An Evaluation of Generic Bulk Loading Techniques*

Jochen van den Bercken
IXOS SOFTWARE AG
Grasbrunn/Germany
jochen.van.den.bercken@ixos.de

Bernhard Seeger
Fachbereich Mathematik und Informatik
Philipps-Universität Marburg
seeger@mathematik.uni-marburg.de

Abstract

Bulk loading refers to the process of creating an index from scratch for a given data set. This problem is well understood for B-trees, but so far, non-traditional index structures received modest attention. We are particularly interested in fast generic bulk loading techniques whose implementations only employ a small interface that is satisfied by a broad class of index structures. Generic techniques are very attractive to extensible database systems since different user-implemented index structures implementing that small interface can be bulk-loaded without any modification of the generic code.

The main contribution of the paper is the proposal of two new generic and conceptually simple bulk loading algorithms. These algorithms recursively partition the input by using a main-memory index of the same type as the target index to be build. In contrast to previous generic bulk loading algorithms, the implementation of our new algorithms turns out to be much easier. Another advantage is that our new algorithms possess fewer parameters whose settings have to be taken into consideration.

An experimental performance comparison is presented where different bulk loading algorithms are investigated in a system-like scenario. Our experiments are unique in the sense that we examine the same code for different index structures (R-tree and Slim-tree). The results consistently indicate that our new algorithms outperform asymptotically worst-case optimal competitors. Moreover, the search quality of the target index will be better when our new bulk loading algorithms are used.

1 Introduction

Recently, there has been an increasing interest in designing methods for processing a set of homogenous operations on an index in bulk. Among the different bulk operations, bulk loading of an index has attracted most of the research attention. In this paper, we address the problem of bulk loading an index for a given data set as fast as possible. We are primarily interested in creating indexes from non-traditional index structures which are suitable for managing multidimensional data, spatial data or metric data. For these kinds of data, it is in general not advisable or even not possible to apply classical *sort-based bulk loading* where first, the data set is sorted and second, the tree is built in a bottom-up fashion. In the context of non-traditional index structures, the method of bulk loading also has a serious impact on the search quality of the index. We therefore aim for a bulk loading method that gives comparable or better search performance than *tuple-loading* where an index is built up by inserting tuples one by one.

In addition to sort-based methods, there has been another two broad classes of bulk loading techniques. The one class called *Buffer-based bulk loading* employs the buffer-tree technique [Arg 95] which can generally be used to preserve efficiency of main-memory algorithms when the data does not fit into memory anymore. Due to this technique, bulk loading can be performed as fast as external sorting (in an asymptotic sense) on index structures which support an insertion of a record in logarithmic time. The second class called *sample-based bulk loading* employs a sample that fits into memory to build up the target index. Although bulk loading is of interest for any kind of index structure, most of the available methods were presented in the context of R-trees or closely related index structures. Little attention has been given to other index structures although it might be important from the perspective of an extensible database system to provide a generic bulk loading method which will be applicable not only to many different index structures currently available but also to index structures that will be developed in the future.

In this paper, we present two new sample-based bulk loading techniques which are applicable to a broad class of tree-based index structures. Both techniques are generic in the sense that they make use of an interface that is satisfied by a broad class of index structures. While the one

*This work has been supported by grant no. SE 553/2-1 from DFG.

Proceedings of the 27th VLDB Conference
Roma, Italy, 2001

approach is applicable to many tree-based index structures like B+-trees, R-trees [Gut 84], M-trees [CPZ 97] and S-trees [Dep 86], the other technique can be used for bulk loading almost any kind of tree-based index structure that uses the concept of node splitting, including KDB-trees [Rob 81] and their relatives hB-trees [LS 90] and LSD-trees [HSW 89]. The new techniques are conceptually simpler than most of the previous ones and therefore, their implementation is not too difficult. We implemented our new methods as well as the most promising competing methods which have been previously proposed. Our experiments indicate that our new methods perform fast in most cases, but, as known from other algorithms based on sampling, their performance is also influenced by the quality of the sample. Moreover, the search performance of the trees generated by our sampling-based methods is generally better in comparison to others.

Our performance comparison is unique in the sense that all index structures and bulk loading techniques are implemented by using the same building blocks that are part of our fully documented and public available library XXL [BBD+ 01], [BDS 00]. The library provides a powerful and flexible infrastructure for implementing query processing functionality suitable for comparing the runtime of different algorithms. Though the absolute timings of the bulk loading methods are of minor relevance, we believe that the relative timings are excellent indicators for a performance comparison of the different bulk loading methods.

The paper is organized in the following way. In Section 2, our most important notation is introduced. Moreover, we give a review of the current research results on bulk loading. In Section 3, we present Path-based bulk loading, a new bulk loading algorithm applicable to the broad class of Grow&Post-trees. Section 4 is dedicated to Quickload, a special algorithm for bulk loading index structures from an important subset of Grow&Post-trees. The results from a preliminary performance comparison are reported in Section 5. Section 6 concludes the paper.

2 Preliminaries

In this section, we introduce the basic assumptions which are required from the index structures in order to use our generic bulk loading techniques. Moreover, we also present the underlying cost model. At the end, we give a brief review of the bulk loading techniques.

2.1 Grow&Post-Trees

In the following, we assume an index structure to be a tree. Associated to each node of the tree is a *reference* that, except for the reference of the root, is stored in the parent node. Data records are stored in the *data nodes* which correspond to the leaves of the tree, whereas the *index nodes* are internal nodes.

Most tree-based index structures show great similarities in their internal interface. In common with [Lom 91] we assume that a tree is a Grow&Post-tree (*GP-tree*) where the following operations are supported:

- *chooseSubtree*: Given a data record and an index node. Compute the reference to the subtree where an insert operation of the record should be forwarded to.
- *grow*: Given a data record and a data node. Insert the record into the node.
- *split&post*: Given an overflown node. Split the node into two and post the information about the split (e.g. the new reference) to the parent node.
- *search*: Given a query and data node (index node). Return all data records (references) stored in that node being relevant to the query.

An insertion of a new record into a GP-tree is performed in the following way. First *chooseSubtree* is iteratively called starting at the root of the tree until a data node is retrieved. Then, the data record is inserted into the data node, i.e, the node *grows* by one record. When this causes an overflow, the node is first *split* into two and then, the relevant information about the split is *posted* to the parent node. If the root node is split the tree grows by one level. The most prominent example of a GP-tree is the B+-tree which supports one-dimensional queries. However, many of the well-known multidimensional index structures [GG 98] like the KDB-tree [Rob 81], hB-tree [LS 90], R-tree [Gut 84], MVB-tree [BGO+ 96], M-tree [CPZ 97] and Slim-tree [TTSF 00] also satisfy the interface of the GP-tree.

An important subclass of GP-trees is the class of overlapping-predicate-trees (*OP-trees*). In addition to the functionality of a GP-tree, an OP-tree also supports insertions of entire trees whose height is lower than the height of the target tree. An insertion of a tree is generally performed by inserting the reference of its root node into an appropriate node of the target index whose level is equal to the height of the tree to be inserted. If the root of the inserted tree is not sufficiently full, an additional merge has to be performed with a sibling node. This additional functionality is possible due to the property that node predicates in an OP-tree may overlap where the *node predicate* of an index node is satisfied by all records in the corresponding subtree. For the R-tree, the node predicate is represented as a minimum rectilinear rectangle that covers the records in the subtree. Since the predicates of an OP-tree may overlap, it is possible to design specific bulk loading algorithms that create an index structure level by level bottom-up. In addition to the R-tree, the M-tree, S-tree and Slim-tree belong to this important class of index structures.

2.2 I/O Model

We assume that a disk is partitioned into pages of fixed size, with random access to each page at unit cost. Our goal is to build up an index on disk and therefore, a node of the tree also corresponds to a disk page where at most B records (data objects) can be stored. Each access to disk transfers one page; we denote this as one I/O. Our generic bulk loading algorithms simply use the available implementations of the GP-tree interface. These implementations generally determines the CPU-cost of bulk loading GP-trees and therefore, we are primarily interested in reducing the I/O-cost. As a consequence, the performance of our algorithms is measured in the number of I/Os required for performing a sequence of N insertions. In particular, we are not interested in the I/O-cost of a single insertion.

An important parameter of bulk loading algorithms is the amount of available main memory that can be used. In the following, we assume that main memory is managed by a database buffer that follows the LRU replacement strategy. Let M be the maximum number of records that fit into the available main memory. 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.

For the I/O model described above 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. Special care concerning this trade-off has to be taken for OP-trees. They may be built naively level by level bottom-up by packing the data records into nodes without performing any kind of preprocessing (e.g. sorting). This bulk loading approach requires $\Theta(n)$ I/Os only, but the search performance of such an index will be unacceptable, in general. Thus, although there exists a linear-time bulk loading algorithm for OP-trees, we are primarily interested in bulk loading algorithms whose I/O complexity meets the lower bound of external sorting.

2.3 Review of Previous Techniques

In the following, we give a brief overview of the different approaches to bulk loading and discuss their unique properties. In order to compare these methods, different aspects have to be taken into account. Some of the methods require that the data source is entirely on disk, whereas other methods may also accept a source directly delivered from an iterator [Gra 93]. Some of the methods are memory-adaptive, whereas others require a fixed amount of memory during the entire runtime. There are also methods which are not limited to bulk loading only, but also support bulk insertions. Another issue is the primary design goal of the methods: should be the search quality of the target index or the build time of the index most important? We primarily distinguish the different bulk loading methods with respect to their main internal techniques (*sorting*, *buffering*, *sampling*) into three different classes.

Sort-based bulk loading is a well established technique since it is used in commercial database systems for creating B+-trees from scratch. Bulk loading of a B+-tree first sorts the data and then builds the index in a bottom-up fashion. For each level of the index, the nodes can be packed entirely full, except for the right most node. It is however advisable to leave some empty space in the nodes, when further insertions are expected right away after bulk loading. The runtime of this approach is dominated by the cost of sorting which requires $O(n \log_m n)$ I/Os. Assuming that an appropriate ordering exists, sort-based bulk loading is not limited to one-dimensional index structures, but can also be applied to OP-trees, since OP-trees support insertions of entire trees. For multidimensional index structures like R-trees, the question arises what kind of ordering results in the tree with best search performance. One of the first approaches [RL 85] suggests to sort the data with respect to the minimum value of the objects in a certain dimension, whereas [KF 93] suggest to order w.r.t. the Hilbert-value of the centers. It is also shown in experiments [KF 93] using spatial data that the Hilbert-ordering gives better performance. Other experiments [DKL+ 94] revealed that the search performance of the R-trees built by using Hilbert-ordering is inferior to the search performance of the R*-tree [BKSS 90] when the records are inserted one by one. In a data warehouse environment where the dimensions are quite different (and hence it may be difficult to come up with a well-defined Hilbert-value) it might still be better to select a dimension and to sort the data according to this dimension [KR 98]. The primary reason for using this approach in a data warehouse is the fact that primarily bulk insertions should efficiently be supported. Several methods are available for supporting bulk insertions on indexes which rely on a linear ordering of the data ([JDO 99], [JNS+ 97], [KR 98], [MNPW 00]).

Another sort-based method for bulk loading R-trees is presented in [LEL 97]. The methods starts sorting the data source w.r.t. the first dimension (e.g. using the center of the spatial objects). Then, $(N/B)^{1/d}$ contiguous partitions are generated, each of them containing (almost) the same number of objects. In the next step, each partition is sorted individually w.r.t. the next dimension. Again, partitions are generated of almost equal size and the process is repeated until each dimension has been treated. The final partitions will eventually contain at most B objects. In [LEL 97] it was shown that this method of sort-based bulk loading creates R-trees whose search quality is superior to those R-trees which have been created w.r.t. the Hilbert-ordering. However, the method also requires the input being sorted d times.

Quite a different approach to bulk loading is based on sampling. The bulk loading method of the M-tree [CP 98], for example, follows this idea. The method randomly samples objects, we call them representatives, from the input and builds up a structure also known as a seeded-tree [LR 98]. Then, the remaining records of the input are assigned to one of the representatives. For each representative, the associated data objects are treated again in the same way. The result of this approach is basically a M-tree of M-trees and hence, the structure offers some structural properties (underfilled nodes, unbalanced structure) that violates the invariants of the original M-tree [CPZ 97]. The authors discuss different strategies to obtain the desired structural behavior. For example, second-level trees with a small number of objects are deleted and their objects are assigned to other representatives. This even may result in a structure where only one representative exists. In this case, [CP 98] suggests to start again with a different sample.

Note that the M-tree and related structures like the Slim-tree [TTSF 00] deals with metric data and therefore, there is no natural ordering of the data. Consequently, a sort-based bulk loading cannot directly be applied to bulk loading an M-tree. An alternative might be to map the metric data into vector data using for example fast map [FL 95] and then applying again sort-based bulk loading. However, the quality of the M-tree might substantially suffer from such an approach.

Another approach based on sampling is given in [BBK 98] where a kd-tree structure is built up using a fast external algorithm for computing the median (or a point within an interval centered at the median). The sample is basically used for computing the skeleton of a kd-tree that is kept as an index in an internal node of the index structure as it is known from the X-tree [BKK 96]. The method however relies on a recursive partitioning of the data set into two as it is known from Quicksort. This results in a large I/O overhead since the data set has to be read and written quite often. However, sequential I/Os can be used in order to reduce the total I/O cost.

Buffer-based bulk loading algorithms as presented in [BSW 97] and [AHVV 99] are completely different from those algorithms described above. The methods employ external queues (so-called buffers) that are attached to the internal nodes of the tree except for the root node. An insertion of a record can be viewed as a process that is temporarily blocked after having arrived at a node. Instead of continuing the traversal down to the leaf, the record is inserted into the buffer. Whenever the number of records in a buffer exceeds a pre-defined threshold, a large portion of the records of the buffer is transferred (via individual calls of *chooseSubtree*) to the next level. The bulk loading method of [BSW 97] builds up the tree level by level and is therefore restricted to OP-trees, whereas the method of [AHVV 99] can also be used for GP-trees. The number of I/Os for both methods is $O(n \log_m n)$ which is asymptotically equal to the lower bound of external sorting. An advantage of the method [AHVV 99] is that it can easily be extended to support other types of operations, e.g. insertion, in bulk. A disadvantage of [AHVV 99] is however that from the available m main memory pages only $B^{\lfloor \log_B m \rfloor}$ are actually used. In other words, the algorithms only uses m/B main memory pages in the worst case. The obvious question we address in our experiments is how the performance of bulk loading is influenced by the bad memory usage. Moreover, the implementation complexity of Buffer-based bulk loading is high, particularly, when the implementation is not tightly coupled to a specific index structure.

A slightly modified version of the buffer-based methods is presented in [JDO 99] for B+-trees. The unique feature of the method is that the buffers are not emptied in a final phase, but remain attached to the nodes. Since the buffers are organized in a special way, the index (with its buffers) also supports processing of on-line queries. Genericity is however lost for that method since it is tightly coupled with B+-trees.

3 Bulk Loading GP-trees

In this section we first present a new generic algorithm for bulk loading GP-trees. Though the algorithm does not meet the worst-case bound of external sorting, we expect an excellent average-case performance. As an advantage, the algorithm is conceptually much simpler than its worst-case optimal competitors.

3.1 Path-based Bulk Loading

Path-based bulk loading can be viewed as a top-down algorithm where the data is partitioned in a recursive fashion until the partition fits in memory. Path-based bulk loading is applicable to any kind of GP-trees including hB-trees [LS 90] where bulk loading has been considered as an open problem to the best of the authors knowledge.

Bulk loading an index is generally easy when the entire input fits in memory. The desired index is first built in memory and then transferred to disk. If memory is too small, we start building an in-memory index by inserting records from a sample until the available memory is filled up. Next, we associate to each leaf a bucket on disk. The remaining records of the input are then assigned to the buckets by calling *chooseSubtree* repeatedly until a leaf has been reached. Note that nodes are not split during this phase of the algorithm. In case of OP-trees, however, it is still necessary to update the routing information of the references. When all the data records have been processed, the nodes in main memory are written to disk. Moreover, the pairs of non-empty buckets and the references to their corresponding leaf nodes are written in a to-do-list on secondary storage. The algorithm is illustrated in Figure 1 assuming that main memory consists of 4 pages. On the left hand side the situation is depicted where the index is built up in memory, but where data records still have to be processed. The right hand side refers to the situation after having processed the entire data set.

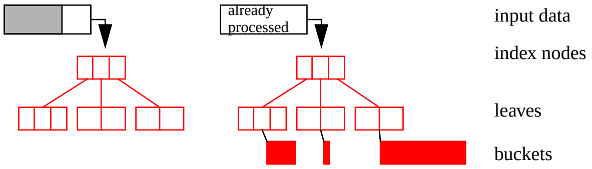

FIGURE 1: Example of Path-based bulk loading for m=4

In the next phase, a pair is taken from the to-do-list. The corresponding bucket is considered as the source and the records from the bucket are processed as described above in a recursive fashion. For GP-trees which are not OP-trees, e.g. kdB-trees, the insertions of these records will be on a single path of the index that has been created previously. Therefore, many insertions can be performed until the available main memory is fully utilized again. The algorithm for this kind of tree is given as pseudo code below in Algorithm 1.

For OP-trees, this approach seems to be not efficient since the insertions of records from a single bucket are not restricted to a single path. In order to improve efficiency, the algorithm is modified such that insertions are limited to those nodes which are newly created or belong to the path from the root to the corresponding leaf.

Another technical feature of the modified version is that insertions do not start at the original root, but at a pseudo-root which is firstly set to the corresponding leaf. Whenever the pseudo-root is split, the parent node of the pseudo-root is set to be the new pseudo-root. In order to keep the algorithmic description simple we did not include these issues in Algorithm 1.

Algorithm 1: PathBasedBulkLoading

```
PathBasedBulkLoading (Tree tree, Iterator source, int maxNodes) {
    let toDoList be a (external) queue;
    loop{
        insertObjects(tree, source, maxNodes, toDoList);
        if toDoList is empty
            return;
        remove the next pair (reference, bucket) from toDoList;
        perform a query using an object stored in bucket to find the leaf
            referenced by reference;
        set source to an iterator on bucket;
    }
}
void insertObjects (Tree tree, Iterator source, int maxNodes,
    Queue toDoList) {
    while source isn’t empty and the number of nodes in memory is less
        than maxNodes {
        take the next object from source;
        insert object into tree;
    }
    flush all leaves from main memory to disk;
    while source isn’t empty {
        take the next object from source;
        insertIntoBucket(tree, object);
    }
    foreach reference stored in index nodes residing in main memory
        and pointing to a leaf {
        let bucket be the buffer associated with reference;
        if bucket isn’t empty
            insert the pair (reference, bucket) into toDoList;
    }
    flush all index nodes from main memory to disk;
}
void insertIntoBucket (Tree tree, Object object) {
    let reference be the reference pointing to the root of tree;
    while reference doesn’t reference a leaf node {
        let node be the node referenced by reference;
        set reference to the result of applying chooseSubtree to node
            using object;
    }
    insert object into the bucket associated with reference;
}
```

On the average, the method performs fast in practice as it is shown by our experiments. The performance of the algorithm will be excellent as long as the distribution of the input among the buckets is uniform. In order to verify this statement, let us assume that the *N* input records are equally distributed among the *m* buckets such that each bucket receives *N/m* records. It follows that the total number of I/Os is $\theta(n \cdot \log_m n)$. Hence, the total cost meets the lower bound of external sorting. It is obvious that such a perfect distribution of records will seldom occur in practice. However, the central limit theorem tells us that the occupation of a bucket is close to the mean value *N/m*.

The worst case arises when only one of the buckets receives records and the other buckets remain empty. The runtime of the algorithm then degenerates to $\theta(n^2/m)$. When applying Path-based bulk loading to one-dimensional index structures (B+-trees), the worst case occurs e.g. for sorted input. For multidimensional structures, it is more difficult to come up with a data distribution where the performance is poor.

In comparison to worst-case optimal methods, there are a few other advantages with respect to memory management. First, the method completely employs the available main memory, whereas the worst-case optimal method presented in [AHVV 99] might use only a small fraction. In general, we expect that runtime of the methods will improve when more memory can be used. Moreover, it might also be possible to adapt the amount of memory during runtime. The memory in use (except the memory for buffering a path) can be determined for each call of *insertObjects* individually.

4 Bulk Loading OP-trees

OP-trees represent an important class of index structures including R-trees [Gut 84], S-trees [Dep 86], M-trees [CPZ 97] and others. A reference of an OP-tree consists of a predicate, say *P*, and a pointer to a subtree where each of the records in the subtree satisfy *P*. Predicates within an index may overlap, i.e., a new record may satisfy more than one predicate (or none of the predicates). As a consequence, OP-trees efficiently support insertions of entire trees whose height is lower than the height of the target tree. This unique property also gives more freedom for the design of bulk loading algorithms in comparison to more general GP-trees. An extreme approach would be to pack the incoming data records into pages without any preprocessing, one by one, and to build up the tree bottom-up. This algorithms causes the lowest cost O(*n*) for bulk loading an index, but the index would not support queries efficiently.

4.1 Quickload

In the following, we present a generic algorithm called *Quickload* for bulk loading any kind of OP-trees, including R-trees and M-trees. Quickload is conceptually close to the bulk loading algorithm of the M-tree [CP 98], but Quickload completely overcomes its serious deficiencies. The basic idea of Quickload is as follows: a sample is first taken to partition input data. In order to improve I/O efficiency, we do not partition the data set into only two as it was proposed in [BBK 98], but in a large number of partitions. Quickload is then applied to each partition in a *recursive* fashion. The size of the sample is chosen as large as the available main memory. An OP-tree (of the same type as the target index) is used for organizing the sample in memory. The only difference to an external index is that the size of the internal nodes might be set differently in order to improve CPU performance, whereas the size of the leaves still corresponds to a page on disk.

If the entire input fits in memory, the leaves of the in-memory structure correspond to leaves of the target index. Otherwise, records are inserted into the tree until memory is filled up. Thereafter, buckets are attached to the leaves. An insertion is then not guided anymore to a leaf, but to the corresponding bucket. Besides updating the routing information being part of the references, the structure will not be changed while the records are distributed among the buckets. When all records of the input are treated, we distinguish two cases. If a bucket is empty, the reference to the corresponding leaf is inserted into a file on disk. The references contained in this file will be used later during a subsequent pass of the algorithm to build the next upper level of the tree. Otherwise (the bucket contains records), a pair consisting of the reference to the corresponding leaf and a reference to the bucket is inserted into a to-do list. The algorithm is then applied recursively to the elements in the to-do list. When the to-do list becomes empty the file on disk contains exactly the references to the leaves of the target index. These references now serve as the input for Quickload in order to create the next upper level of the target index. If there is only one reference left (pointing to the root of the target tree), the algorithm stops. Note that the recursive processing is the reason why Quickload is limited to OP-trees.

In Figure 2, we illustrate the important aspects of the algorithm where an OP-tree is built from the input records $R_1, \ldots, R_{13}$. We assume that at most three leaves fit in memory. On the left hand side, the situation is depicted when memory is fully utilized, but some of the input records are still unprocessed. Note that S_1, S_2 and S_3 refer to predicates which cover the records in the corresponding leaves. On the right hand side of Figure 2 the situation is illustrated after the remaining records are inserted into the corresponding buckets. The leaf which belongs to S_2 is already a leaf of the target index because its bucket is empty, whereas Quickload has to be applied once again to the other leaves assuming that the input is taken from the corresponding buckets. The example for processing the leaf and the bucket which belong to S_1^* is illustrated in Figure 3 which results in two leaf pages that will be part of the target index.

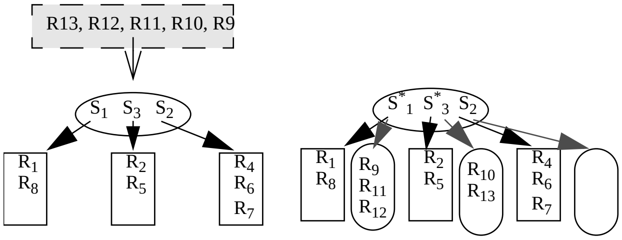

FIGURE 2: An example of Quickload when at most three leaves fit in main memory

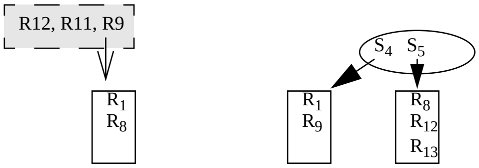

FIGURE 3: Recursive step of Quickload

A detailed description of the algorithm is given in Algorithm 2.

Algorithm 2: Quickload

```
Quickload (OPTree tree, Iterator source, int maxNodes) {
    let levels be 1;
    loop {
        let nextLevelReferences be the queue returned by
            createLevel(tree, source, maxNodes);
        if nextLevelReferences contains exactly one reference {
            set the height of tree to levels;
            return;
        }
        set tree to an empty tree;
        increase levels by one;
        set source to an iterator on nextLevelReferences;
    }
}
Queue createLevel (OPTree tree, Iterator source, int maxNodes) {
    let toDoList be an (external) queue;
    let nextLevelReferences be an (external) queue;
    loop {
        insertObjects(tree, source, maxNodes, toDoList,
            nextLevelReferences);
        if toDoList is empty
            return nextLevelReferences;
        remove the next tupel (reference, bucket) from toDoList;
        set the root of tree to the node referenced by reference;
        set source to an iterator on bucket;
    }
}
void insertObjects (OPTree tree, Iterator source, int maxNodes,
    Queue toDoList, Queue nextLevelReferences) {
    while source isn't empty and the number of tree's leaves is less than
        maxNodes {
        take the next object from source;
        insert object into tree;
    }
    flush all leaves from main memory to disk;
    while source isn't empty {
        take the next object from source;
        insertIntoBucket(tree, object);
    }
    foreach reference of tree pointing to a leaf {
        let bucket be the bucket associated with references;
        if bucket is empty
            insert reference into nextLevelReferences;
        else
            insert the tupel (reference, bucket) into toDoList;
    }
    delete each internal node of tree residing in main memory;
}
void insertIntoBucket (OPTree tree, Object object) {
    let reference be the reference pointing to the root of tree;
    while reference doesn't reference a leaf {
        let node be the node referenced by reference;
        set reference to the result of applying chooseSubtree to node for
            object;
        insert object into the bucket associated with reference;
    }
}
```

Similar to Path-based Bulk Loading, it can be shown easily that the total number of I/Os is $\theta(n \cdot \log_m n)$ on the average. The worst-case of Quickload arises when only one bucket receives the records and the other buckets remain empty. Then, the runtime of the algorithm is $\theta(n^2/m)$. Quickload can be also adaptive when the size of the available memory will change during runtime. Whenever an element is taken from the to-do list, the algorithm can reallocate its memory.

5 Experiments

In this section we report experimental results for bulk loading R*-trees [BKSS 90] and Slim-trees [TTSF 00] by using the different bulk loading approaches. Both index structures belong to the family of OP-trees. Therefore, we were interested not only in the cost of bulk loading an index but also in its search performance.

We also performed several experiments for B+-trees. Here, the classical approach of sorting the data and packing the nodes of the tree is the clear winner. This result even holds for packing the nodes just to 2/3 of their maximum capacity so that search performance is comparable to the other bulk loading algorithms.

The cost of bulk loading an index was measured in milliseconds of elapsed time. This gives a more realistic impression than just counting I/O- or CPU-cost. Please keep in mind that we are not interested in the absolute amounts of time needed. In order to compare the search performance of the created indexes, we treated every 10th record inserted into the tree as a query. Following the results of [LL 98], we measured query performance by counting the number of pages that were read and written to disk.

Throughout the experiments, we used data sets from the TIGER files [Bur 96] containing rectangles reflecting the borders of environmental objects. While these real world data sets are predestined to be used for bulk loading R*-trees, we mapped the rectangles to a four dimensional data space to be used in the context of Slim-trees. We will present the results for the *CAL_HYDRO* data set (360.000 hydrographical features from California), the *TEX_HYDRO* data set (360.000 hydrographical features from Texas) and the *EAST_RAIL* data set (360.000 railroad items from the east of the US). However, for many bulk loading algorithms the cost of bulk loading as well as the query performance is highly influenced by the order in which the objects are processed. To demonstrate this property, we once sorted the data sets according to a Hilbert space filling curve and once created a random permutation. In the following, we will refer to a sorted data set by appending the suffix *sorted* to its name and to a permuted one by appending *shuffled*.

Though we characterized Path-based bulk loading and Quickload to be based on sampling, our implementations are restricted in that the sample always consists of the first objects delivered by the data source. In our experiments we show that even for highly clustered data both algorithms perform fairly well. Another point is that sampling requires the data set to be materialized. However, in case of data delivered by an iterator [Gra 93] this would require to store the data first before bulk loading can be applied.

Due to its greater flexibility, we decided to prefer the Buffer-based algorithm according to [AHVV 99] to the one presented in [BSW 97]. This decision influenced the different settings for the amount of main memory provided for the bulk loading algorithms.

5.1 Implementation Details

All experiments were performed on a PC running under Windows NT 4.0 with an Intel Pentium III-Processor clocked at 500 Mhz and 256 MBytes of main memory. The I/O-device was a Maxtor Diamond Plus 6800 using DMA.

The main memory available to the bulk loading algorithms was organized as a database buffer of *m* pages using the LRU strategy. The size of the disk pages was set to 2 Kbytes.

We tried to achieve a fair comparison of the different bulk loading approaches by using the same set of classes from our library XXL for all implementations. The classes representing index structures were derived from generic classes modeling GP- and OP-trees supporting the interface given in chapter 2.1. The generic bulk loading algorithms were developed independently from special tree classes and do not use specific properties of the given index structure to be bulk loaded. As a consequence, it is very easy to apply the same generic algorithm to different index structures derived from the generic tree classes.

5.2 Bulk Loading R*-trees

Before presenting the comparative results of different algorithms, we examine the impact of parameters on the runtime of the individual algorithms. We are interested in those parameter settings where the created index provides fast bulk loading on the one hand and good search performance on the other hand. We performed our experiments with each of the three TIGER data sets. However, we observed very similar behavior of the algorithms according to the data set.

5.2.1 Parameter Settings

Let us first consider Quickload. Here, the question arises how to choose the fanout of the internal nodes of the in-memory trees that are used to partition the data. We performed experiments for two settings of the fanout. First, we set the fanout in such a way that the size of the nodes corresponds to 2 KBytes which is also the size of the leaf nodes. Second, the fanout was set to 5. As one can expect, a smaller value may reduce the CPU-cost, because the algorithms *chooseSubtree* and *split&post* have linear and super-linear runtime, respectively. However, a small fanout decreases the possibilities for placing new records in the tree which may also reduce search performance. In our experiments, we observed exactly this behavior, but the performance difference of the two settings was less than 1% in most cases. For sake of simplicity, we decided to use a node size of 2 KBytes.

In case of the Buffer-based approach, one has to determine the buffer capacity *p*. In our experiments, we observed a value of *m*/2 as an optimal setting for *p* regarding the bulk loading time one the one hand and query performance on the other.

Another parameter is also the amount of data to be cleared from an overflown buffer. We distinguished two strategies, a pessimistic and an optimistic one. The pessimistic strategy, as described in [BSW 97] and [AHVV 99], processes the records of *p* pages of an overflown buffer and propagates the records one level down. Even in case that all of those records are directed to just one of the child buffers, this buffer will not contain more than 2*p* pages afterwards. However, it is more likely that the records are distributed more uniformly among the child buffers. Thus, in most cases it is possible to process more than *p* pages of a overflown buffer at once. This optimistic approach however requires to stop clearing the buffer if a child buffer contains 2*p* pages. Note that both strategies result in a worst-case optimal bulk loading algorithm.

We figured out in our experiments that, especially when processing uniformly distributed data, the optimistic strategy behaves superior to the pessimistic one. With respect to the search quality of the generated indexes, there was no difference between the two strategies. Therefore, we used the optimistic one throughout our experiments.

5.2.2 Comparing the Algorithms

In the following, we compare the different bulk loading approaches for R*-trees. In addition to the generic approaches, we also investigated two approaches based on sorting rectangles w.r.t. Hilbert values. *Sort-based* bulk loading [KF 93] refers to the classical approach of sorting and packing the nodes of the R*-tree. The other approach, which we call *Sorted-Tuples-based* bulk loading, is even simpler. The data records are first sorted and then inserted into the tree one by one. Correspondingly, *Tuple-based* bulk loading simply inserts the records into the tree without any kind of preprocessing.

As depicted in Figure 4, each of the bulk loading algorithms performs significantly better than the Tuple-based approach. The ranking of the bulk loading algorithms shown here is also quite typical for other experiments. The well-known Sort-based approach is the clear winner, which is more than three times faster than the second best approach (Quickload). However, when we consider search performance, Sort-based bulk loading shows major deficiencies, see Figure 5. Among the generic algorithms, Buffer-based bulk loading requires most time to create an index, while Quickload is faster by a factor of 1.5. Path-based bulk loading performs just in between, similar to the Sorted-Tuples-based approach. Figure 5 shows the search performance for the same settings. The price that has to be paid for using Sort-based bulk loading is its rather poor search performance. Interestingly, in case of the CAL_HYDRO_shuffled data set, both Quickload and Path-based bulk loading create trees that show a better search performance than those obtained by just inserting the records one by one. This observation could also be made for many other parameter settings.

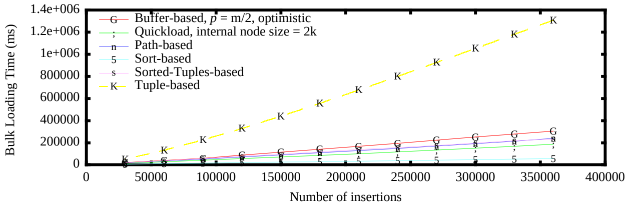

FIGURE 4: EAST_RAIL_shuffled, m=2*fanout

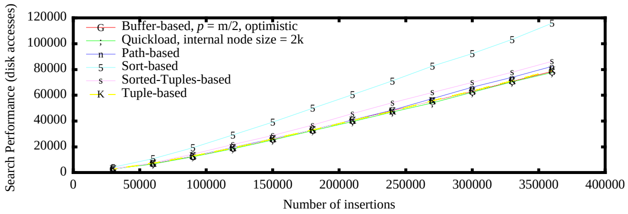

FIGURE 5: EAST_RAIL_shuffled, m=2*fanout

In case of clustered data sets like the TEX_HYDRO data set (Figure 6), results changed quite dramatically.

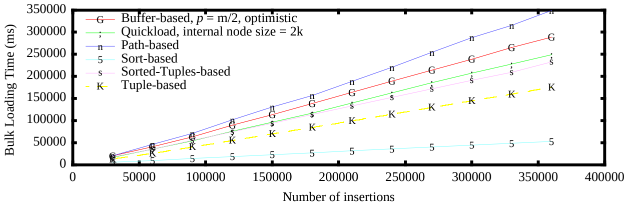

FIGURE 6: TEX_HYDRO, m=2*fanout

Once again, the Sort-based approach is the clear winner, but the Tuple-based approach comes next. The original data sets show similar behavior to the sorted ones, resulting in high locality when records are inserted. Because the main memory available to the algorithms is organized as a LRU buffer, most of the node accesses do not cause disk accesses. Furthermore, the Tuple-based approach does not need any further processing, which results in a better performance than the remaining algorithms. Though the data is almost sorted, Quickload performs faster than Buffer-based bulk loading, while Path-based bulk loading is influenced more strongly by the fact that the sample quality is poor. Once again, the TEX_HYDRO data set results in trees that are very similar according to their search performance.

In the last set of experiments with R*-trees, we were interested in memory utilization of the different approaches. In Figure 7, we varied the amount of memory *m* from 2*fanout to fanout^2/2. While both Quickload and Path-based bulk loading benefit from this additional amount of memory, Buffer-based bulk loading shows only minor decreases in the index construction time. The reason is that Quickload and Path-based bulk loading are able to use the entire memory very effectively, while in case of Buffer-based bulk loading the fanout of the tree determines the usage of memory. For *m* = fanout^2/2, every level of the tree still has to be equipped with buffers. Only when *m* is greater than fanout^2, Buffer-based bulk loading is able to exploit more memory since buffers will only occur on every second level of the tree.

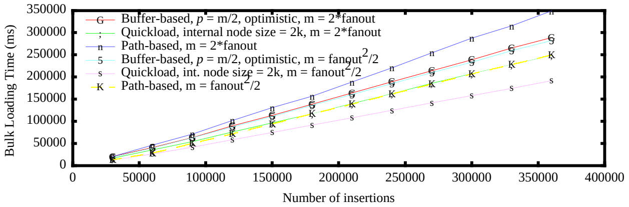

FIGURE 7: TEX_HYDRO

5.3 Bulk Loading Slim-trees

M-trees and its relatives like Slim-trees [TTSF 00] are much more CPU-bound than the members of the R-tree family. As a consequence, the only bulk loading algorithm dedicated to the M-tree family developed so far [CP 98] tries to cope with this special property. In contrast to R*-trees, we were hence more interested in the impact of our algorithms on the trees' search performance.

When repeating the experiments presented in section 5.2 for Slim-trees, we found that the relative performance of the different algorithms did not change. However, some of the results for the CAL_HYDRO_shuffled data set are quite interesting. Note that none of the approaches relying on sorting the data are applicable to Slim-trees.

In Figure 8, the graphs give the cost for creating Slim-trees from the CAL_HYDRO_shuffled data set. Similar to the results of the R*-tree, each of the generic algorithms performs significantly better than Tuple-based bulk loading. Remember that these generic algorithms primarily reduce disk accesses, thus for Slim-trees which are CPU-bound the performance improvements are not very high. Once more, Quickload and Path-based bulk loading outperform Buffer-based bulk loading by a factor of 1.5.

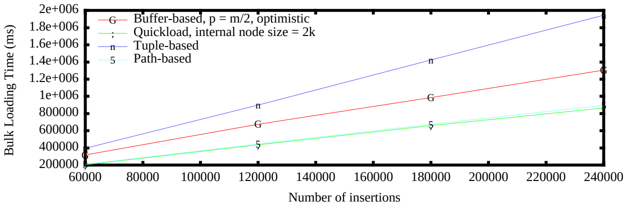

FIGURE 8: CAL_HYDRO_shuffled, m=2*fanout

A very interesting result is shown in Figure 9. Compared to Tuple-based bulk loading, the search performance of trees created by Quickload and Path-based bulk loading is much better (by a factor of 1.8). On the other side, the quality of trees suffers much from applying Buffer-based bulk loading to the CAL_HYDRO_shuffled data set. Hence, for Slim-trees it is much more important than for R*-trees to take the index quality into consideration when comparing bulk loading algorithms.

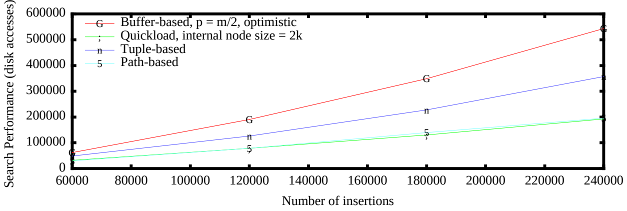

FIGURE 9: CAL_HYDRO_shuffled, m=2*fanout

In our second set of experiments, we used a 9-dimensional data set WEATHER containing weather data obtained by satellites. Because the size of each of the data objects is fairly high, we decided to enlarge the disk pages to 8 Kbytes. Figure 10 shows that for high dimensional data the overall bulk loading time is determined by the CPU-time. Therefore, our generic algorithms do not show any advantages over Tuple-based loading. However, the overhead of the Buffer-based approach is much higher than that of the other generic algorithms (Figure 11). In contrast to the results of the CAL_HYDRO_shuffled data set, the search performance of Quickload and Path-based bulk loading is quite similar to that of Tuple-based bulk loading. On the other hand, Buffer-based bulk loading once again produces trees with worse search performance.

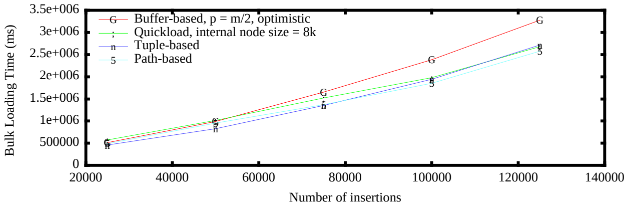

FIGURE 10: WEATHER_shuffled, m=2*fanout

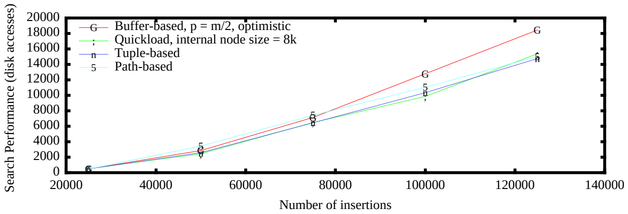

FIGURE 11: WEATHER_shuffled, m=2*fanout

6 Conclusions

In this paper, we outlined several types of generic bulk loading algorithms applicable to the broad class of Grow&Post-trees (GP-trees) which includes R-trees, M-trees, S-trees, MVB-trees, KDB-trees, hB-trees and others. Special attention was paid to overlapping-predicate trees (OP-trees) which is an important subset of GP-trees. We presented two generic bulk loading algorithms where the Path-based method is applicable to GP-trees and Quickload is limited to OP-trees. Both methods employ a large sample to build up a tree.

We found that Path-based bulk loading as well as Quickload are easy to implement, whereas a generic implementation of a worst-case optimal method turns out to be difficult. In a performance comparison with real world data sets, we demonstrated that the average case performance of Path-based bulk loading and Quickload is consistently superior to the performance of worst-case optimal methods. This even holds for cases where the sample quality is poor (e.g. the first tuples of a sorted sequence). For OP-trees, we also examined the search quality of the index and found that the quality of the target index is substantially better for Quickload in comparison to a tree build from a worst-case optimal method. In particular for bulk loading M-trees, we observed that the search quality of the resulting trees differs considerably. In summary, each of our new generic methods provides excellent performance with a low implementation overhead.

References

[AHVV 99] L. Arge, K. Hinrichs, J. Vahrenhold, J. S. Vitter: *Efficient Bulk Operations on Dynamic R-trees*. ALENEX 1999: 328-348

[Arg 95] L. Arge: The Buffer Tree: *A New Technique for Optimal I/O-Algorithms* (Extended Abstract). WADS 1995: 334-345

[AV 88] A. Aggarwal, J. S. Vitter: *The Input/Output Complexity of Sorting and Related Problems*. CACM 31(9): 1116-1127 (1988)

[BBD+ 01] J. van den Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, B. Seeger: *XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries*. VLDB 2001

[BBK 98] S. Berchtold, C. Böhm, H.-P. Kriegel: *Improving the Query Performance of High-Dimensional Index Structures by Bulk-Load Operations*. EDBT 1998: 216-230

[BDS 00] J. van den Bercken, J.-P. Dittrich, B. Seeger: *javax.XXL: A prototype for a Library of Query processing Algorithms*. SIGMOD Conference 2000: 588

[BGO+ 96] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer: *An Asymptotically Optimal Multiversion B-Tree*. VLDB Journal 5(4): 264-275 (1996)

[BKK 96] S. Berchtold, D. A. Keim, H.-P. Kriegel: *The X-tree : An Index Structure for High-Dimensional Data*. VLDB 1996: 28-39

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

[BSW 97] J. van den Bercken, B. Seeger, P. Widmayer: *A Generic Approach to Bulk Loading Multidimensional Index Structures*. VLDB 1997: 406-415

[Bur 96] Bureau of the Census: *Tiger/Line Precensus Files: 1995 technical documentation*. Bureau of the Census, Washington DC. 1996

[CP 98] P. Ciaccia, M. Patella: *Bulk loading the M-tree*. Proc. of the 9th Australian Database Conference, pp. 15-26, 1998

[CPZ 97] P. Ciaccia, M. Patella, Pavel Zezula: *M-tree: An Efficient Access Method for Similarity Search in Metric Spaces*. VLDB 1997: 426-435

[Dep 86] U. Deppisch: S-Tree: *A Dynamic Balanced Signature Index for Office Retrieval*. SIGIR 1986: 77-87

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

[FL 95] C. Faloutsos, K.-I. Lin: FastMap: *A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets*. SIGMOD Conference 1995: 163-174

[GG 98] V. Gaede, O. Günther: *Multidimensional Access Methods*. Computing Surveys 30(2): 170-231 (1998)

[Gra 93] G. Graefe: *Query Evaluation Techniques for Large Databases*. ACM Computing Surveys 25(2): 73-170 (1993)

[Gut 84] A. Guttman: R-Trees: *A Dynamic Index Structure for Spatial Searching*. SIGMOD Conference 1984: 47-57

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

[JDO 99] C. Jermaine, A. Datta, E. Omiecinski: *A Novel Index Supporting High Volume Data Warehouse Insertion*. VLDB 1999: 235-246

[JNS+ 97] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, Rama Kanneganti: *Incremental Organization for Data Recording and Warehousing*. VLDB 1997: 16-25

[KF 93] I. Kamel, C. Faloutsos: *On Packing R-trees*. CIKM 1993: 490-499

[KR 98] Y. Kotidis, N. Roussopoulos: *An Alternative Storage Organization for ROLAP Aggregate Views Based on Cubetrees*. SIGMOD Conference 1998: 249-258

[LEL 97] S. T. Leutenegger, J. M. Edgington, M. A. Lopez: *STR: A Simple and Efficient Algorithm for R-Tree Packing*. ICDE 1997: 497-506

[LL 98] S. T. Leutenegger, M. A. Lopez: *The Effect of Buffering on the Performance of R-Trees*. ICDE 1998: 164-171

[Lom 91] D. B. Lomet: *Grow and Post Index Trees: Roles, Techniques and Future Potential*. SSD 1991: 183-206

[LR 98] M.-L. Lo, C. V. Ravishankar: *The Design and Implementation of Seeded Trees: An Efficient Method for Spatial Joins*. TKDE 10(1): 136-152 (1998)

[LS 90] D. B. Lomet, B. Salzberg: *The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance*. TODS 15(4): 625-658 (1990)

[MNPW 00] P. Muth, P. E. O'Neil, A. Pick, G. Weikum: *The LHAM Log-Structured History Data Access Method*. VLDB Journal 8(3-4): 199-221 (2000)

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

[Rob 81] J. T. Robinson: *The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes*. SIGMOD Conference 1981: 10-18

[TTSF 00] C. Traina Jr., A. J. M. Traina, B. Seeger, C. Faloutsos: *Slim-Trees: High Performance Metric Trees Minimizing Overlap Between Nodes*. EDBT 2000: 51-65