Bulk loading a Data Warehouse built upon a UB-Tree *†

Robert Fenk[x] Akihiko Kawakami[+] Volker Markl[x]
Rudolf Bayer[x] Shuichi Osaki[+]

[x]Bavarian Research Center for
Knowledge Based Systems
Orleansstraße 34, 81667 Munich, Germany
Phone: +49-89-48095-216, -191
Fax: +49-89-48095-203
{fenk,markl}@forwiss.de, bayer@in.tum.de

[+]Information System Research and
Development Institute,
Teijin Ltd.
Urbannet-Yokohama Bldg., 5-2, Nihon-Odori, Naka-ku,
Yokohama 231-0021 Japan
{kawakami,osaki}@tjnsys.co.jp

Abstract

This paper considers the issue of bulk loading large data sets for the UB-Tree, a multidimensional index structure. Especially in dataware housing (DW), data mining and OLAP it is necessary to have efficient bulk loading techniques, because loading occurs not continuously, but only from time to time with usually large data sets.

We propose two techniques, one for initial loading, which creates a new UB-Tree, and one for incremental loading, which adds data to an existing UB-Tree. Both techniques try to minimize I/O and CPU cost. Measurements with artificial data and data of a commercial data warehouse demonstrate that our algorithms are efficient and able to handle large data sets. As well as the UB-Tree, they are easily integrated into a RDBMS.

Keywords: *bulk loading, UB-tree, multidimensional index, dataware housing, data mining, OLAP*

1 Introduction

In case of loading a huge amount of data into a data base indexed table, it is usually not feasible to use the standard insert operation of the index, since this would result in a big overhead caused by unnecessary page accesses. In most cases, tuples emerge in random order or an order, which is not suitable for the used index and therefore will lead to random inserts. With *random insert* we denote that consecutive tuples will be inserted into different pages and consequently an index search and a data page access is necessary for each insertion. In case of just one data page no random access to disk (it is cached) nor a page split happens. However, random page accesses become frequent after subsequent page splits. Therefore, the cost for loading will not increase linear to the number of required pages but linear to the number of new tuples.

Additionally, the query performance is of crucial interest, depending on clustering and page utilization. When loading indexes, clustering and page utilization might depend on the insertion order of tuples or the data distribution [5], e.g., k-d-B-Trees [19] cannot guarantee a certain page filling degree. But this is not desirable.

So far there has been already some work on bulk loading of multidimensional index structures. Various papers exist on this subject for other multidimensional index structures, e.g., for R–Trees [21, 8, 13, 11, 6], Gridfiles [12] and quad trees [7], and now we like to address this issue for the UB-Tree.

The contribution of this paper is to describe two simple and efficient bulk loading algorithms for the UB-Tree, one for initial bulk loading and one for incremental bulk loading. We present how to reuse existing techniques, which have been tested and proven to be robust and fast in practice. This simplifies the integration of these algorithms to a large extent.

The rest of the paper is organized as follows: Section 2 gives an short introduction to the UB-Tree. In section 3 we discuss the general problem and in section 4 we present specific solutions for the UB-Tree. A short performance analy-

*Japanese Patent filed on 22 May 2000. Application number: 2000-149648

sis is given in section 5 and section 6 presents measurement results. Section 7 concludes the paper.

2 The UB-Tree

The UB-Tree is a multidimensional clustering index, which inherits all good properties of B-Tree [4]. Logarithmic performance guarantees are given for the basic operations of insertion, deletion and point query, and a page utilization of 50% is guaranteed. The UB-Tree clusters data according to the space filling Z–curve [16] (see Figure 1 (a)) and introduces the new idea of partitioning the data space into disjoint Z–regions (see Figure 1 (c) and (d)). The Z–address, which is the position of a tuple on the Z–curve (see Figure 1 (b)), determines the Z–region to which the tuple belongs. Z–regions are stored on the data pages of the underlying B-Tree variant identified resp. separated from each other by the last included Z–address. This Z–address is used for searching Z–regions stored in the B-Tree. Page overflows are handled by splitting the affected page in the middle, where the split Z–address divides the tuples of the page. Then these tuples are distributed to the existing page, which is updated, and a newly created page identified by the split address.

For example insertion of the tuple (4,5) into the two dimensional universe of 8×8 points as depicted in Figure 1 (b) and (c) is performed as follows. We calculate the Z–address of this point, which is 38, and then locate the region containing this point, which is region *5*. Now we retrieve the page corresponding to this region and insert the point. When necessary a split is performed before storing. Finally the page is stored.

These Z–regions in conjunction with a sophisticated algorithm for multidimensional range queries [3] and the Tetris [15] algorithm for sorted reading of multidimensional ranges offer excellent properties [14] for multidimensional applications like dataware housing, archiving systems, temporal data management, etc. Integrating the UB-Tree into a RDBMS using a B-Tree is very simple [18], since the UB-Tree is a multidimensional extension to the B-Tree, or any of its variants.

3 General Problem Description

Efficient bulk loading is an issue when loading a huge amount of data into an indexed table. When creating a new index we want to provide it as fast as possible to the users and we want to gain the best possible performance for range queries. According to [17] getting data into a data warehouse is a critical process in the initiation and maintenance of a data warehouse application. The sheer volume of data involved dictates the need for a high-performance data loader. The ability to store vast data sets efficiently can have a dramatic effect on the overall cost associated with the maintenance of a data warehouse application.

The source data is usually provided in some kind of portable format, e.g., ASCII flat files, XML, Excel-Table, etc., but not in binary format. This happens especially in dataware housing and data mining applications when moving data from the OLTP systems to the OLAP system.

The main goals, which should be achieved by bulk loading techniques, are the following ones:

1. Minimize random disk accesses,
2. minimize disk I/O,
3. minimize CPU load,
4. optimize clustering and
5. optimize page filling.

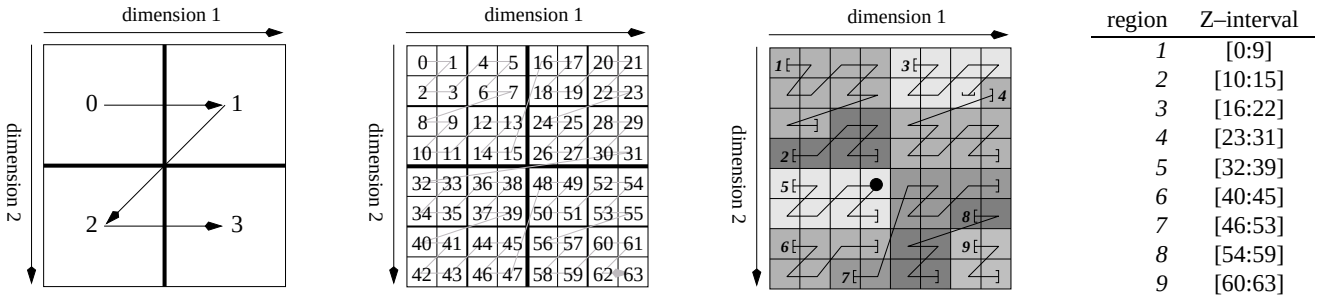
The main cost for loading arises by accesses to secondary storage, strictly speaking the loading process is I/O bound. Therefore, it is essential to minimize disk I/O and especially to avoid random accesses wherever possible. On the other hand linear disk accesses are quite fast because of caching techniques of todays hard disks and operating systems. Consequently the CPU load is also to be recognized as critical factor for the loading process and should be minimized as well.

With respect to query performance it is important to provide good clustering, since this reduces random disk accesses for range queries. There are two types of clustering, namely tuple and page clustering. For bulk loading it is possible to achieve both, in other words not only the tuples within one page should be clustered, but also the pages should be clustered according to the criterion of the used index.

Additionally, a good degree of page filling should be achieved in order to decrease the number of pages to load for answering range queries. For databases, which are only loaded once without ever adding new data, e.g., CD-ROM databases, the pages should be filled up to their maximum. A given page utilization can be guaranteed for initial loading, but for incremental loading only 50% is guaranteed. However, it might be higher depending on the data distribution with respect to the loaded data.

4 Algorithm Description

Multidimensional bulk loading algorithms can be classified according to two groups: a) Algorithms which apply a certain partition to the multidimensional input data and load those partitions into the index, b) and algorithms which apply a total sort order to the input data and load the pre-sorted data into the index.



region	Z–interval
1	[0:9]
2	[10:15]
3	[16:22]
4	[23:31]
5	[32:39]
6	[40:45]
7	[46:53]
8	[54:59]
9	[60:63]

(a) Z–curve/ 1st level (b) Z–curve/ 3nd level (c) Z–region Partitioning (d) Z–intervals

Figure 1. Z–curve and Z–regions used by the UB-Tree

In the first category algorithms like [6] for R–Trees and [12] for Grid–Files are available, while the second group applies to [8] for R–Trees which sorts the data according to the Hilbert curve as well as to the algorithms we present here.

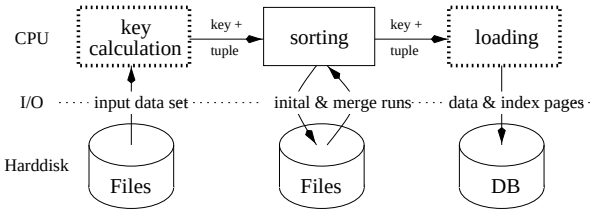


Figure 2. Bulk Loading Architecture for one dimensional clustering indexes

The bulk loading algorithms in this second category have three processing steps in common (Figure 2), first they calculate the key for each tuple of the data set, second the tuples are sorted according to the key and third the sorted data is loaded into the index.

In order to speed up the whole process it is useful to arrange these steps in a pipeline as depicted in Figure 2, since this avoids writing temporary results between processing steps to secondary storage, i.e., between the calculation and the sort step and between the sort and the loading step. Additional performance can be gained by using a binary format for the intermediate results, because it avoids conversion from internal binary representation and external ASCII representation and vice versa. This saves usually some disk I/O, but what is even more important it saves a lot of CPU time, because subsequent parsing and key calculation for each tuple is avoided.

4.1 Key Computation

The key (in DW the dimensions), that is used to identify a tuple is usually a subset of the attributes of a tuple. However, internally the used index might use a different representation. For example compound B-Trees use just a concatenation of the key attributes, but with space filling curves there is an additional computation, which calculates the scalar value on the curve representing the point specified by the key attributes resp. coordinates. B-Trees that allow such computation are called functional B-Trees. The UB-Tree uses such a B-Tree and adds calculation of the Z–address. This calculation is very efficient, because it only requires bit interleaving of the key attributes (the dimensions) [16].

The keys are also used in the sorting and loading process and therefore it is useful to add the key to each tuple of the temporary data sets in order to reuse it for the subsequent processing steps. This and using a binary format for the intermediate files minimizes CPU load.

4.2 Merge Sorting

The best choice for external sorting of large data sets is the merge sort algorithm [9]. It pre-sorts junks of the data set, which fit into RAM and writes them back to disk as initial runs. The best strategy to get the longest possible runs is to use a heap for internal sorting, i.e., one gets $2M$ long initial runs [9], where M is the number of tuples which fit into internal memory. Those initial runs are then merged, where as much files as possible are merged at once in order to avoid subsequent merge runs requiring additional disk accesses.

The disk accesses necessary for this are sequential accesses, which are quite fast, because of the pre-fetching of hard disks and operation systems. In order to avoid unnecessary I/O resp. disk accesses it is possible to integrate the

creation of initial runs into the key computation and just perform merging in the second step.

4.3 Initial Loading

When creating a new index from scratch we speak of *initial loading*. In case of the UB-Tree it is quite simple, because once the data set is sorted according to the Z–address, we can use B-Tree standard techniques.

For the loading algorithms we use a special kind of page data structure, called *large page* which is equal to the pages stored on disk, but it can store twice the tuples of a standard page. It is used only internally and may have a page utilization of 200% with respect to a standard page. This is necessary in order to simplify the algorithms and all percentage statements, which are given in the following are with respect to standard pages. We also do not discuss the maintenance of the index pages of the underlying B-Tree, because this can be archived by standard techniques [20], where we just cache the index pages, which lead to the current data page.

The initial loading algorithm works as following: It starts with an empty large page called the `page`. Now it reads tuples and adds them to this large page until the page utilization is two times the specified fill-degree. When it exceeds this limit it splits the current large page in the middle into two normal pages `pg1` and `pg2` containing each half of the tuples of the large page. `pg1` is now complete, it has the specified page utilization, and is written to disk. The contents of `pg2` is copied to the large `page`. The algorithm continues now adding tuples to the large page and splitting it as before, until no more tuples are left.

When finished, it has to check, if the large `page` needs a final split before storing it, since it might be filled to more than 100% of a standard page. If so it splits the page and stores the two new standard pages `pg1` and `pg2` otherwise the page can immediately be stored as standard page, because its page utilization is less or equal than 100%.

This algorithm allows for guaranteeing a certain page utilization. Just for the last two pages it is not possible to guarantee this and they might be filled only to 50%. The algorithm provides tuple clustering as well as page clustering, since the input data set is already sorted according to the Z–curve. In case of CD-ROM databases one can achieve the maximum page utilization, which is up to 100%.

This algorithm can also be used to reorganize an existing UB-Tree, by reading the tuples not from a flat file or the like, but from an existing UB-Tree. With some more modification it is also easily possible to merge a pre-sorted data set and an existing UB-Tree in order to get a new UB-Tree. This method (*initial merge loading*) is superior when the new flat file would contribute tuples to each page of the existing UB-Tree, because it can guarantee a specified page filling degree and clustering. However, when new tuples contribute only to a subset of pages of the existing UB-Tree and the majority of tuples contribute to new pages, it is much faster to use an incremental loading algorithm – although it is no longer possible to give the strict guarantees for clustering and page utilization.

4.4 Incremental Loading

When extending an existing UB-Tree we speak of *incremental loading*. Incremental loading differs only slightly from initial loading, because it does not only create new pages as initial loading does, but additionally it updates existing pages.

The algorithm works as follows: We start with a large `page` marked as invalid. Now the algorithm reads the first tuple from the input data set and checks if the tuple belongs to the current `page`. If not, it stores the current large `page` with the function `writeLastPage`, which cares for a split. Now it retrieves the existing page to which the tuple belongs from the UB-Tree. This page can be retrieved with a simple B-Tree point search. That page is copied to the large `page` which is used internally. Now it inserts the tuple into the large `page` and checks if its page utilization is two times the specified fill-degree. If so, it splits the current large `page` in the middle into two normal pages `pg1` and `pg2` containing each half of the tuples of the large page. The page containing the just inserted tuple is copied to the large `page` and the other one is written to disk. The algorithm continues adding tuples to the current `page`, checking and splitting it as before until no more tuples are left. When finished, it stores the current `page` with the function `writeLastPage`.

Table 1 depicts this algorithm in pseudo C-code. It should be mentioned that the `storePage` functions needs to check if the page to be stored does already exist or if a new page needs to be created. This can be achieved by marking those pages, which have been retrieved from the UB-Tree. When splitting such a page it is necessary to create the page `pg1` identified by the split address and `pg2` has to be updated, because it keeps its Z–address.

Compared to the initial loading algorithm there are two new parts. One is the check if a tuple belongs to the current page and the other, a new strategy for keeping the right page after a page split. The desired page utilization can be specified by `filldegree` and will be guaranteed except for the last two pages or the last two pages of a continuous sub-part of the Z–curve occurring when the last page needs to be split before storing. Those might be filled only to 50%. The last two pages of a sub-part of the Z–curve occur when a tuple does not belong to the current page (but to another existing page of the UB-Tree) and if the current page needs a split before storing. But, in order to increase

```
Tuple tuple;
ZAddress zaddr;
LargePage page = INVALIDPAGE;

while (datasetNotExhausted) {
  readTupleAndKey(&tuple, &zaddr);
  if (zaddr >= page.endzaddr ||
      page == INVALIDPAGE) {
    if (page != INVALIDPAGE) {
      writeLastPage(page);
    }
    page = retrievePage(zaddr);
  }
  insertTuple(&page, zaddr, tuple);
  if (getFillDegree(page) >
      2 * filldegree) {
    Page pg1, pg2;
    splitPage(page, &pg1, &pg2);
    if (zaddr <= pg1.zaddr) {
      storePage(&pg2);
      page = pg1;
    }
    else {
      storePage(&pg1);
      page = pg2;
    }
  }
}

writeLastPage(page);
```

Table 1. Core of Incremental Loading

the page utilization in this case, it is possible to apply the enhancements known for B*-Trees [10, 2] i.e., to use a set of pages to distribute tuples between those pages in order to get a better page utilization.

Page clustering can be guaranteed only for the newly created pages. When the majority of tuples from a new input data set contributes to new pages then it is much faster to use incremental loading instead of initial merge loading, which merges the new data set and an existing UB-Tree.

4.5 DBSM-Kernel Integration

The UB-Tree utilizes an underlying B-Tree, therefore one dimensional bulk loading techniques can be reused. This makes it easy to integrate the UB-Tree [18] as well as UB-Tree bulk loading. It is only necessary to extend existing loading tools by Z–address calculation and page splitting for UB-Trees.

This means there are two simple changes in the bulk loading tools having the architecture depicted in Figure 2. The affected parts are designated by stripped boxes. The first part of the loading process (key calculation) requires adding Z–address calculation for UB-Trees. This should be nothing more than adding a new key calculation function. The other change is in the third part of the loading process. Here, it is necessary to use the UB-Tree page splitting algorithm in order to get well formed regions, i.e., regions which are as rectangular as possible, since this affects the query performance.

Thus, it is possible to use existing loading techniques, e.g., for TransBase HyperCube, the first DBMS with integrated UB-Tree, we have made these changes without changing the actual loading algorithm of the Transbase loader.

5 Performance Analysis

The complexity of external sorting in general has been discussed in [9, 1].

With merge sort in practice, sorting a file of P pages with n tuples is linear and requires P reads for the input data, P writes for the initial runs, P reads for the initial runs and P writes for the resulting output data. This makes $2P$ sequential reads and $2P$ sequential writes. CPU cost divides into key calculation, which has also $O(n)$ plus the cost for the used internal sorting algorithm e.g., worst case for heap sort is $O(n \log n)$.

Concerning only the I/Os of data pages without caching we can make the following worst case estimations for the number C of page reads and writes, where M is the number of tuples which fit onto one page, n the number of new tuples and o the number of old tuples, which reside already in the index. Index pages are neglected, because they can be cached usually within main memory.

random insert: $C = \frac{n}{M} + 2n$, because $\frac{n}{M}$ pages are read clustered from the input file and one page has to be retrieved and stored from the existing UB-Tree for each tuple.

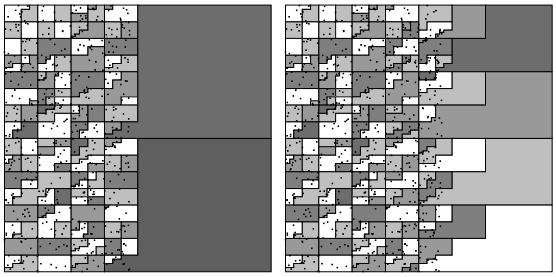
initial loading: $C = 2\frac{n+o}{M}$, because $\frac{n}{M}$ pages of data are read clustered from the input file and $\frac{o}{M}$ pages are read from the existing UB-Tree, while $\frac{n+o}{M}$ pages are created.

incremental loading: Worst case is $C = \frac{n}{M} + 2n$, when each input tuple belongs to another page. This can happen only when the number of input tuples is equal or less than the number of pages in the existing UB-Tree. The best case happens when there are nearly no updates of existing pages. It is $C = 2\frac{n}{M}$, since $\frac{n}{M}$ pages are read clustered from the input file and $\frac{n}{M}$ pages are newly created.

For incremental loading it is a bit more complicated, because we have to take into account if new tuples contribute to existing pages of the UB-Tree or to new pages. In worst

case, when each tuple contributes to a different existing page, we get the same performance as for random insert. However, when new tuples mainly contribute to new pages, we get a performance which is similar to initial loading of a new UB-Tree.

Figure 3 depicts this case where loading new data contributes to empty parts of an existing UB-Tree. This affects usually only a limited number of pages and therefore only a few pages of the existing UB-Tree have to be updated. In the example we have loaded a new data set into the area left to the middle of the two dimensional data space. This required only two updates of existing pages. Another incremental load further left would require 8 or 4 page updates, depending on the position of the new data. The necessary updates can be estimated with some changes to the cost functions for UB-Tree presented in [14].



(a) Before loading (b) After loading

Figure 3. Changes of UB-Tree space partitioning during incremental loading of a new data set into an unpopulated part of a two dimensional universe

For data warehousing new input data usually contributes to a part of the cube, which contains no data at this time. Therefore, incremental loading is superior to initial loading, which performs a merge of the possibly huge existing UB-Tree and the new data set.

6 Performance Evaluation

The measurements presented here have been made with the prototype implementation of the UB-Tree, since this allows for controlling all involved parameters. It is realized as middleware between a database management system and a database application.

In cooperation with Teijin, which is developing an OLAP server application, we have implemented the presented loading algorithms for the Transbase and Oracle UB-Tree middleware.

Tuple #	Random Order		Page Numbers	
	ASCII	Binary	Initial	Incremental
50k	55s	39s	1191	1235
1m	2259s	939s	23810	24537

Table 2. Comparison for artificial data

Previous measurements have proven that the different DBMSs perform equally, therefore we present only the measurements for the Transbase version.

It is obvious that page and tuple clustered UB-Trees with a guaranteed page utilization are best for range query performance. Therefore, we do not present measurements of range queries, but only informations about the clustering resp. the number of updated pages for incremental loading. See also [20] for a discussion of time and space optimality in B-Trees.

6.1 Artificial Data

The used machine for this measurement was a Sun Ultra 1 with a 167 MHz CPU and 64MB RAM. The secondary storage medium was an external IBM Ultrastar 18XP hard disk with 18GB storage.

We have used a data warehouse cube according to one used at Teijin, which had the four dimensions *Customer*, *Organization*, *Product* and *Time*. The data distribution was uniform and we loaded two different data sets. One with 50000 (50k) tuples and another one with one million (1m) tuples. The tuples size was 32 bytes and 56 tuples fit to one 2kB page.

The measurement results depicted in Table 2 compare the following cases: a) Loading of binary vs. ASCII format of temporary files, b) the number of created pages of initial loading vs. incremental loading.

We can neglect the order of input data in practice, since no existing indexes deliver Z–order. Therefore we present only loading of random ordered data.

The loading time was measured in seconds. This yields that using binary format will result in $\approx$ 30% shorter loading time. For bigger data sets it should be even faster, since more merge runs are necessary. This is clearly visible in case of the 1m-random measurement.

The number of pages created with incremental loading of 50k tuples is 3% higher than with initial loading. However, building the complete 1m DB with one initial and 6 incremental runs took 1722, while merging the existing UB-Tree and the new data sets with initial loading took 6280 seconds. Therefore incremental loading is $\approx$3.6 times faster than initial loading. For greater data sets it will become even worse.

6.2 Market Analysts Institute Data Warehouse

The used machine for this measurement was a Sun Enterprise Server 450 with two Sparc-Ultra4 248 MHz CPUs and 512MB RAM. The secondary storage medium was an external 90 GB RAID system.

In order to evaluate our algorithms with real world data we have loaded a data warehouse from a leading German consumer-market analysts institute (MAI). They store data pre-aggregated to two month periods, in order to reduce the data volume and because some data sources do not deliver data in a finer granularity. The data was stored in a cube with the three dimensions *Product*, *Segment* and *Period*.

The snapshot of the data warehouse used for our measurements consists of $\approx$ 43 million fact tuples belonging to fifteen two-month periods. In our measurements we have only considered the fact table, since this is the biggest table and new data contributes mainly to this table. The source data was stored in ASCII flat files and the binary representation of one tuple had 56 bytes. The page size was 2kB, but due to the overhead for page management only 31 tuples could be stored per page. Page utilization was set to 78%, which results in 24 tuples per page.

Figure 4 depicts the data distribution in tuples according to the periods. This is interesting, since loading time should reflect the data distribution of the periods.

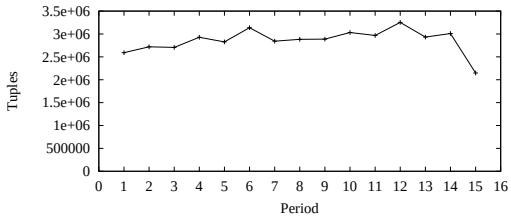


Figure 4. Data Distribution of MAI DW

The periods have been loaded one by one into a UB-Tree. The first one was loaded with initial loading and the subsequent ones with incremental loading resp. initial merge loading, which merges the new periods data with the existing UB-Tree into a new UB-Tree. Random insert is not considered, because it is not competitive.

Figure 5 shows the measured times for initial merge loading and incremental loading. The times for Z–address calculation plus creation of initial runs and sorting is also plotted. For loading the first period we have used initial loading with ASCII temporary files and incremental loading with binary temporary files, which loaded a UB-Tree consisting of one empty page. We can see that difference in the format of the temporary files gains a speedup of factor of two. For loading the second period the two algorithms perform similar, but for the subsequent periods incremental loading is clearly faster, because it only depends on the number of input tuples. This can be seen also in comparison with the data distribution in Figure 4. For an integrated version of the loading algorithms one should expect a speedup of several factors, because the current implementation of the UB-Tree causes a lot of interprocess communication and performs own page handling.

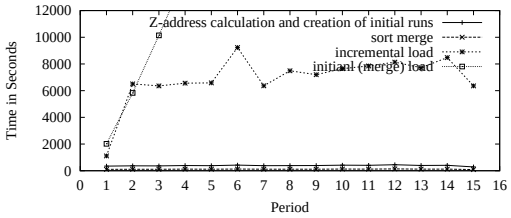


Figure 5. Loading Performance for MAI DW

A look at the page statistics in Figure 6(a) yields that in average there have been 20 updated pages for loading one period where there are approximately 119393 new pages. This confirms our assumptions and analysis that loading new data into an unpopulated part of the cube updates only a small portion of the existing pages. Figure 6(a) shows that 99.99% of the pages of the final UB-Tree are filled with 24 resp. 23 tuples (a page utilization 74% and 77%). The other 0.0174% of the tuples are stored on pages with an average page utilization of 65%!

With initial loading of all periods one gets 1786163 data pages. This is means incremental loading produces just 4736 pages more than initial loading.

Very interesting is the fact that incremental loading usually requires to update only a few pages of the existing UB-Tree. This happens because the different periods contribute data to different parts of the cube, due to their difference in the *Time* dimension. Therefore, no measurable performance disadvantage for range query processing will occur when using incremental loading instead of initial loading, but there is a big speedup when loading the data.

7 Conclusion

In this paper, we have considered the problem of bulk loading the UB-Tree. We have presented two UB-Tree bulk loading algorithms which are simple, robust and provide excellent performance.

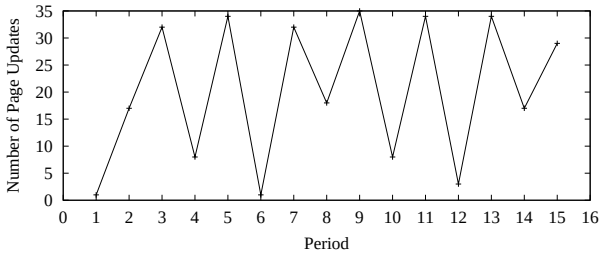
Initial loading provides tuple and page clustering, which lead to optimal range query performance. It can also be used for reorganizing UB-Trees and merging an existing UB-Tree with others or a new data set. When initial loading is too expensive, since it affects only a subset of pages of an existing UB-Tree, we can use incremental loading, which is superior compared to random insertion. It is usually much

Period	total	updated	created
1	112641	1	112640
2	225952	17	113311
3	338714	32	112762
4	460728	8	122014
5	578525	34	117797
6	709274	1	130749
7	827758	32	118484
8	947911	18	120153
9	1068263	35	120352
10	1194613	8	126350
11	1318283	34	123670
12	1453864	3	135581
13	1576039	34	122175
14	1701450	17	125411
15	1790900	29	89450
total	–	303	1790899
average	–	20.2	119393.26

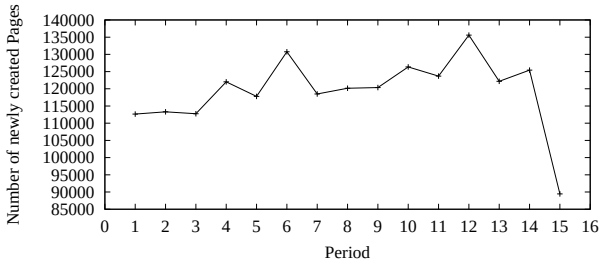
(a) Pages creations and updates

% of all pages	page count	page utilization	tuple number	% of all tuples
0.0018	33	51%	528	0.0012
0.0018	33	54%	561	0.0013
0.0020	36	58%	648	0.0015
0.0021	37	61%	703	0.0016
0.0020	36	64%	720	0.0017
0.0024	43	67%	903	0.0021
0.0024	43	70%	946	0.0022
6.2913	112670	74%	2591410	6.0451
93.6913	1677917	77%	40270008	93.9398
0.0004	7	80%	175	0.0004
0.0003	5	83%	130	0.0003
0.0005	9	87%	243	0.0006
0.0004	7	90%	196	0.0005
0.0002	4	93%	116	0.0003
0.0003	5	96%	150	0.0003
0.0008	15	100%	465	0.0011

(b) Page distribute according to utilization



(c) Page Updates



(d) Created Pages

Figure 6. Page Statistics for MAI DW

faster than random inserts and it is able to create partial page clustering.

Incremental loading is always beneficial when new data contributes only to so far unpopulated parts of the indexed space. Data warehouse applications have this property and therefore incremental loading is very beneficial because the number of page updates is minimal. Thus, we can give page utilization and clustering guarantees. Of course our techniques are also applicable to bulk loading UB-Trees in general.

Additionally we have shown that UB-Tree bulk loading can easily be integrated into a DBMS which uses B-Tree. Existing techniques for sorting etc. and the existing infrastructure of a DBMS can be reused. One may decide to make only minor changes in order to provide bulk loading tool for UB-Trees by extending an existing B-Tree bulk loading. However, one may also integrate the presented incremental bulk loading algorithm with some more effort in order to gain its benefits.

References

[1] A. Aggarwal. External Memory Algorithms and Data Structures: Dealing with MASSIVE DATA. In *Draft Version, February 4*. http://www.cs.duke.edu/ jsv/Papers/catalog/, 2000.

[2] R. A. Baeza-Yates. The Expected Behaviour of B^+-Trees. In *Acta Informatica 26(5)*, pages 439–471, 1989.

[3] R. Bayer and V. Markl. The UB-Tree: Performance of Multidimensional Range Queries. Technical Report TUM-I9814, Institut für Informatik, TU München, 1997.

[4] R. Bayer and E. McCreight. Organization and Maintainance of large ordered Indexes. In *Acta Informatica 1*, pages 173–189, 1972.

[5] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu. Client-Server Paradise. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB'94, Proceedings of 20th International*

Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile, pages 558–569. Morgan Kaufmann, 1994.

[6] Y. J. García, M. A. Lopez, and S. T. Leutenegger. A Greedy Algorithm for Bulk Loading R-Trees. In *ACM International Workshop on Advances in Geographic Information Systems*, pages 163–164, 1998.

[7] G. R. Hjaltason, H. Samet, and Y. J. Sussmann. Speeding up Bulk-Loading of Quadtrees. In *ACM International Workshop on Advances in Geographic Information Systems*, pages 50–53, 1997.

[8] I. Kamel and C. Faloutsos. On Packing R-trees. In *CIKM*, pages 490–499, 1993.

[9] D. E. Knuth. *Sorting and Searching*. Addison - Wesley, 1973.

[10] K. Küspert. Storage Utilization in B*-Trees with a Generalized Overflow Technique. In *Acta Informatica 19*, pages 35–55, 1983.

[11] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A Simple and Efficient Algorithm for R-Tree Packing. In A. Gray and P.-Å. Larson, editors, *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K*, pages 497–506. IEEE Computer Society, 1997.

[12] S. T. Leutenegger and D. M. Nicol. Efficient Bulk-Loading of Gridfiles. In *Proceedings of the IEEE Transactions on Knowledge and Data Engineering, Volume 9(3)*, pages 410–420, 1997.

[13] M.-L. Lo and C. V. Ravishankar. Generating Seeded Trees from Data Sets. In *Symposium on Large Spatial Databases*, pages 328–347, 1995.

[14] V. Markl. *Processing Relational Queries using a Multidimensional Access Technique*. PhD thesis, DISDBIS, Band 59, Infix Verlag, 1999.

[15] V. Markl, M. Zirkel, and R. Bayer. Processing Operations with Restrictions in RDBMS without External Sorting: The Tetris Algorithm. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 562–571. IEEE Computer Society, 1999.

[16] J. A. Orenstein and T. H. Merrett. A Class of Data Structures for Associative Searching. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, April 2-4, 1984, Waterloo, Ontario, Canada*, pages 181–190. ACM, 1984.

[17] A. Paller. Rely on Red Brick's Performance for Data Warehouse Applications. Technical report, Data Warehousing Institute, Informix Corporation, 2000.

[18] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhard, and R. Bayer. Integration the UB-Tree into a Database System Kernel. In *VLDB2000, Proceedings of International Conference on Very Large Data Bases, 2000, Cairo, Egypt*, 2000.

[19] J. T. Robinson. The K-D-B-Tree: A Search Structure For large multidimensional dynamic Indexes. In Y. E. Lien, editor, *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, April 29 - May 1, 1981*, pages 10–18. ACM Press, 1981.

[20] A. L. Rosenberg and L. Snyder. Time- and Space-Optimality in B-Trees. *TODS*, 6(1):174–193, 1981.

[21] N. Roussopoulos and D. Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-Trees. In S. B. Navathe, editor, *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 28-31, 1985*, pages 17–31. ACM Press, 1985.