Fully Persistent B$^+$-trees

Sitaram Lanka*
Computer Science Department
The Pennsylvania State University
University Park, PA 16802
lanka@cs.psu.edu

Eric Mays
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
emays@ibm.com

ABSTRACT

In this paper, we investigate efficiently maintaining multiple versions of B$^+$-trees. We propose two main schemes: *the fat field* method and *the pure version block* method. The former method is suitable for range queries or whenever lookup on a released version is an important issue. The latter method is more space efficient, and is suitable when the total number of keys from all the versions are small.

1 Introduction

For many applications, such as CAD and CASE systems, databases are required to provide support for version maintenance [5] [1]. That is, the update and retrieval of objects (or records) in the database are performed with respect to a particular version of the database. In this paper, we investigate efficient retrieval of objects in versioned databases using versioned B$^+$-trees. The techniques we develop here are oriented towards index maintenance within a database system which supports versioning of objects, so that query and update operations may be more efficiently supported.

We have been investigating database support for the development of large shared knowledge bases by a group of cooperating knowledge engineers. In a previous paper [6] we have described a storage mechanism for knowledge bases which is based on a large number of small versioned objects. In such a scheme, it is necessary to efficiently retrieve and update any version of the knowledge base during the development phase. To support efficient retrieval on any given version of the knowledge base, we proposed an indexing mechanism which reduces the number of block accesses required to resolve version identifiers. That method is extremely costly in terms of space, and is intended to be supported for only a few selected versions of the knowledge base. The versioned B$^+$-trees described in this paper grew out of our desire to obtain comparable retrieval efficiency on all versions of the knowledge base. The technique of versioned B$^+$-trees is general, however, and is applicable to other applications of versioned databases as well. In this paper, we are concerned only with the maintenance of versioned B$^+$-trees, and do not address any specific applications.

We adopt a model of versioned databases in which the versions form a partial order. We maintain this partial order information in a version graph. Any version may have an additional successor version created at any time. We do not allow destructive updates into versions. That is, a version is created, updates are applied with respect to that version, and the version is committed. Once a version is committed, updates may no longer be applied to that version. Since it is desirable to merge versions, the version graph forms a directed acyclic graph. The versioned B$^+$-tree techniques we develop in this paper fully support this model of versioning. Our work on making B$^+$-tree fully persistent has been inspired by work on making data structures fully persistent [3]. We have adopted the term *"fat node"* from this paper. We have also adopted the B$^+$-trees formalism described in [7] because of its simplicity.

This paper is structured as follows. Following a brief description of B$^+$-trees, we introduce the fat node method and describe the update and retrieval operations for this method. We then introduce an extension of the fat node technique which incorporates fat fields to improve on the space utilization. An alternative technique based on pure version blocks is described. Finally, we discuss the relative benefits of these techniques.

*Part of this work was performed while the author was visiting IBM T.J. Watson Research Center, Yorktown Heights.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1991 ACM 0-89791-425-2/91/0005/0426...$1.50

2 B^+-trees

A B^+-tree [2] [4] [7] is a generalization of the idea of building an index on top of a data file. We assume that the data file is packed tightly on several blocks with records in no particular order. The leaves of the B^+-tree contain pointers to the records of the data file. As in [7] we assume the maximum number of records in an index block of the B^+-tree to be an odd integer $(2d-1) \geq 3$. That is, an index block contains at least d records and at most (2d-1) records. The records in the leaf blocks are sorted by their key values and each record is a pair <k, p> where k is the key value of a data file record and p is a pointer to the data file record. The second level index is similar to the first level index except that p now points to a block in the first level index. Similarly, the other index levels are defined. Except for the root node all other nodes are required to have at least d records. The key value of the first record in an index block is omitted to save space. Retrieving a record for a particular key value requires at most $(1 + \log_d n)$ block accesses, where the data file has n records.

If insertion of a record into a leaf block B_1 would require B_1 to have more than (2d-1) records, then a new block B_2 is acquired, and the records of B_1 are divided into two groups of d records each and placed in B_1 and B_2. This is referred to as *splitting a block*. Similarly, if deleting a record from a leaf block B_1 would cause B_1 to have (d-1) records then a sibling of B_1, say B_2, which is immediately to the left or right of B_1 is combined with B_1. If B_2 has exactly d records then combining B_1 and B_2 will have (2d-1) records and they are placed in B_2 (and B_1 is marked as being empty). If B_2 has more than d records then the records of B_1 and B_2 are evenly distributed between B_1 and B_2 such that each of them have at least d records. This is referred to as *merging blocks*. For both the split and merge operations the parent index block must be updated. The details of the update operations on a B^+-tree are as described in [7].

3 The Fat Node Method

A standard B^+-tree data structure is referred to as an **ephemeral B^+-tree** because an update to it destroys the old values and only the new values are retained. In this section, we describe how to convert an ephemeral B^+-tree into one where none of the old values of the B^+-tree are lost due to incoming updates. This is achieved by maintaining different versions of the B^+-tree such that when a version of the B^+-tree is updated, it gives rise to a new version of the B^+-tree. Such a B^+-tree is referred to as **fully persistent**, because we maintain multiple versions of a B^+-tree such that any version can be accessed or updated. We achieve full-persistence by capturing the changes made to a node in the B^+-tree at the node itself without throwing away any of the old values. As updates come in, information at a node grows, and hence the nodes are referred to as **fat nodes**. We have taken this phrase from [3] where they have studied making data structures fully persistent. There is a one-to-one correspondence between a node in an ephemeral B^+-tree and a fat node in its corresponding fully persistent B^+-tree. We introduce a data structure called a **version block** to convert a node in an ephemeral B^+-tree into a fat node. A node in an ephemeral B^+-tree consists of an index block, whereas a fat node in a fully-persistent B^+-tree consists of a version block and one or more index blocks. A version block is a list of ordered pairs of a version identifier and a pointer to an index block, where a version identifier uniquely identifies a version of the B^+-tree. A fat node encapsulates all the changes made to a node. Within a fat node, an entry in the version block along with the associated index block corresponds to a version of the node. Therefore, whenever a fat node is to be updated, a new entry in made in its version block and a new index block pointed to by this entry holds the updated information. Note that in order to be space efficient, a new version of the B^+-tree holds only the incremental changes made to the old version. Hence, to create a new version of the B^+-tree, it is not necessary to update every fat node in the old version. Only those fat nodes which have been changed from the previous versions need be updated. Another important point to note is that a new version need not be created for each update. That is, a version may incorporate a set of updates, only the first of which requires creation of a version. Note that any update to a version may require creation of a version block entry and index block.

We allow any version of the B^+-tree to be updated, therefore, the versions of a B^+-tree form a partial order. This partial order information is maintained as a DAG[1] called a **version graph**. A version graph is maintained in conjunction with the versioned B^+-tree data structure. The nodes of the version graph correspond to version identifiers and an edge from node i to node j exists if version j is obtained by updating version i.

Since different versions of the B^+-tree could be rooted at separate nodes (see Figure 1) we maintain an additional data structure called *root**, which is similar to a version block. The entries in the root* version block contain pointers to the roots of the B^+-tree. Whenever a new root node for version i is created we store the pair <i, p> in root* where p points to the root of the ith version of the B^+-tree.

Notation and Definitions We adopt the following no-

[1] A DAG allows us to represent any partial order information. For instance, it is general enough to capture the merge of different versions.

tation: i, j denote version identifiers; B, B_1, ..., B_n denote secondary storage blocks; and L, P denote fat nodes. A version i only holds the incremental changes from its previous version. Therefore, we need a mechanism to collect all the nodes in the version graph from the root to the node i, as each of these nodes could potentially hold information corresponding to version i. We call such a collection as the **ancestor set** of version i, denoted as **Anc(i)**. An ancestor set is defined on a version graph and is the union of the version identifiers from all the paths from the root to node i. A key k_1 **covers** k_2 if $k_1 \leq k_2$ and $\nexists\, k_3$ such that $k_3 > k_1$ and $k_3 \leq k_2$. A block B **corresponds to** a version i in L if in the version block of a fat node L, there is an entry <i, p> such that p points B. Sometimes when the fat node is obvious, in that case we can drop the qualifier "in L". B_1 and B_2 are **sibling blocks** in version i if the fat nodes to which B_1 and B_2 belong have the same parent fat node, and B_1 and B_2 correspond to versions which are maximal in Anc(i).

Lookup To retrieve a record with key k_1 from version i start at root* and at each fat node choose an entry in its version block such that the version identifier is maximal in Anc(i). If the block B corresponding to this entry is, (1) a non-leaf index block, find a record in B whose key that covers k_1, and go to the fat node pointed by that record; (2) a leaf index block, find the record in B with key value k_1 and go to the data file block pointed by that record.

Lookup takes at most $(2 + 2 \log_d n)$ block accesses where n is the number of records in that version. We assume that a version block fits on one physical block. The path length from the root to a leaf node is at most $(\log_d n)$, and at each fat node we require one block access to get its version block and one block access for its index block. To initiate lookup (access the appropriate root via root*) requires at most one block access and, once at a leaf index block, a block access is required to access the record from the data file.

3.1 Update Operations

Insert operation Let us consider inserting a record with a key value k_1 in version i to yield version j. Perform lookup to find block B which corresponds to version i in the leaf fat node L. Copy B into B_1 and insert the new record into B_1; B_1 corresponds to j in L. If B_1 has less than 2d records, we are done. On the other hand, if B_1 has 2d records then B_1 must be split. This is carried out by dividing the records in B_1 evenly between B_1 and a new block B_2, where B_2 corresponds to j in L_1. The first d records are placed in B_1. The remaining d records are placed in B_2. At the parent fat node of L it must be reflected that L and L_1 are siblings. This can be thought of as an insert at the parent fat node of L, and the insert procedure described above is recursively applied. In the process if any block of the ancestors of L has 2d records then that block must also be split. Thus, the effect of an insertion can ripple up the tree to the root, in which case the root is split and a new root is created.

Delete operation Consider deleting a record with a key value k_1 in version i to yield version j. Perform lookup to find block B which corresponds to version i in the leaf fat node L. Copy B into B_1 and delete the record with key k_1 from B_1; B_1 corresponds to j in L. If B_1 has at least d records and k_1 is not the leftmost key in B, we are done. If B_1 has at least d records and k_1 is the leftmost key in B then at the parent fat node of L, say P, the new left most key value of B_1 must be reflected in P. If k_1 is the leftmost key in P then the parent of P must be updated recursively.

On the other hand, if after deleting k_1, B_1 has less than d records then it must be merged with its sibling block, say B_2 which corresponds to j in L_1. The merge is carried out in one of the following two ways.

1. If B_2 has more than d records then the records of B_1 and B_2 are evenly distributed between B_1 and a new block B_3, which corresponds to j in L_1, such that each of them have at least d records. As described above, the parent fat node of B_1 and B_3 should be changed to account for the leftmost keys of B_1 and B_3.

2. If B_2 has exactly d records then the records of B_1 and B_2 are merged to fit on B_1. After the merge, the parent fat node of L_1 must reflect that L_1 is no longer its child in version j. As a result, the parent of L_1 may have fewer than d records, and thus the process applies recursively. The deletion could ripple all the way up to the root of the tree, in which case the children of the root may be combined to form a new root corresponding to version j.

3.2 An Example

Figure 1 is a fully persistent B^+-tree built using the fat node method. Version 1 corresponds to an ephemeral B^+-tree, version 2 corresponds to inserting a record with the key value 32 in version 1, and version 3 corresponds to deleting a record with the key value 64 from version 2. The example has been adapted from [7] and d is assumed to be 2.

The insertion is carried out in fat node 3. Upon insertion that block has 2d records and must be split. As a result, a new fat node 12 is created, and its parent fat node 9 must be updated. After insertion at fat node 9, the block contains 2d records and must be split, thus creating fat node 13. In turn, its parent fat node 11 must be updated, which causes the corresponding block

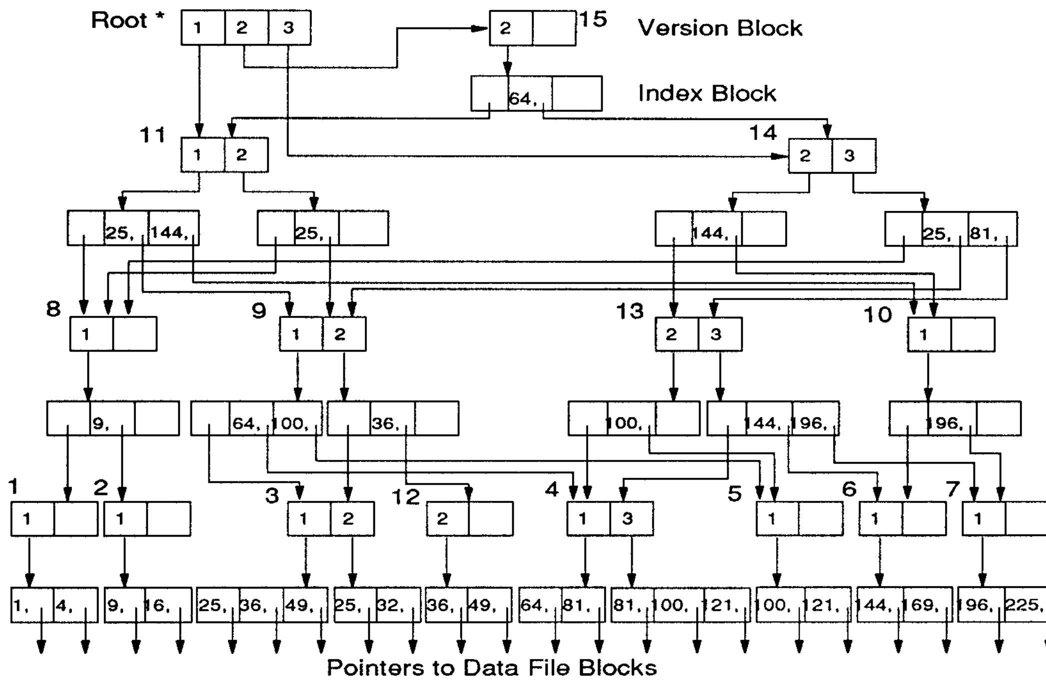

Figure 1: The Fat Node Method

to contain 2d records. Splitting that block creates fat node 14. Finally, a new fat node 15 is created, which is the root of the B^+-tree for version 2.

The deletion is carried out in fat node 4. This causes the block to contain less than d records and is merged with its sibling block (from fat node 5). The parent fat node 13 must reflect that fat node 5 is no longer its child in version 3. This makes the block in fat node 13 to have fewer than d records. After merging with a sibling block (from fat node 10), the parent node 14 is updated to reflect that fat node 10 is no longer its child. This results in the block at fat node 14 having fewer than d records, so the block in fat node 11 is merged into it. Now, fat node 14 is the root of the B^+-tree with respect to version 3. Also, note that the key 64 was the leftmost key on the block corresponding to version 2 in fat node 4, and this must be reflected as the new leftmost key in its parent fat node 13. Since 64 was also the left most key corresponding to version 2 at fat node 13, we must in turn reflect this in its parent fat node 14.

Lookup of a record with the key value 81 with respect to version 2 is carried out as follows. Anc(2) = {1, 2}; at fat node 15 we choose the block corresponding to version 2 as it is maximal in Anc(2). The key 64 covers 81, and at fat node 14 we choose the block pointed to by version 2. The first key there covers 81 and at fat node 13 again the block corresponding to version 2 is chosen. The first key there covers 81 and at fat node 4, the block corresponding to version 1 is chosen as it is maximal in Anc(2). This is a leaf index block, and the record with the key value 81 has the pointer to the record in the data file.

4 Fat Field Method

In the fat node method, whenever a fat node is updated in a new version a new block is required to store its effects. To improve space utilization, the space remaining in a block corresponding to a version can be used to hold records from its descendant versions. Therefore, it is not always necessary to use a new block to hold the records of a new version. To facilitate such an efficient encoding, we propose the following enhancement to the data structure described in section 3.

Each index block record is a four tuple <in, out, k, p> where *in* indicates the version in which the record is inserted, *out* indicates the versions from which the record is deleted, *k* is the key value of the record and *p* is a pointer to either the record in the data file or a fat node. Note that since the versions form a partial order, a record can be deleted from more than one version, hence, *out* denotes a set of versions. We use * to denote an empty *out* field. As a record now has more fields, we assume the maximum number of records in an index block to be an odd integer $(2d' - 1) \geq 3$. That is, an index block must have at least d' records and at most $(2d'-1)$ records. The key value of the first record in an index block is omitted, to save space. Since fat fields are variable length records, d' must be chosen such that $(2d'-1)$ times the minimal length of a fat field is less than or equal to the physical block length. Note that a

block may be full, but yet hold less than (2d'-1) records because of variable length records.

Two conditions must hold in order to store the records of version j in the block which corresponds to version i: (a) the block corresponding to version i is not full, and (b) i is the maximal entry in the version block in Anc(j)[2].

Definitions The **accessible records** in a block for a given version i are those whose *in* field is a member of Anc(i) and whose *out* field has no element which belongs to Anc(i). If a block B holds records from a version then that version belongs to the **relevant version set** of B. A block B **is balanced** if for every version i in the relevant version set of B, the number of accessible records for i is at least d'. A fat node is balanced if every block in it is balanced, and a B^+-tree is balanced if every fat node in it is balanced.

Copy Operation Consider a situation where an insert causes a block to be full. Normally in this situation we would split that block. However, the fat field method stores records on a block which are no longer accessible from the version in which the insert was performed. That is, the block does not contain (2d' - 1) records accessible from the version. Splitting the block in this situation would result in an unbalanced B^+- tree (shown in lemma 4.1). The copy operation reclaims space by eliminating those records which are no longer accessible from a given version and by minimizing the size of the fat fields. The fat fields are reduced to a minimal size since they contain a null *out* field. Note that due to the earlier described constraint on the choice of d', the copy operation is guaranteed to create a block which can hold (2d' - 1) records. Consider inserting a record in version j into block B which corresponds to i, and let the insertion cause B to be full due to B containing records which are no longer accessible from j. We take advantage of this by copying only the accessible records from j in B into a new block B_1 such that B_1 corresponds to version j. In this case, B_1 would have less than 2d' records.

The copy operation from a version i on a block B consists of two steps: (1) copy the accessible records from i in B onto a new block B_1 such that B_1 corresponds to i and belongs to the fat node of B; note the relevant version set of B_1 is {i}. (2) perform garbage collection on block B. That is, erase the effects of insertion or deletion from version i in B. Inserted records are merely removed and from the *out* field of a deleted record remove i. Note that the effect of this is that i is removed from the relevant version set of B.

Split Operation

[2]This implies that j is a descendant of i, and updates to version j are allowed only after version i is committed.

Lemma 4.1: If the relevant version set of a block B is not a singleton set and B is over-full then splitting B results in an unbalanced block.

Proof: Case 1: B is over-full and has less than 2d' records, i.e., the *out* field in one or more records in B contain a number of version identifiers. Splitting B would result in a block being unbalanced.

Case 2: B is over-full and has 2d' records. Let B corresponds to version i, and let its relevant version set be {i, j}. B can hold at most (2d'-2) and at least d' records from version i. Suppose B has 2d' records from version i, it can be split in two ways: (a) B has d' records from i and B_1 has (d'-2) records from i, and relevant version set of B and B_1 is {i} and {i, j}, respectively, and (b) both B and B_1 have (d'-1) records from i and the relevant version set of both B and B_1 is {i, j}. At least a block in both (a) and (b) is unbalanced with respect to version i. The same argument holds if B has less than (2d' - 2) records (and up to d') records) from version i. □

As the above lemma suggests, rather than splitting a block B whenever it is over-full due to an update from version i, a copy operation is first performed on B from i to yield B_1. One of the following cases can occur: (i) B_1 is not over-full and it has less than 2d' records, (ii) B_1 is not over-full and it has 2d' records, and (iii) B_1 is still over-full and has 2d' records[3]. If it is (i) we are done. Observe that (ii) (and possibly even (i)) is possible because of the following property: after a copy operation on a block the *out* fields of all the records in it will be null. Lemma 4.2 shows that splitting B_1 in the case of (ii) and (iii) results in a balanced blocks.

Lemma 4.2: In case of (ii) and (iii) above splitting B_1 would leave the B^+-tree balanced.

Proof: From the copy operation we know that the relevant version set of B_1 is {i}, and that it contains only the accessible records from i. After splitting B_1 both the resulting blocks will trivially contain d' records each, and their relevant version set will be {i}. □

Merge Operation A delete operation can potentially unbalance a B^+-tree as a block can contain less than d' records with respect to a version. A block B is merged with its sibling block B_1 if B has less than d' accessible records from version i. It is meaningful to merge two blocks if both contain only the accessible records from the same version, i.e., their relevant version set is singleton. Therefore, prior to merging B and B_1, apply the copy operation on B and B_1 from i to yield B_2 and B_3, respectively. Now merge B_2 and B_3.

[3]If d' is appropriately chosen, case (iii) should not occur. However, further investigation (through experimentation) is called for to determine an appropriate d'.

Delaying Update of Index Blocks Whenever the leftmost key of an index block changes, it must be reflected in one of its ancestor index blocks. Recording such a change requires a new block. It is not necessary to immediately record such changes as delaying update of a non-leaf index block will still guarantee that the lookup procedure will find a record if the record exists[4]. However, such changes can be recorded at a later time when the copy operation is performed as it anyway requires a new block. Whenever, a copy operation is performed at a non-leaf index block check if all the keys in the resulting block are current, and if not bring them up-to-date. This costs nothing in terms of space, but, updating the leftmost keys is very costly in terms of time, and thus it is probably best to not perform the update.

Creating Fat Nodes on Demand So far we have assumed that every node on the B^+-tree is a fat node. It would be space efficient if we started out with an ephemeral B^+-tree (however, the records in the blocks are as described in the fat field case). The operations copy and split are the only ones that generate a new block, hence we focus on them. A copy operation always generates a new block, therefore, if a copy operation is applied to a non-fat node then the node is first converted to a fat node to accommodate the new block. In order to convert a non-fat node to a fat node we must first fetch the version identifier that must be associated with the non-fat node. This information is available at its parent node[5]. The split operation is applied only on blocks in a fat node as the copy operation precedes it. The result of a split operation is placed in two blocks. The block containing the first d' records is placed in the fat node of the split block. The block containing the latter d' records is stored as a non-fat node. We adopt this strategy in the update algorithm.

4.1 Update Operations

Insert operation Consider inserting a record with a key value k_1 in version i, to yield version j. Insert the record in block B which corresponds to i in a leaf fat node L. If B is full, it must first be copied to a new block B_1 which corresponds to version j. For the remainder of this description B refers to B_1 if a copy was performed. After the insert if B contains less than $2d'$ records, we are done. If B has $2d'$ records then it must be split. This is carried out by dividing the records in B evenly between B and B_2. The first d' records are placed in B, the remaining d' records are placed in B_2 which corresponds to j, and as pointed out in section 4.3.2 this need not be a fat node. L and B_2 are siblings and this must be reflected at the parent node of L. This can be thought of as an insert at the parent of L, and the insert procedure described above is recursively applied. The effect of the insertion can ripple up in the tree to the root, in which case, the root is split and a new root is created.

Delete Operation Consider deleting a record with a key value k_1 in version i, to yield version j. Delete the record from block B which corresponds to i in L. If B is full, it must first be copied to a new block B_1 which corresponds to version j. For the remainder of this description B refers to B_1 if a copy was performed. Mark the record as deleted from version j by making an entry in the *out* field. If B has at least d' accessible records from version j, we are done. On the other hand, after deleting k_1 if B has $(d'-1)$ accessible records from version j then it must be merged with its sibling block, say B_2. The merge of B and B_2 is carried out in one of two ways.

1. If B_2 has greater than d' accessible records from version j, the merge of B and B_2 is carried out by evenly distributing their records between B and B_2. Note that B or B_2 may become full and require copying. Each of them have at least d' records.

2. If B_2 has exactly d' accessible records from version j, the records of B and B_2 fit on one block, and are placed in B. Note that B may become full and require copying. As a result of the merge, the parent node of B_2 should reflect that B_2 is no longer its child in version j. This could cause the parent node to have less than d' accessible records and thus may ripple up to the root.

Lookup Lookup is similar to the one described in the case of the fat node method. To retrieve a record with key k_1 from version i, start at root* and at each fat node find an entry in its version block such that its version identifier is maximal in Anc(i). Now, if at a non-leaf index block, choose the record such that its *in* entry is maximal in Anc(i), no member of its *out* entry belongs to Anc(i) and whose key covers k_1. If at a leaf index block, choose the record such that its *in* entry is maximal in Anc(i), no member of its *out* entry belongs to Anc(i) and its key value is equal to k_1.

Lookup takes at most $(2 + \log_{d'} n + l \log_{d'} n)$ block accesses, where n is the number of records with respect to a given version. The path length from the root to

[4] Consider the out-of-date value of a leftmost key in a leaf index block be k_1 and let the new leftmost key be k_2. That is, a non-leaf index block holds k_1 instead of k_2. To successfully lookup k_2, we need to show that k_2 is covered by some key in the B^+-tree. Both k_1 and k_2 are in the B^+-tree, and for k_2 to replace k_1 it must be that k_2 is in the sub-tree of k_1, i.e., k_1 covers k_2.

[5] If its parent node is a non-fat node then we must visit the parent of its parent and so on. Ultimately, if we are at the root and it is a non-fat node then its version identifier is taken as the starting version identifier.

a leaf block is at most ($\log_{d'}$ n) block accesses and of which factor l of the nodes are fat nodes, $0 \leq l \leq 1$, so we may require ($l \log_{d'}$ n) additional block accesses to get the version blocks. Initiating lookup with respect to a version requires one block access and, once at a leaf index block, one block access is required to fetch the record from the data file.

4.2 An Example

Figure 2 is a fully persistent B$^+$-tree using the fat field method. Version 1 corresponds to an ephemeral B$^+$-tree (that is, initially all the nodes are non-fat nodes), version 2 is a successor of version 1 where a record with the key value 125 is inserted, and version 3 is a successor of version 1 where the following updates were performed: (1) insert a record with the key value 85, (2) delete the record with the key value 81, (3) delete the record with the key value 64, and (4) insert a record with the key 104. In this example we assume d' to be 2.

Handling the update that yields version 2 is straightforward. Let us consider the updates that yield version 3. Update (1) requires inserting a record with the key 85 in node 4. Update (2) requires an entry to be made in the *out* field of the record with the key 81, in node 4 such that it indicates that the record is deleted in version 3. Similarly, the update (3) requires an entry in the *out* field of the record with the key value 64, in node 4. Now, the block corresponding to node 4 is less than half full with respect to version 3 and it must be merged with its sibling block (from node 5). The copy operation from version 3 is performed on the blocks at node 4 and 5 (thus converting them to fat nodes) and the resulting blocks are merged. The merge fits in a single block (<3, *, 85, pointer>, <1, *, 100, pointer>, <1, *, 121, pointer>) and it is placed in fat node 4. Note that the blocks resulting at node 4 and 5 due to copying are now discarded. At node 7 an entry in the *out* field of the record with the key value 100 is made to account for the fact that node 5 is no longer its child in version 3. At this stage, note that in node 7 we do not reflect that 64 is no longer the leftmost key of the block corresponding to version 3, in fat node 4. The update (4) causes the block in fat node 4 to contain 2 d' records, hence it must be split. Performing the split requires a copy. The result of the split is placed in fat node 4 and in node 9. In node 7, <3, *, 104, pointer> is inserted to reflect that node 9 is its child in version 3. The insertion causes the block to be full and the copy operation with respect to version 3 results in ($2d'$-1) records and this is stored in a new block corresponding to version 3, in fat node 7. Note that since the leftmost keys are not updated in the non-leaf index blocks, the prior leftmost keys serve as cover for the new leftmost key.

Lookup of a record with the key 81 from version 1 is carried out as follows. Anc(1) = 1; at node 8 the key 25 covers 81. At fat node 7 we choose the block corresponding to version 1 as it is maximal in Anc(1), and the record with the key 64 covers 81. At fat node 4 we choose the block corresponding to version 1 and once at the block the record with the key 81 provides the pointer to the data file record.

5 Pure Version Block Method

In this scheme, the index blocks of a B$^+$-tree do not carry the versioning information. The B$^+$-tree that is built in this case is similar to the one built in the ephemeral case, except that the B$^+$-tree accounts for all the keys that appear in all the versions. A record of an index block has the same format as the record of an ephemeral B$^+$-tree index block (see Section 2). As the records of the B$^+$-tree do not contain any version information the leaves of the B$^+$-tree do not point to the records in a data file. Instead, an additional level of indirection is necessary to resolve the version information. Hence, a record in a leaf index block points to a block referred to as a **pure version block**, which in turn points to a record in the data file. A pure version block consists of a collection of variable length records with the format: (k, <i, p>$^+$) where k is a key value, i is a version identifier, p is a pointer to a record with a key k in version i, in the data file. Interpret this as, a record with a key value k can be modified in one or more versions and corresponding to each version it is modified in, there is pair – version identifier, pointer. The records in a pure version block are sorted by key value. The different versions form a partial order and this information is maintained in a version graph as in Section 3.

For example, if a record with a key k_1 is inserted in version i then the pure version block that contains it would have the record (k_1, <i, pointer>). Now, if k_1 is deleted in version j then the pure version block contains the record: (k_1, <i, pointer>, <j, *null*>), where *null* pointer indicates that it is deleted in version j and its descendants.

Splitting a Pure Version Block A pure version block is split whenever it is over-full, and since it contains variable length records, care must be taken not to split a record, otherwise a single record would be spread over two blocks. Hence, a pure version block is split along record boundaries. Splitting a pure version block B results in approximately two equal parts and they are placed in B and a new pure version block B_1. It must be ensured that some of the pointers from the leaf index block correctly point to B and B_1. Note that these changes do not propagate up the B$^+$-tree.

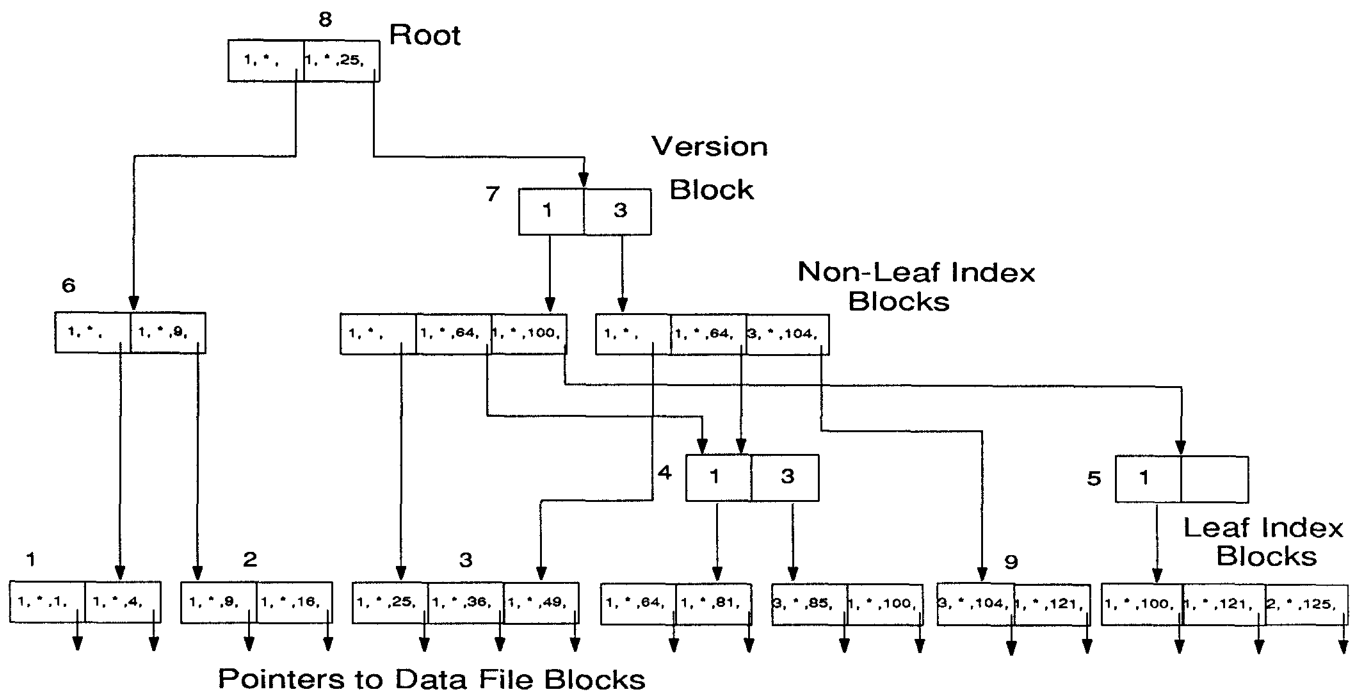

Figure 2: The Fat Field Method

5.1 Update Operations

Insert Operation Consider inserting a record with a key value k_1 in version i to yield version j. Two cases arise,

1. k_1 is a new key and does not appear in the B^+-tree. We have to insert k_1 into the B^+-tree as is done in the ephemeral case. The record of k_1 is placed in the same pure version block as the record of the key immediately to its left in the B^+-tree. If there is no key to its left then place it in the same pure version block as the record with the key immediately to its right in the B^+-tree. If an insert into a pure version block B causes it to be over-full, it must be split.

2. k_1 already appears in the B^+-tree. The B^+-tree need not be updated, but the record with the key k_1 in a pure version block must be updated. If the insertion causes the pure version block to be over-full, it must be split.

Delete Operation Consider deleting a record with a key value k_1 in version i to yield version j. Perform lookup to find the pure version block B_1 that contains the record with key k_1. Mark that record to be deleted in version j by inserting in it <j, *null*>. If the insertion causes B_1 to be over-full then it must be split, otherwise we are done.

Lookup Lookup of a record with a key k_1 from version i is performed on the B^+-tree as in the ephemeral case. The leaf index block contains a pointer to a pure version block B that holds the record. From B the record with key k_1 is chosen. Within that record choose the pair such that (1) its version identifier is maximal in Anc(i), and (2) its pointer field is not *null*.

Lookup takes at most $(2 + \log_d m)$ block accesses, where m is the total number of keys appearing in the entire data file with respect to all the versions. It takes at most $(\log_d m)$ block accesses to get to a leaf of the B^+-tree, one block access to fetch the pure version block and another to get the data file record.

5.2 An Example

In Figure 3 version 1 corresponds to an ephemeral B^+-tree, version 2 is a successor of version 1 where a record with the key value 125 is inserted, and version 3 is a successor of version 1 where the following updates were performed: (1) insert a record with the key value 85, (2) delete the record with the key value 81, (3) modify the record with the key 16. In this example we assume d to be 2.

The key 125 is a new key so it is inserted at index block 5. Its corresponding record is placed in the pure version block VB_5. This causes it to be over-full and is split into VB_6 and VB_7 which contain the records (81, <1, pointer>)(100, <1, pointer>) and (121, <1,

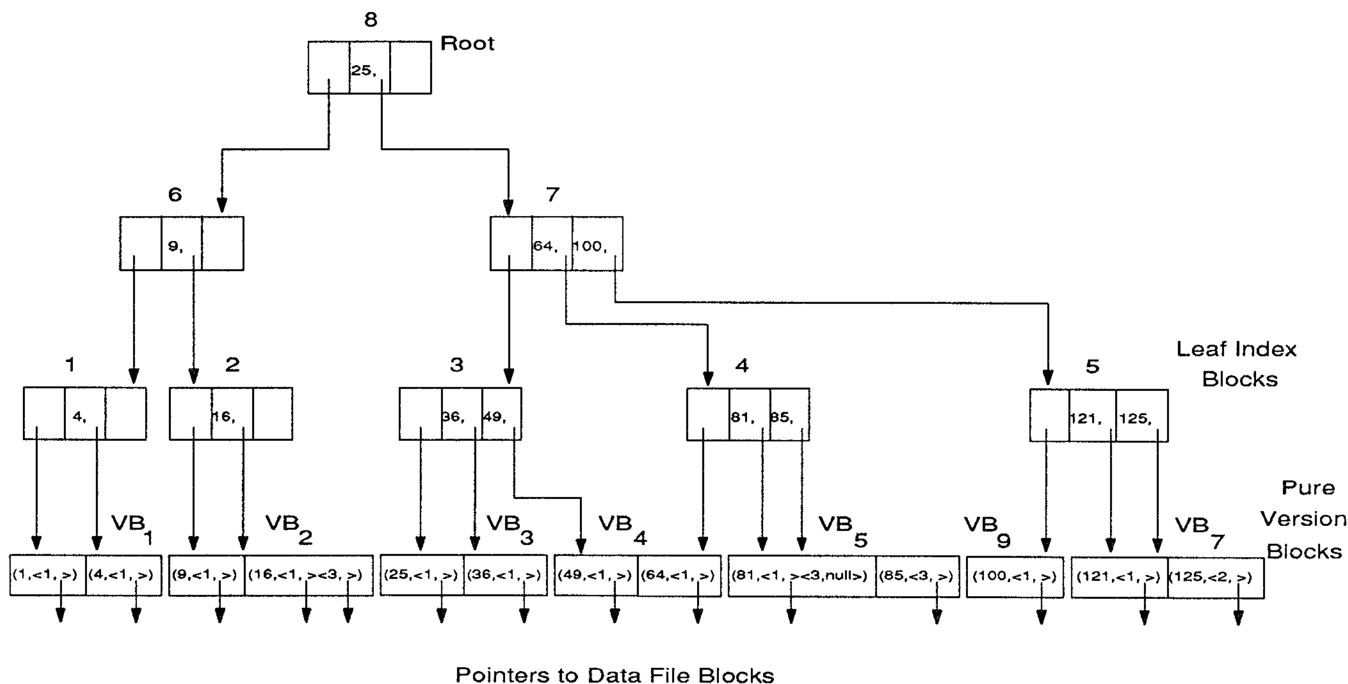

Figure 3: The Pure Version Block Method

pointer>)(125, <2, pointer>) respectively. In update (1) 85 is a new key and it is inserted in the leaf index block 4 and the pure version block record is placed in VB_6. For update (2) no changes are necessary on the index blocks. Its corresponding record in the pure version block VB_6 is updated to reflect that it is deleted in version 3. This causes VB_6 to be over-full and it is split into VB_8 and VB_9 which contain the records (81, <1, pointer>, <3, null>)(85, <3, pointer>) and (100, <1, pointer>) respectively. Update (3) again does not involve updating the index blocks. The record with the key 16 in the pure version block VB_2 is updated.

Lookup of a record with the key 16 from version 3 is carried out as follows. Anc(3) = {1, 3}; the lookup procedure on the index blocks yields the leaf index block 2, and the record with the key 16 points to VB_2. The record with the key value 16 in VB_2 is chosen, and within this record we choose the appropriate version by picking <3, pointer> as 3 is maximal in Anc(3), and the data file record pointed to by *pointer* is fetched.

6 Discussion

In this section we compare the fat field method with the pure version block method of maintaining multiple versions of a B^+-tree. The pure version block method is space efficient as the B^+-tree does not contain any version information, and as a consequence an index record is never duplicated. To perform lookup of a single record the fat field method requires at most $(2 + \log_{d'} n + l \log_{d'} n)$ block accesses and the pure version block method requires at most $(2 + \log_d m)$ block accesses. Assume $l = 1$, and $d \approx d'$, then (i) if $m \leq n^2$ the pure VB method requires fewer block accesses to perform a lookup over the fat field method, and (ii) if $m > n^2$ the fat field method is more efficient than pure version block method.

Aggregate Lookup Aggregate lookup is defined as the number of blocks required to lookup a set of r records with keys $k_1, \ldots, k_r$ with respect to a given version i.

Case 1: Upper Bound – each of the r records is in a different leaf index block. The fat field method requires at most $(1 + r \log_{d'} n + l r \log_{d'} n + r)$ block accesses. One block access to initiate lookup; since each record is on a different index block, traverse r times from the root to a leaf index block and at each node in this path an additional block access required to fetch the version block; and r block accesses to fetch the data file records. The pure version block method takes at most $(r \log_d m + 2r)$ block accesses. Once at a leaf index block we need one block access to fetch the pure version block and one block access to fetch the data file record. For example, suppose $n = 10^5$, $m = 10^6$, $r = 100$. For $l = 0.8$, the pure version block method requires fewer block accesses. On the other hand for $l = 0.4$ then the fat field method requires fewer block accesses. If r is much larger, say 1000, then the fat field method is very efficient.

Case 2: Range Queries – the r records are placed consecutively in the leaf index block but they need not be stored consecutively in the data file. The fat field method requires at most $(1 + \log_{d'} n + l \log_{d'} n + (\lceil r/d' \rceil -1) + l (\lceil r/d' \rceil - 1) + r)$ block accesses. r/d' represents the minimum number of leaf index blocks the r records fit on, and once at a leaf index block we need to fetch $(\lceil r/d' \rceil - 1)$ other leaf index blocks to visit all the r records at the leaf index blocks. The pure version block method requires at most $(\log_d m + (\lceil r/d \rceil - 1) + 2r)$ block accesses. Other than for very small r ($r \leq 2$) the fat field method is very efficient.

Lookup on a Released Version

A version of the B^+-tree could be chosen (because it constitutes the completion of a design phase or meets some such criteria) and *released* [5] to the user community at large. On such a version it would be beneficial to provide additional mechanisms such that lookup would be more efficient than would be possible otherwise. One such mechanism is to build an index for the released version[6]. The idea behind building the index is to bypass the information related to the other versions and make it possible to directly access the information pertinent to the released version. Thus, given the object corresponding to a fat node, the version block is bypassed and directly gain access to the pertinent record at that node. The details of constructing such an index mechanism is described in [6] and the lookup for a released version is almost the same as before with some minor variations.

We can build such an index only when there is a separation between the version information and the actual data that is being versioned. For this reason, such an index can be only built in the fat field method, and not in the pure version block method[7]. Hence, for a released version the lookup is considerably speeded up in the fat field method (as the version blocks need not be fetched) while it remains the same in the pure version block method. The fat field method requires at most $(2 + \log_{d'} n + b)$ block accesses for unit lookup; at most $(1 + \log_{d'} n + (\lceil r/d' \rceil - 1) + r + b)$ block accesses for range queries. In the above we have accounted for *b* block access to fetch the index for the released version.

We have not been able to discuss in this paper the implication of merging different versions of the B^+-tree. We will just note that the pure version block method has better worse case behavior than the fat field method.

7 Conclusions

Many applications of databases such as CAD and software development require version maintenance. In this paper we have studied the problem of efficiently maintaining multiple versions of B^+-trees such that any version of it can be accessed and updated. This is referred to as full persistence. We have mainly proposed two schemes called *the fat field* method and *the pure version block* method. The suitability of these methods depends on the nature of the application. For instance, the pure version block method is space efficient and performs lookup efficiently as long as the total number of keys from all the version is not too large (greater than the square of the number of keys in a single version). On the other hand, lookup with respect to a version in the fat field method is independent of the other versions. Hence, even if the number of versions increase, the performance does not degrade. Also, for range queries, and lookup on a released version is performed more efficiently in the fat field method.

Acknowledgments: We would like to thank Robert Weida and Bob Dionne for many insightful discussions, and James Wolfe for providing comments on an earlier draft of the paper.

REFERENCES

[1] H. Chou and W. Kim. A Unifying Framework for Version Control in a CAD environment. In *Proc. VLDB*, 1986.

[2] D. Comer. The Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[3] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[4] G.Held and M. Stonebraker. B-trees reexamined. *Communications of ACM*, 21(2):139–143, 1978.

[5] R.H. Katz and T.J. Lehman. Database support for version and alternatives of large design files. *IEEE Transactions on Software Engineering*, 1984.

[6] E. Mays, S. Lanka, B. Dionne, and R. Weida. A Persistent Store for Large Shared Knowledge Bases. *To appear in IEEE Transactions on Knowledge and Data Engineering*, 1991.

[7] J. D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 1. Computer Science Press, Rockville, MD, 1989.

[6] Note that such an index can itself be a B^+-tree.

[7] In a pure version block the the version information and the data are intertwined.