CONCURRENCY CONTROL IN B^+-TREES DATABASES USING PREPARATORY OPERATIONS

Y. Mond and Y. Raz

Technion - Israel Institute of Technology
Haifa 32000, Israel

ABSTRACT

A simple concurrency control mechanism for B^+-trees Databases is introduced. It is based on early splitting and catenating of nodes during a process, which are candidates for such operations later on in that process. These early operations enable to lock only the current node in the process and its father node, and hence to increase the degree of concurrency relatively to other known mechanisms.

1. INTRODUCTION

B^+-trees are considered standard organization for indexes in database systems. They enable both random and sequential access, and thanks to their balancing mechanism, insert/delete operations are not expensive to carry out.

In a multi-user environment, where the database is maintained as a tree in general, and a B^+-tree in particular, several problems arise concerning simultaneous multi-access to the database.

One of the solutions to the problem of concurrency control, particularly in trees, is the use of locking techniques. Samadi [SAMD76], Bayer and Schkolnick [BAYR77], Miller and Snyder [MILR78] and King and Collmeyer [KING73] introduced several locking mechanisms. When search trees are concerned, every process locks all or some of the nodes on his path in the tree, according to the concurrency control mechanism which exists in the system.

Updaters, while inserting or deleting in a B^+-tree might cause changes to the structure of the tree. Thus it is common to check the "safety" of every node on the path. A node is considered "safe" if inserting a key into it will not end in its splitting and if deleting a key from it will not cause its merging with its neighbor [BAYR77]. If the node is unsafe, it will be locked by the process. Thus a chain of locked nodes will have to be kept for every updater, and the level of concurrency in the system might get lower.

The idea of guaranteeing a path of safe nodes for a simpler concurrency control is mentioned in Guibas & Sedgewick [GUIB78]. Keshet [KESH81] suggested the idea of immediate splitting or merging of unsafe nodes in order to avoid long chains of locks. By making these preparatory operations only a pair of locks has to be kept: on the current node and on its father node. Thus the portion of the tree locked by the process is getting smaller as the process advances. This idea was implemented and investigated by Y. Mond [MOND84].

In Section 2, the motivation behind the proposed mechanism is discussed. In Section 3 a modified B^+-tree is introduced and its properties are examined. In Section 4 a simple concurrency control mechanism taking advantage of the properties of the modified B^+-tree is introduced. In Section 5 some conclusions are given.

2. B^+-TREES DATABASES IN A MULTI-USER TRANSACTION ORIENTED ENVIRONMENT

A transaction is a collection of read or update processes, each accessing a data item. Read processes do not change the structure of the tree, while update processes do. Although it consists of different processes, the effect of the transaction on the database state should be like that of an atomic operation. The concurrency control system should control the concurrent execution, so that the computer resources will be used as efficiently as possible while preserving the atomicity. In a system based on B^+-trees (see [KNUTH73]) the data, the targets of the transaction processes, which are identified by keys, lie in the leaves, while the internal nodes are used to direct the processes. Every process starts looking for its target in the tree root, and advances through a path of internal nodes.

The B^+-tree is kept balanced by using node splitting or catenating for the rebalancing needed due to operations of inserting new keys into the structure or deleting existing keys from it.

The split and catenate operations may propagate from the leaves up to the root. In order that processes executing concurrently with changes being done in the tree (splitting or catenating)

do not loose their way to the target, mechanisms of locking are common for nodes that may be changed due to split or catenate operations. A process cannot access a node which is blocked for him. Using such mechanisms, all the nodes, from the leaf and up to the highest point, the "critical point" [MILR78], where propagation of the changes can reach, should be locked. In the worst case, the whole tree may be blocked, even though the changes take place only in part of the tree, i.e., the degree of concurrency gets much lower even where this is not really necessary. Thus, it is clear why a solution which locks a smaller number of nodes and lower nodes in the tree is desired.

The idea of preparatory operations appears in [GUIB78], where new algorithms are introduced which update and re-balance a search tree in one pass, on their way down to the leaf. Thus, when the desired data item is found, the appropriate operation can be immediately completed.

A possible disadvantage of the preparatory operations mechanism is that a certain amount of overhead will be needed due to the preparatory split/catenate. On the other hand, it seems that the level of concurrency achieved by such a mechanism is rather high. In a transaction driven system, where operations on the actual data at the leaves level of the B^+-tree may be delayed for a considerable amount of time by the concurrency control protocol, this advantage may be even bigger.

3. THE PO-B^+-TREE

In order to carry out the idea of preparatory operations, a minor modification in the definition of the B^+-tree is necessary. In the modified B^+-tree called "preparatory operations B^+-tree" (PO-B^+-tree) the number of keys in a node may be between $n-1$ and $2n+1$ rather than n and $2n$ in the usual B^+-tree, where n is the rate of the tree. The reason for this is the following: If n is the threshold for merging, then by merging an n-size internal node with a neighbour which also has size n, we get a $2n+1$ size node, since by the usual merging algorithm the separating key of the two merged nodes is also contained in the resulting node. By splitting a $2n$-size internal node ($2n$ is the threshold for splitting), we get n and $n-1$ size nodes, while one key is moved to the father of the splitted node.

Thus, let us define a PO-B^+-tree as follows:

Definition 1: PO-B^+-tree of rate n is a tree satisfying the following properties:
(a) Every node contains $2n+1$ keys at the most (has $2n+2$ sons at the most);
(b) Every node, except for the root, contains at least $n-1$ keys (has at least n sons);
(c) The root contains at least one key;
(d) All the leaves lie on the same level, and contain all the keyed data;
(e) A node which is not a leaf and contains k keys will have $k+1$ sons.

Accordingly, the definition of the operations on PO-B^+-trees will be the following:

Definition 2: PO-insert and PO-delete are the usual insert and delete operations for B^+-tree with the addition of preparatory splitting or catenating while passing during the operation a node with not less than $2n$ or not more than n keys respectively. (The root is an exception and it is left unchanged while being the current node in the case of PO-delete.)

Note, that a node is unsafe for insertion or for deletion if it has $n-1$ or $2n+1$ keys respectively.

Using the known properties and the usual transformations (for insertion and deletion) of B^+-trees, we can summarize now the relationship between the structure and operations by the following theorem.

Theorem 1: PO-B^+-tree properties are preserved under PO-insert and PO-delete.

The concurrency control mechanism is based on a property of the PO-insert and PO-delete described as follows:

Theorem 2: During a PO-insert or PO-delete operation, while accessing a node on the path to the desired leaf, there is no unsafe node among the nodes already passed.

Proof: It is proved by showing, using induction on the node location on the path, that the father node of the accessed one can gain a key in the case of PO-insert or loose a key in the case of PO-delete, remaining safe.

Case 1: PO-insert.

If the root has $2n$ or $2n+1$ keys it is split into $n-1$ and n key nodes or into two n key nodes respectively. A new root with one key is then created. Whenever a $2n$ or $2n+1$ key node is encountered, a preparatory splitting is invoked. Thus, when the current node is an internal node which contains $2n$ or $2n+1$ keys, it is split, creating two nodes, one with n keys and the other with $n-1$ or n respectively.

(Remark: When the current node is a leaf node which contains $2n$ keys, it is split, and the two resulting nodes contain n keys each.)
The father node gains one key and can have $2n$ keys at most, since by the induction hypothesis, after the previous step, it had $2n-1$ keys at most.

If the current node does not split it has $2n-1$ keys at most.

Thus, the father node of the next step (the current of this step) can gain a key remaining safe.

Case 2: PO-delete

If the current node has no more than n keys

and it is a son of the root which has only one key there are two possibilities: If the other son of the root has more than n keys, keys are shifted without catenation. If it has no more than n keys, the sons are merged becoming the new root with 2n-1 or 2n or 2n+1 keys. Whenever an n or n-1 key node (which is not the root) is encountered, a preparatory catenating is invoked. Thus, when the current node is an internal node which contains n or n-1 keys, it is catenated with its neighbour if the latter has n keys or n-1 keys. The resulting node has 2n+1 or 2n or 2n-1 keys. (One key is coming from the father node.) If the neighbour has more than n keys, keys are transferred from the neighbour without catenating.

(Remark: When the current node is a leaf node which contains n keys, it is catenated with its neighbour if the latter has n keys, and the resulting node contains 2n keys. If the neighbour has more than n keys, keys are shifted without catenating.)

In case of catenating, the father node looses one key and has at least n keys, since by the induction hypothesis, after the previous step, it had at least n+1 keys.

If catenating does not occur the current node has at least n+1 keys.

Thus, the father node of the next step (the current of this step) can lose a key remaining safe.

From the two cases above, we conclude that each node already passed was either safe in the first place or was made safe by the appropriate preparatory operation. □

The implications concerning the tree update operations are the following:

Corollary: During PO-insert or PO-delete, after updating the appropriate leaf the operation has been completed. (There is no need to update any other node as can happen during the usual insert or delete.)

4. A CONCURRENCY CONTROL MECHANISM FOR THE PO-B$^+$-TREE

The concurrency control mechanism is based on two kinds of locks - a lock for operations which do not change the database structure (read process and in-place update process) and a lock for operations which do change the structure (PO-insert and PO-delete). These locks will be referred to as a "read-lock" and "update-lock" respectively.

A read-lock blocks update-lock processes, but enables read-lock processes to access the node. An update-lock blocks any other process.

For each process the locks are applied according to the following rules:

1. Any process starts by locking the root with the appropriate lock (read-lock for update-lock) and accessing it (provided the process is not blocked by a lock already on the root).
2. During any process, while accessing a node on the path to the desired leaf, the node and its father are locked by the appropriate lock; before accessing the next node on the path, the father is unlocked.

The following theorem states the correctness of the described mechanism referred to as the Preparatory Operation Mechanism (POM).

Theorem 3: The POM preserves the PO-B$^+$-tree structure while running processes concurrently, for any time when no node update operation is being performed; every process "sees" a consistent tree.

Proof: It is sufficient to show that:
(1) when a node is being changed it is accessed by the changing process only, and
(2) any node change preserves the PO-B$^+$-tree constraints.

During an interaction of two read-lock processes no blocking is necessary, since there is no change in the structure.
A read-lock blocks update-locks processes, and hence no node change can occur.
An update-lock process blocks any other process from accessing the current node or its father. Hence, while changing them that process is the only process which can access them. A neighbour of the current node may also be changed. Before the process accesses the neighbour it should be free of any lock (possibly of another process). However, after it has been accessed by the process, no other process can access it, since its common father with the current node is locked. The upper nodes on the path are safe by Theorem 2, and may not be changed by that process. Before accessing the next node on the path, all the possible changes in the neighbour and the father have been completed for that process and hence the father can be unlocked.

Since each such node change preserves the PO-B$^+$-tree constraints by definition, and since update-lock processes do not mix on any node because of the locks, the constraints are preserved and each process "sees" a portion of the tree which gives the correct information to navigate it to the target data item. □

The overall concurrency control mechanism is a combination of the above mechanism and a protocol for synchronizing transactions in the system. Such a protocol should be applied only for the leaves of the PO-B$^+$-tree (the data level), and since it is independent of the above mechanism any protocol can be used.

The above mechanism by itself cannot introduce a deadlock, because it treates different processes independently, and since the tree structure induces a partial order on the blocked processes. However, the protocol for the trans-

actions may cause deadlocks (if it is not deadlock-free) which should be taken care of.

5. CONCLUSION

The concurrency control mechanism described enables a higher degree of concurrency relatively to mechanisms keeping longer chains of locks for processes which change the structure of the tree, since the longer the chain kept is, the larger is the blocked portion of the tree.

The preparatory operations introduce some overhead by increasing the number of operations performed upon the tree, but as the rate of the tree increases the relative number of unsafe nodes in the tree is reduced, and hence this overhead becomes small.

The intuitive advantages of the mechanism were verified by an implementation in a simulation system for a PO-B^+-tree database system with a single CPU and multiple I/O channels [MOND84]. The mechanism was also implemented successfully together with the 2PL-protocol in a database management system by J. Navon [NAVN84].

REFERENCES

[BAYR77] Bayer, R. and Schkolnick, M.: "Concurrency of Operations on B-Trees", Acta Informatica, 9, pp. 1-21 (1977).

[GUIB78] Guibas, L. and Sedgewick, R.: "A Dichromatic Framework for Balanced Trees", in Proc. of the 19th Symp. on Foundation of Computer Science, pp. 8-21, (1978).

[KESH81] Keshet, Y.: "Concurrency Control Problems in B^+-Tree Databases", The Technion-IIT, M.Sc. thesis (1981).

[KING73] King, P.F. and Collmeyer, A.J.: "Database Sharing - An Efficient Mechanism for Supporting Concurrent Processes", AFPIS National Computer Conference, pp. 271-275 (1973).

[KNUT73] Knuth, D.: "The Art of Computer Programming", B-Trees, Vol. 3, pp. 473-480 (1973).

[MILR78] Miller, R.E. and Snyder, L.: "Multiple Access to B-Trees" (preliminary version), Proc. of the 1978 Conf. on Information Sciences and Systems, J. Hopkins Univ., Baltimore, Maryland (1978).

[MOND84] Mond, Y.: "Concurrency Control in B^+-Trees Databases Based on Preparation Operations", The Technion-IIT, M.Sc. thesis (1984).

[NAVN84] Navon, J.: "Concurrency and Recovery for a Simple Database System", The Technion-IIT, M.Sc. thesis (1984).

[SAMD76] Samadi, B.: "B-Trees in Systems with Multiple Users", Infor. Processing Letters, 5,4 pp. 107-112 (1976).