

Bulk loading the large datasets

Author: Dhananjay Patil
Organization: Evaltech, Inc.
**Evaltech Research Group,
Data Warehousing Practice.**
Date: 08/25/04
Email: erg@evaltech.com

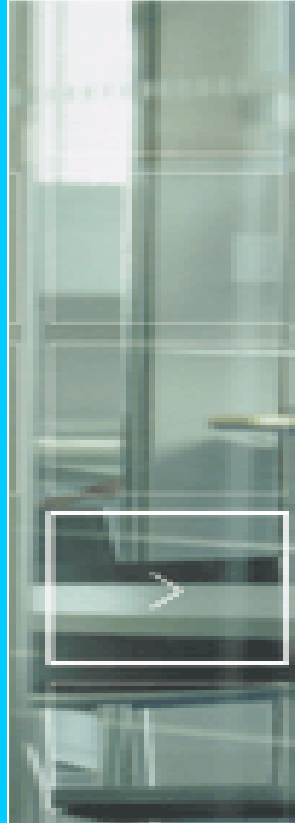


Abstract:

In data warehousing (DW), data mining and OLAP it is necessary to have efficient bulk loading techniques. In practice loading occurs not continuously so usually there is a large datasets. This white paper describes the techniques for initial and incremental loading. Also it includes the description of how these techniques achieves main goals i.e. minimize random disk accesses, minimize disk I/O, minimize CPU load, optimize clustering and page filling.

Intellectual Property / Copyright Material

All text and graphics found in this article are the property of the Evaltech, Inc. and cannot be used or duplicated without the express written permission of the corporation through the Office of Evaltech, Inc.



■ Summary

In data warehousing (DW), data mining and OLAP it is necessary to have efficient bulk loading techniques. In practice loading occurs not continuously so usually there is a large datasets. This white paper describes the techniques for initial and incremental loading. Also it includes the description of how these techniques achieves main goals i.e. minimize random disk accesses, minimize disk I/O, minimize CPU load, optimize clustering and page filling.

This white paper includes following topics.

- ❖ Introduction
- ❖ The UB tree (Multidimensional index structure)
- ❖ General problem
- ❖ Algorithm description and bulk loading architecture
- ❖ Initial loading and incremental loading
- ❖ Conclusion

■ Introduction

The initial loading technique creates a new UB-Tree i.e. extension to B-Tree, and incremental loading, techniques adds data to an existing UB-Tree. Both techniques try to minimize I/O and CPU cost. Also the UB-Tree, are easily integrated into a RDBMS.

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.

An index search and a data page access are 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.

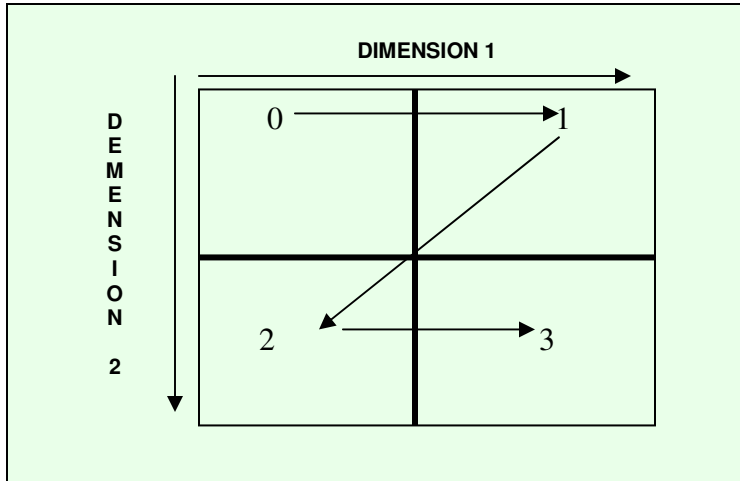
When loading indexes, clustering and page utilization might depend on the insertion order of tuples or the data distribution but this is not desirable. So that UB tree algorithms for initial and incremental loading is important.

■ The UB tree (Multidimensional index structure)

The UB-Tree is a multidimensional clustering index, which inherits all good properties of B-Tree. 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 introduces the new idea of partitioning the data space.

Integrating the UB-Tree into a RDBMS using a B-Tree is very simple, since the UB-Tree is a multidimensional extension to the B-Tree, or any of its variants.

Following diagram shows the UB-Tree cluster data according to the space filling Z-Curves



■ 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 it should be fast as possible to the users

But 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 data warehousing 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:

- Minimize random disk accesses,
- Minimize disk I/O,
- Minimize CPU load,
- Optimize clustering and
- Optimize page filling.

The main cost for loading arises by accesses to secondary storage, 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 today's 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.

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

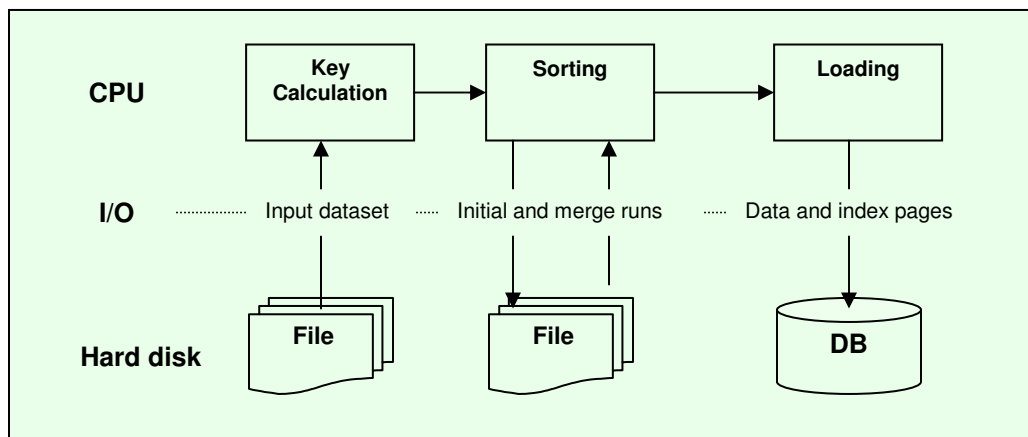
Additionally, a good degree of page filling should be achieved in order to decrease the number of pages to load for answering range queries. For databases, which are only loaded once without ever adding new data, e.g., CD-ROM databases, the pages should be filled up to their maximum. A given page utilization 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.

■ Algorithm Descriptions

Multidimensional bulk loading algorithms can be classified according to two groups:

- 1) Algorithms which apply a certain partition to the multidimensional input data and load those partitions into the index
- 2) Algorithms, which apply a total, sort order to the input data and load the pre-sorted data into the index.

◆ Bulk Loading Architecture for one-dimensional clustering indexes



The bulk loading algorithms of second category have three processing steps in common shown in above Figure, 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 shown in above Figure, 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.



Key Computation

The key (in DW the dimensions) is usually a subset of the attributes of a tuple, which is used to identify tuples. However, internally the used index might use a different representation.

For example compound B-Trees use just a concatenation of the key attributes, but in UB-Tree with space filling curves there is an additional computation. B-Trees that allow such computation are called functional B-Trees. The UB-Tree uses such a B-Tree and adds calculation. This calculation is very efficient, because it only requires bit interleaving of the key attributes.

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.



Merge Sorting

The best choice for external sorting of large data sets is the merge sort algorithm. It pre-sorts chunks 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.

■ Initial Loading

In case of the UB-Tree initial loading is simple, because once the data set is sorted according to the Z-address, then B-Tree's standard techniques can be used.

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.

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 content 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 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.

■ Incremental Loading

When extending an existing UB-Tree known as 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.

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 fill degree 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 the page utilization in this case, it is possible to apply the enhancements known for B-Trees 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.

■ Conclusion

- 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 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.
- UB-Tree bulk loading can easily be integrated into a DBMS, which uses B-Tree.