http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 7 (7): 1061-1066, 2008
ISSN 1812-5638
© 2008 Asian Network for Scientific Information

An Index Structure for Large Order Database Maintenance Using Variants of B Trees

K.M. Azharul Hasan
Khulana University of Engineering and Technology, Khulana, 920300, Bangladesh

Abstract: This study proposes and evaluates a B^+ tree based indexing scheme with the objective of accelerating some major operations such as insertion and retrieval using both primary memory and secondary storage. It also analyzes the effect on non leaf nodes of a B^+ tree as well as the height of tree. It is shown that the non leaf nodes specially the height of B^+ tree has significant effect for insertion and retrieval of key values. In addition, a new data structure and its operations are explained where the non leaf nodes of B^+ tree is replaced by a single list. Cost models are developed for theoretical analysis. Sufficient experimental results are provided to show the performance improvement and the cost models are validated.

Key words: Indexing, B^+ tree, B tree, database performance, range searching

INTRODUCTION

To support fast random access or range searches on column values, database systems allow creating secondary indexes on tables. Many techniques for organizing a file and its index have been analyzed in many literatures (Evangelidis *et al*., 1997; Bertino and Kim, 1989) while no single scheme can be optimum for all applications. The technique of organizing a file and its index called the B tree has become widely used. Much research has been focused (Bayer and Unterauer, 1977; Chen *et al*., 2001; Giinther and Bilmes, 1991) on improving node fan out, minimizing the tree height, impact of B tree page size and internal node architecture. B and B^+ trees are widely used index structures for relational databases and data warehouses (Eugene *et al*., 2001; Gupta *et al*., 1997) object oriented databases (Bertino and Kim, 1989) and parallel data bases (Taniar and Rahayu, 2002, 2004). B^+ trees are also used as container of compressed records (Hasan *et al*., 2005, 2006) itself for its low space cost and high retrieval performance. To access the height for the non leaf nodes of the B tree and its variants specially B^+ tree (Taniar and Rahayu, 2002) takes long time for insertion/deletion and retrieve operations. This study analyzes the effect of non leaf nodes of a B^+ tree as well as the height of the tree. It is shown that the non leaf nodes specially the height of B^+ tree has significant effect for insertion and retrieval performance. To resolve accessing the height of the tree, a data structure is proposed for efficient retrieval and insertion.

In a B^+ tree, all keys reside in the leaves of the tree (Comer, 1979). The upper levels, which are organized as a B tree, consist only of an index, a roadmap to enable rapid location of the index and key parts. In particular, leaf nodes are usually linked together left-to-right that links allow easy sequential processing.

In this study B^+ tree based indexing scheme with the objective of accelerating insertion and retrieval performance is proposed and evaluated. It is shown that the non leaf nodes of B^+ tree has very negligible effect on storage requirement and thus stored on main memory. As main memory gets cheaper, it becomes increasingly affordable to build computers with large main memories. It is possible to configure machines with gigabytes of main memory for small costs.

The classic paper of B tree was introduced by Bayer and McCreight (1972). The idea of delaying the splitting of a page until two neighboring pages is completely filled, in which case the two pages are split into three. This variant has become known as a B^* tree (Knuth, 1998). Actually B^* tree is a B tree in which each node is at least 75% full. The original version of the B tree stored complete records at all levels of the tree. The idea of storing all records at the leaf level and just keys in internal pages is described by Knuth (1998). This variant is known as a B^+ tree. The idea of storing short separators instead of complete keys in internal pages is known as Prefix B Tree (Bayer and Unterauer, 1977).

Cash Sensitive Search Trees (CSS Tree) (Rao and Ross, 1999) and Cash Sensitive B^+ trees (Rao and Ross, 2000) are proposed for main memory indexing techniques. The CSS tree is especially very compact and space efficient B^+ trees. CSS trees are essentially very pact with keys and laid out contiguously, level by level, in main memory.

B^+-tree indexes on a primary key are dynamic as well as the leaf level of the index consists of pages that contain the actual data items. Such a single-attribute

clustering index is very efficient for answering range queries on the indexed attribute, since data items with comparable values for their indexed attribute will be in the same or neighboring pages (Evangelidis *et al*., 1997).

Using a B^+ tree, each non-leaf node may consist of up to f keys and f+1 pointers to the nodes on the next level on the tree hierarchy (i.e. child nodes). All child nodes, which are on the left-hand side of the parent node, have the key values less than or equal to the key of their parent node. On the other hand, keys of child nodes on the right-hand side of the parent node are greater than the key of their parent node. The structure of leaf nodes is slightly different from that of non-leaf nodes. Each leaf node consists of up to f keys, where each key has a pointer to the actual record called data pointer and each node has one node pointer to a right-side neighboring leaf node. Having all data pointers stored on the leaf nodes is considered better than storing data pointers in the non-leaf nodes like the original B trees. Furthermore, by having node pointers in the leaf level, it becomes possible to trace all leaf nodes from the left most to the right most nodes producing a sorted list of keys.

THE CDL DATA STRUCTURE

The data structure that is proposed here is a two level tree structure. The first level of the structure is a one way list containing key pointer pairs where key is the first key and pointer is the starting address of a node of the second level. The first level of the structure serves as a gateway to the second level. Hereafter the first level will be called as head and the second level will be called leaf nodes as shown in Fig. 1. The second level contains the nodes of the tree. The leaf nodes contain at most k keys. The keys are search values and stored in ascending order. The key values are assumed to be unique. Within a node the keys are $K_1 < K_2 < \ldots < K_q$ where $q \le k$.

When a key value K_i is to be inserted, the largest value smaller than equal to K_i is determined in head and the pointer associated with the key value is followed to determine the corresponding leaf node. The key value K_i is inserted to the corresponding node.

If the node becomes over flow i.e., more than k keys needs to be entered in the node, the node is split into two in the middle such that one node contains [k/2] keys and another node contains [k/2] keys and the ascending order of the keys are maintained. The (key, pointer) pair of the new two nodes is stored into the head. The key K_i is then inserted to the appropriate leaf node. The (key, pointer) pair in the head are stored in such a way that the list head is ordered in terms of key values. Figure 1 shows the data structure CDL after inserting the keys 10, 16, 28, 32, 40, 24, 30, 36 in the order for k = 3.

Whenever a key value K_i is searched, the largest value smaller than equal to K_i is determined in head and the pointer associated with the key value is followed to determine the corresponding leaf node which contains the desired key value K_i. If the key value does not exist in the desired node the key is not present in tree. To search a range of key values, the traversal is performed by determining the leaf node containing the lowest value in the range then sequential search is performed on the leaf nodes until the node containing highest value in the range is determined or the end of the node is reached. The next pointer in the head is followed to determine the next leaf node and searched the leaf node until the largest value in range is found.

The head is placed on main memory and the leaf nodes are stored on secondary storage. Binary search is adopted to search the head.

It will be analyzed in the subsequent Sections that the storage requirement of head is small and can easily be placed in main memory. In the following the data structure is termed as Compressed Data List (CDL).

The proposed CDL index structure has the most important properties that should follow by an index structure (Lomet and Salzberg, 1990) such as good average storage utilization both in the index and in the data pages; large index fan-out; easy dynamic and incremental reorganization as the file grows; simple algorithms for different operations having no special case and ability to handle range searches effectively.

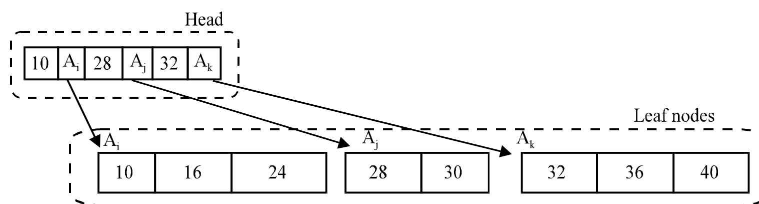

Fig. 1: The CDL data structure

COST ANALYSIS

In the following, the CDL implementation is called as CI and the conventional B^+ tree implementation is called as BI. In this Section the cost model for both CI and BI are developed and compared. The parameters that are considered are as follows. Some of these parameters are provided as input, while others are derived from the input parameters. All lengths or sizes are in bytes.

Parameters

NR : Total number of keys
d : Order of a node (both CI and BI)
f : Average fan out from a node of both BI and CI; $d \leq f \leq 2d$ for the leaf nodes of CI and the nodes of BI except the root node. For root node of BI, $d \leq 2 \leq 2d$.
kn : Average number of keys in a node, namely kn = f-1
pl : Length of a pointer
kl : Length of a key
P : Disk page size
X : Average size of a node
LN : No. of leaf nodes for both BI and CI, i.e., $LN = \lceil NR/kn \rceil$
NLN : No. of non leaf nodes for BI

The number of non leaf nodes NLN is determined as follows:

$$NLN = \lceil LN/f \rceil + \lceil \lceil LN/f \rceil / f \rceil + \ldots + Y$$

where each term is successively divided by f until the last term Y is less than f. If the last term Y is not 1, 1 is added to the total (due to the root node). The number of terms in the expression for NLN represents the number of non leaf nodes that must be accessed while scanning the B^+ tree. This is denoted by h_B. The height of the BI, is therefore, h_B+1.

The NLN non leaf nodes are organized as a linked list in the memory for the proposed CDL datastructure. Hence the total length of the head of CI is equal to NLN. Therefore, if the NLN increases then length of the head for CI in creases of course but the height of BI i.e, h_B will increase as well.

Assumptions: To simplify the cost model, following assumptions are made.

- The page size is greater than or equal to the node size of both CI and BI.
- The leaf nodes are same size for both BI and CI. Although in BI there are kn+1 pointers and in CI there are kn pointers in leaf nodes.

Storage cost

Cost of CI = No. of leaf pages (LP)
Cost of BI = No. of leaf pages (LP) + No. of non leaf pages (NLP)

Number of nodes that can fit in a page is determined by $TR = \lfloor P/X \rfloor$. (Assumption (i)). Hence total number of leaf pages for both CI and BI is $LP = \lceil LN/TR \rceil$

Total number of non leaf pages for BI is $NLP = \lceil NLN/TR \rceil$

Each node of a B^+ tree has kn keys and kn+1 (f = kn+1) pointers, hence Length of a node, $X = kn \times kl + f \times pl$ (Assumption (ii)).

Insertion cost

Cost for BI: The insertion cost can be determined in two cases.

- **Case 1:** For inserting a key in BI the least work is required if no splitting occurs, then it needs to access h_B nodes and 1 node to be updated or written.
- **Case 2:** The most amount of work is required if all the nodes in the retrieval path including the root node split into two. Since the retrieval path contains h_B nodes and a new root node will have to write. Hence nodes accessed is h_B and pages to write $2h_B+1$. Note that h_B always denotes the height of the old B^+ tree.

Cost for CI: For inserting a key in CI the least work is required if no splitting occurs for leaf nodes. Then it needs to access 1 leaf node and 1 leaf node to be updated or written. The most amount of work is required if the node in which the key is to be inserted is split into two. Then it needs to access 1 leaf node and 2 leaf nodes to be updated or written. Since with the increasing number of keys inserted into the head of CDL the length of head increases it has a effect for insertion a key. For high speed of the accessing main memory this cost is ignored.

Retrieval cost: The cost model for range key query is developed. A range key query has a single predicate of the form (key < value) or (key > value) or (key between value1 and value2). In formulating the range key queries, the following additional parameters are assumed.

NRQ: Number of key values present in the specified range.

NDRQ: Number of nodes to be visited for the range NRQ, namely $NDRQ = \lceil NRQ/kn \rceil$

The traversal is performed to determine the leaf node containing the lowest value in the range then sequential search is performed on the leaf nodes until the node containing highest value in the range is determined. In the following, the number of nodes (NA) to be accessed is formulated first and hence number pages accessed (PA) is determined both for CI and BI.

Cost for BI: Number of nodes accessed for BI is

$NA = h_B + NDRQ$ and
$NP = h_B + \lceil NDRQ/P \rceil$

h_B is the number of non leaf nodes that must be accessed for BI.

Cost for CI: Number of nodes accessed for CI is

$NA = NDRQ$ and
$NP = \lceil NDRQ/P \rceil$

It can be noted from the cost of BI and CI that for range key retrieval with a specified range NRQ CI has advantages of not accessing the height of the tree h_B.

To analyze experimental results, prototype systems are constructed based on BI and CI. The head is placed on main memory and leaf nodes are stored on secondary storage for CI on the other hand both leaf and non leaf nodes for BI is placed on secondary storage. The performance for storage, insertion and retrieval cost are analyzed in this section. The experimental data set are created automatically. The cost model developed in the previous Section is validated through the implementation.

EXPERIMENTAL RESULTS

Experimental set up: The values of the parameters that are used in the experiment are shown in Table 1. The parameter kl is chosen in such a way that the node size can be equal to (or smaller than) page size of the system (Assumption A. (i)). This is because the variants of B trees are developed with the objective of accessing minimum blocks or pages for database indexing. Hence the parameter kn is calculated as $Kn \leq P/(kl+pl)$. The parameter P and pl is machine dependent for experiment results. Hence for the varying values of kl the parameter Kn will be also be varied.

Storage cost: Figure 2 shows the storage requirement of BI and CI for the same data set for NR = 100000 to 400000. It is clear from Fig. 2 that the storage cost for both CI and BI are nearly equal. The non leaf nodes (NLN) of a BI has very negligible effect for storage cost comparing to the leaf nodes (LN). In the experiment, it is realized that the storage requirement for the non leaf nodes are less than 1% of the total storage of BI. Hence it can be concluded that if the non leaf nodes are stored in main memory its effects on memory management will also be negligible. But accessing the non leaf nodes needs the disk page to be accessed for insertion and retrieval process which is h_B. Hence it has great effect for insertion and retrieval. In the subsequent subsections it will be analyzed the effect of non leaf nodes for insertion and retrieval processes.

Table 1: The parameter values for the experiment

kl	8
P (System P)	8192
Pl	8
Kn	512

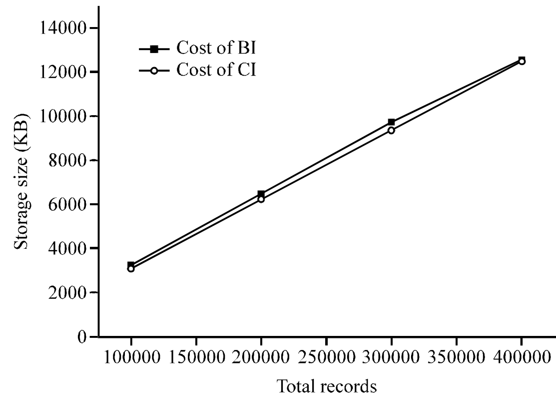

Fig. 2: The storage requirement for BI and CI

Moreover for a fixed size node, it is very rare that the nodes are more than 75% filled. Although a node is at least 50% filled for B^+ tree and 75% filled for B^* tree. Hence the empty part of non leaf nodes has significant effect for storage cost of BI. On the other hand, in CI the head is a simple list and has no occurrences of empty nodes.

Insertion cost: The insertion cost is analyzed for NR = 200000 both for BI and CI. The insertion time of each of the key in CI and BI is shown in Fig. 3a and b.

The insertion cost for CI has great advantages over BI. This is because the height of BI, h_B, which is to be accessed to insert a key in BI. The insertion cost for BI has so many uneven situation comparing to CI this is because when the nodes split into two it takes more time to insert a key. And when the non leaf nodes also split then the insertion cost for BI increases. When all the non leaf nodes including the root node split into two then the largest time is taken for BI as shown in Fig. 3a. In this case $2h_B + 1$ nodes need to be updated or written. Figure 3b shows the insertion cost for NR = 65000. In this Fig. 3 the

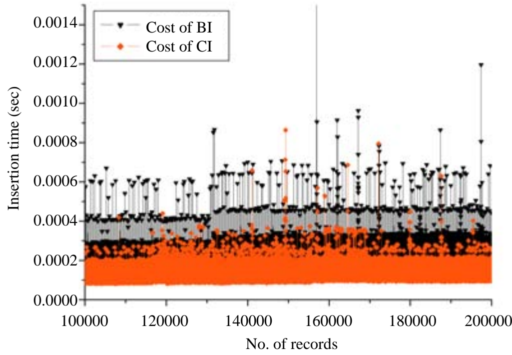

Fig. 3a: The insertion cost for BI and CI for large data set

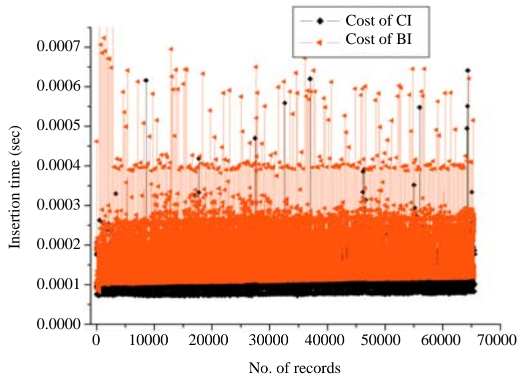

Fig. 3b: The insertion cost for BI and CI for small data set

similar results are found and the advantages for insertion cost for CI over BI can be realized clearly. The uneven insertion cost for BI proves the situation that discussed in cost analysis section.

Retrieval cost: The performance of range key retrieval for BI and CI are analyzed in this Section for varying NR and NRQ. Figure 4 shows the retrieval performance for range key retrieval for varying NRQ having NR =1000000. The cost for CI has advantages over the cost of BI. For a fixed NR to retrieve keys from BI it needs to scan the height of the tree h_B but in CI the head nodes are necessary to be searched which are less expensive than to face the disk on secondary storage. From Fig. 4 it can be realized that for varying NRQ the retrieval has significant improvement of CI over BI. The improvement found is nearly same for varying NRQ. This is because of the height scanned in retrieving a key in BI. Hence the effect of the height of BI can be realized.

Figure 5 shows retrieval comparison for varying NR having NRQ = 800. The values of NR increased in such a way that the height of BI is increased. Hence the increment of the values of NR in the x axis (i.e. NR) are not

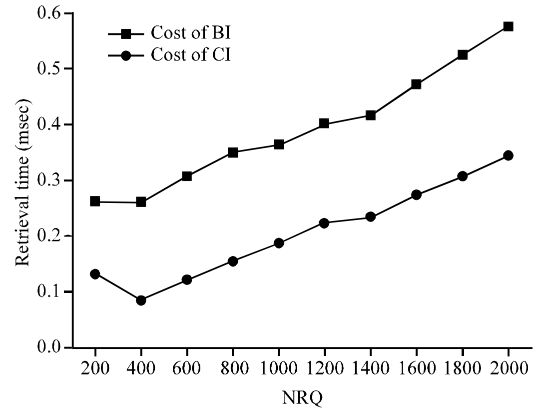

Fig. 4: Retrieval cost comparison for varying NRQ

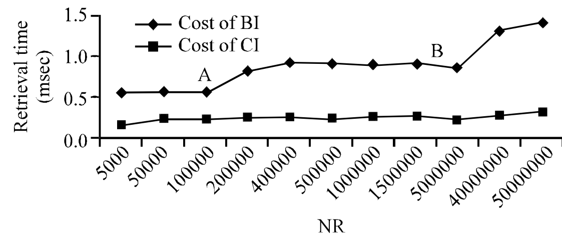

Fig. 5: Retrieval comparison for varying NR

linear. The cost of CI is constant for increasing NR for fixed NRQ as shown in Fig. 5. This is because for increasing NR having fixed NRQ same number of nodes needs to be faced from the disk for CI. Although the length of head increases but from the result it can be seen that this has negligible effect comparing to that of the cost of BI. On the other hand the cost of BI has great effect on increasing NR.

In Fig. 5 the retrieval time of BI increases at point A and B. This is because of the height of BI increases at point A and to access the height the retrieval time increases. After that the cost is nearly constant for BI from point A to B. This is because though NR increases but as the height remains the same the retrieval cost for BI is constant. At B the cost increases again because the h_B increases again for increasing NR.

From the cost analysis and experiment result the conclusion that can be drawn is that although the height of BI has very negligible effect for storage cost but it has great effect retrieval performance.

CONCLUSION

In this study, an index structure using variants of B tree is introduced and evaluated. The cost analysis and experimental results show that better performance for insertion and retrieval comparing with the performance of

conventional B^+ tree. Although the discussions and experimental results have focused placing the head on main memory but the head can also be stored on secondary storage. But before doing any operation, the head should be faced from the storage and construct the list head for getting the desired performance. Hence these can also be used to improve both the I/O performance and the memory performance of the databases. Because the size of an index node residing in disk is typically a disk page (normally 4 or 8 KB) hence the fan out is very small for large size key values but the fan out in head of CDL with main memory index is much larger than the index node residing in disk. This gives the benefits of using the scheme even wider nodes for searching. The range key query performance shown in this paper is applied having one page (8 KB) to accommodate a single leaf node to a page and this shows to have a significant benefit. However, it is shown that the main memory performance is important even for disk-resident databases, so it would be interesting to apply the CDL scheme in wide database application areas.

REFERENCES

Bayer, R. and M. McCreight, 1972. Organization and maintenance of large ordered indexes. Acta Informatica, 1: 173-189.

Bayer, R. and K. Unterauer, 1977. Prefix B-trees. ACM Trans. Database Syst., 2: 11-26.

Bertino, E. and W. Kim, 1989. Indexing techniques for queries on nested objects. IEEE Trans. Knowledge Data Eng., 1: 196-214.

Chen, S., P.B. Gibbons and T.C. Mowry, 2001. Improving index performance through prefetching. ACM SIGMOD Record, 30: 235-246.

Comer, D., 1979. The ubiquitous B-tree. ACM Comput. Surveys, 11: 121-137.

Eugene, I.C., D.C. Souripriya, F.S. Jagannath, A.M. Yalamanchi and J.K. Ramkumar, 2001. B^+ tree indexes with hybrid row identifiers in oracle8i. Proceeding of 17th International Conference on Data Engineering (ICDE, 01), April 02-06, IEEE Computer Society Washington, DC., USA., pp: 341-348.

Evangelidis, G., D. Lomet and B. Salzberg, 1997. The hB^{δ}-tree: A multi-attribute index supporting concurrency, recovery and node consolidation. VLDB J., 6: 1-25.

Giinther, O. and J. Bilmes, 1991. Tree-based access methods for spatial databases: Implementation and performance evaluation. IEEE Trans. Knowledge Data Eng., 3: 342-356.

Gupta, H., V. Harinarayan, A. Rajaraman and J.D. Ullman, 1997. Index selection for OLAP. Proceeding of International Conference on Data Engineering (ICDE' 97), April 7-11, Birmingham, UK., pp: 208-219.

Hasan, K.M.A., M. Kuroda, N. Azuma, T. Tsuji and K. Higuchi, 2005. An extendible array based implementation of relational tables for multidimensional databases. Proceeding of International Conference on Data Warehousing and Knowledge Discovery (DAWAK'05), August 22-26, Danemark, pp: 233-242.

Hasan, K.M.A., T. Tsuji and K. Higuchi, 2006. A parallel implementation scheme of relational tables based on multidimensional extendible array. Int. J. Data Warehousing Mining, 2: 66-85.

Knuth, D.E., 1998. The Art of Computer Programming, Vol. 3, Sorting and Searching. 2nd Edn. Addison Wesley, Reading, MA .

Lomet, D. and B. Salzberg, 1990. The hB-Tree: A multiattribute indexing method with good guaranteed performance. ACM Trans. Database Syst., 15: 625-658.

Rao, J. and K.A. Ross, 1999. Cache conscious indexing for decision-support in main memory. Proceeding of the Very Large Databases (VLDB'99), December 1, pp: 78-89.

Rao, J. and K.A. Ross, 2000. Making B^+ tree cache conscious in main memory. ACM SIGMOD Record, 29: 475-486.

Taniar, D. and J.W. Rahayu, 2002. A taxonomy of indexing schemes for parallel database systems. Distributed and Parallel Databases, 12: 73-106.

Taniar, D. and J.W. Rahayu, 2004. Global parallel indexing for multi-processors database systems. Inform. Sci., 165: 103-127.