOREACH Child from overflow_list
      add ClearBuffer(Child) to new_children;
   new_siblings := InsertChildren(Node, new_children);
   RETURN new_siblings;
END ClearInternalBuffer.
```

Buffer가 꽉차서 record를 아래로 내려야 할때, 내릴 subtree 찾음, child에 반환

buffer에서 첫번째 B*p record를 읽고 레코드가 할당되어야 하는 노드결정.

R을 child buffer 마지막 page에 insert

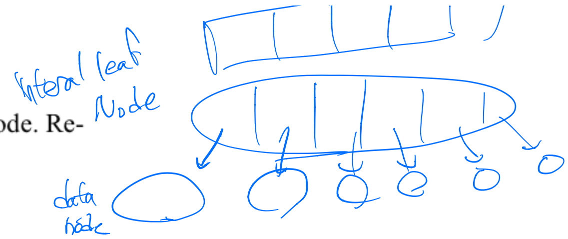
Child에 넣었는데 거기서도 overflow 발생한 경우, overflow_list에 Child 정보를 대입.

overflow_list는 Node를 기준으로 Node의 child 중 (R넣어서) overflow 발생한 child들 목록.

해당 overflow_list에 대해 ClearBuffer 재귀적으로 다시 사용.

그리고 최종적으로 new_siblings에 있는 목록을 Node에 삽입

이 함수의 중요한 점은 버퍼가 첫 번째 B*p 레코드에서 부분적으로 지워진다는 것.




```
ALGORITHM ClearLeafBuffer(Node)
(* Clear buffer of Node which is an internal leaf node. Re-
  turns a list of entries of new siblings of Node. *)
  new_siblings := {};
  FOREACH record R in buffer of Node
  (* reactivate the insertion process of R *)
    DataNode := ChooseSubtree(Node, R);
    InsertIntoNode(DataNode, R);
    (* insertion process is terminated *)
    IF overflow in DataNode
      Split(DataNode);
      (* split routine of the index structure *)
      apply corresponding entries to Node;
      (* immediately *)
      IF overflow in Node
        split Node into two (let N be the new node);
        (* split the routing table and the buffer *)
        insert the entry of N into new_siblings;
        update entry of Node;
        IF BufferLoad(N) > BufferLoad(Node)
          add ClearLeafBuffer(N) to new_siblings;
        ELSE
          add ClearLeafBuffer(Node) to new_siblings;
        RETURN new_siblings; (* exit of algorithm *)
  RETURN new_siblings; (* is empty *)
END ClearLeafBuffer.
```

→ R에 대해 Choose Subtree해서 Data Node Return

→ 받은 DataNode가 overflow

→ 분할.

∴ 분할되면 해당 data Node 참조하는 entry 생성되고, 이것을 Node의 routing table에 삽입할 때 이미 C개의 entry가 있으면 routing table도 분할

그리고 새로운 Node 가리키는 index entry가 new_siblings에 insert.

모든 Record에 대해 StartInsert 후, Node의 buffer는 깊이 우선 순서로 지워짐