The BTR-Tree: Path-Defined Version-Range Splitting in a Branched and Temporal Structure

Linan Jiang[1], Betty Salzberg[2], David Lomet[3], and Manuel Barrena[4]

[1] Oracle Corp.
400 Oracle Parkway, Redwood Shores, CA 94065
linan.jiang@oracle.com
[2] College of Computer Science
Northeastern Univerity
Boston, MA 02115
salzberg@ccs.neu.edu
[3] Microsoft Research
One Microsoft Way, Building 9, Redmond, WA 98052
lomet@microsoft.com
[4] Universidad de Extremadura
Cáceres, Spain
barrena@unex.es

Abstract. There are applications which require the support of temporal data with *branched* time evolution, called *branched-and-temporal* data. In a branched-and-temporal database, both historic versions and current versions are allowed to be updated. We present an access method, the *BTR-Tree*, for branched-and-temporal databases with reasonable space and access time tradeoff. It is an index structure based on the BT-Tree [5]. The BT-Tree always splits at a current version whenever a data page or an index page is full. The BTR-Tree is able to split at a previous version while still keeping the posting property that only one parent page needs to be updated. The splitting policy of the BTR-Tree is designed to reduce data redundancy in the structure introduced by branching. Performance results show that the BTR-Tree has better space efficiency and similar query efficiency than the BT-Tree, with no overhead in search and posting algorithm complexity.

1 Introduction

There are applications which require the support of temporal data with *branched* time evolution, called *branched-and-temporal* data. In a branched-and-temporal database, both historic versions and current versions are allowed to be updated. We present an access method, the *BTR-Tree*, for branched-and-temporal databases with reasonable space and access time tradeoff.

A branched-and-temporal index method not only needs to support *version queries*, such as "show me all the data for branch B at time T", but also needs to support historical queries [6], such as "show me all the previous versions of this record." The BT-tree [5] is the only paginated access method explicitly proposed

T. Hadzilacos et al. (Eds.): SSTD 2003, LNCS 2750, pp. 28–45, 2003.

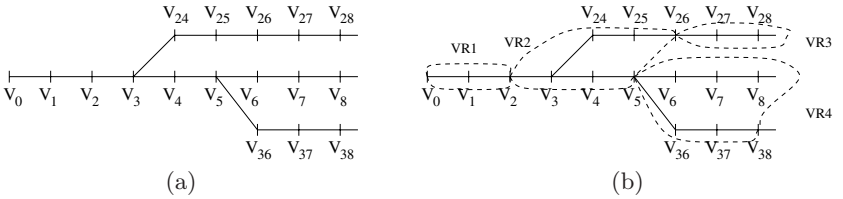

Fig. 1. (a)A version tree. (b) Version ranges of various kinds: $VR1$=$(v_0, \{v_2\})$, $VR2 = (v_2, \{v_{26}, v_5\})$, $VR3 = (v_{26}, \emptyset)$, and $VR4 = (v_5, \{v_{38}\})$

for branched-and-temporal data in the literature, although many access methods (for example, [3], [1], [8], [10] and [9]) have been proposed for temporal data (no branching), and Lanka and Mays proposed (based on ideas from [2]) the "fully persistent B+-tree" [7], a branch-only access method where no time dimension is considered.

The contribution of this paper is to introduce a new branched and temporal access method, the BTR-tree, based on the BT-tree. New splitting options which diminish version branching in pages and hence improve performance while maintaining the simple posting and searching algorithms of the BT-tree are proposed. In particular, in spite of not splitting at current versions, posting of the information about a split need only be made to one parent of the splitting page.

In Section 2 we review the BT-tree and describe the motivation of the BTR-Tree. Section 3 presents our new splitting algorithms used in the BTR-Tree. Performance results are presented in Section 4. We will conclude in Section 5.

2 BT-Tree Overview and Our Motivation

In this section, we first introduce some preliminary definitions. We will then review the BT-Tree and describe the motivation of the BTR-Tree.

2.1 Preliminaries

Time in a branched-and-temporal database is assumed to be discrete, described by a succession of nonnegative integers. Each branch is assigned a unique branch id, represented by a positive integer. A combination of a branch identifier B and a time stamp T is called a *version*. Sometimes we also denote a version (B, T) to be a version v. V is used to represent the version universe.

The whole version space V can be captured by a rooted *version tree*, whose nodes are the versions, with version v_i being the parent of version v_j if v_j is derived directly from v_i. Version v_0 is the root of the version tree. An example is given in Figure 1 (a).

Given two versions $(B1, T1)$ and $(B2, T2)$, if $(B2, T2)$ is derived from $(B1, T1)$, we denote that $(B1, T1) < (B2, T2)$. $(B2, T2)$ is also called a *descendant* of $(B1, T1)$. A version $(B1, T1)$ is defined to be a descendant of itself. Therefore

the set of versions v satisfying $(B1, T1) <= v$ is noted as $des((B1, T1))$. For example, in Figure 1 (a), $v_{25} \in des(v_2)$. If $(B2, T2)$ is a descendant of $(B1, T1)$, we say $(B1, T1)$ is an *ancestor* of $(B2, T2)$.

Branched-and-temporal data is represented by record variants. A *record variant* is characterized by four entries: a branch id, a time stamp, a time-invariant part called a *key*, and a data field. For example, $(3, 80, a, d)$ is a record variant with branch id = 3, time stamp = 80, key = a, and d representing the data value of this record variant. The notation of record variant can also be used to describe the discontinuation of data with a certain key in a specific branch. A *null record variant* $(b, t, k, null)$ indicates that at time t, the record variant with key k in branch b is deleted.

2.2 The BT-Tree Review

The BT-Tree is in fact a DAG. When restricted to one version, it is a tree. Similarly, other versioned structures such as the WOBT tree [3] and the TSB tree [8] are also DAGs, but they are traditionally called trees.

The BT-Tree has many properties in common with the ordinary B-tree. The nodes are disk pages. When page A contains the address of page B, A is a *parent* of B, B is called a *child* of A and there is a directed edge in the DAG from A to B. There is a distinguished page called the *root* which is an entry point for search. Leaf pages are at the same distance from the root page.

Also in common with B-trees, leaf pages contain data and are called *data pages*. Non-leaf pages direct search and contain only search information, not data. These pages are called *index pages*. The BT-Tree stores record variants in data pages. Exact match search (for example, "for a version (B, T) and key value k, what is the data value?") follows a unique path from the root to exactly one leaf, visiting only one page at each level. (Pages are at the same *level* if they are the same distance from the leaves.) The number of index pages is much smaller than the number of data pages. This is because each index page has numerous children.

As in the B-tree, when a new item is to be inserted, search directs the inserter to exactly one leaf. If that leaf page is full, a new leaf page must be allocated and some of the data from the old leaf page must be placed in the new leaf page. This is called a *split*. When a split is made, information is *posted* about the split to a parent node to direct search to the newly allocated node when appropriate.

Unlike the B-tree, which deals only with one-dimensional data, the BT-tree must cope with branching versions. To get an idea how BT-Tree copes with branching versions to achieve both space and query efficiency, we need a few more definitions about branching and versions.

- *Current Version*: A version is *current* if there is no descendent version in the same branch. For example, in Figure 1 (a) versions v_{28}, v_8, and v_{38} (each of them represents a branch and time pair) are all current versions.
- *Start Time and Share Time of a Branch*: If version $(B2, T2)$ is created from version $(B1, T1)$ where $B2 \neq B1$, then branch $B2$ is created from branch

$B1$ with *start time* $T2$ and *share time* $T1$. Version $(B2, T2)$ is also called the *start version* of a branch.

Version Ranges. A *version range* vr is a set of versions represented by the pair $(start(vr), end(vr))$, where *start(vr)* is a version and *end(vr)* is a set of versions satisfying the following condition: if $ev \in end(vr)$, then $start(vr) < ev$ and $\forall ev' \in end(vr)$ and $ev' \neq ev, \neg(ev < ev')$. This condition implies that end versions must be (strict) descendants of the start version and distinct end versions must lie on different branches. For example, in Figure 1 (b), $(v_0, \{v_2\})$ is a version range marked as $VR1$, while $VR = (v_0, \{v_1, v_2\})$ is not a version range because $v_2 \in end(VR)$ and $v_1 \in end(VR)$ and $v_1 < v_2$, in other words, these two end versions v_1 and v_2 lie on same branch.

Since version ranges are sets, we use set notation to describe the relationship among versions and version ranges. Thus, $v \in vr$ if $start(vr) <= v$ and $\forall ev \in end(vr), \neg(ev <= v)$. For example, in Figure 1 (b) $v_{37} \in VR4 = (v_5, \{v_{38}\})$ because $v_5 < v_{37}$ and $\neg(v_{38} <= v_{37})$, while $v_{38} \notin VR4$ because $v_{38} \in end(VR4)$. Similarly $vr1 \subseteq vr2$ if $v \in vr_1$ implies $v \in vr_2$. We freely use set notation and terminology where appropriate.

Note that inside a version range not all branches leading from the start version need be explicitly terminated. Thus, we can have open-ended version ranges. For example, $(v, \emptyset)$ is a version range consisting of version v and every $v' \in des(v)$. Thus $(v_0, \emptyset) = V$, the *version-universe*. Another example of an open-ended version range is shown in Figure 1 (b) where $(v_{26}, \emptyset)$ is the version range marked by $VR3$. The branch in $VR4$ containing v_6, v_7 and v_8 is an open-ended branch. Hence there is no end version for $VR4$ on that branch. Of course, at any given instant, V has some precise number of versions, and even an open-ended range is finite. But additional versions can be added on non-terminated paths, so the number of versions in a range can always increase. For example, in Figure 1 (b), if a new version v_{29} is derived directly from version v_{28}, the new version v_{29} is added to range $VR3$.

Indeed, all version ranges are open-ended in that, with branched versions, it is always possible to add versions to an existing range by producing a new branch from an existing version, even when all current paths to descendants are terminated. For example, for $VR2 = (v_2, \{v_{26}, v_5\})$ in Figure 1 (b), if we create a new version v_{44} from version v_{25}, then the new version v_{44} is added to the version range $VR2$.

There are two important reasons that we define the concept of version ranges. First of all, branched-and-temporal data can be clustered according to version ranges to efficiently support typical version queries. Secondly, to save space, the BT-Tree exploits maximum data sharing among versions in version ranges. For example, in Figure 1 (b), if versions in $VR2$ share the same data value for key k, only one record variant (the one with the earliest time stamp) is needed to represent all the data corresponding to every version in $VR2$ and key k. When record variant (v_2, k, d) (v_2 represents a branch and time pair) is used to

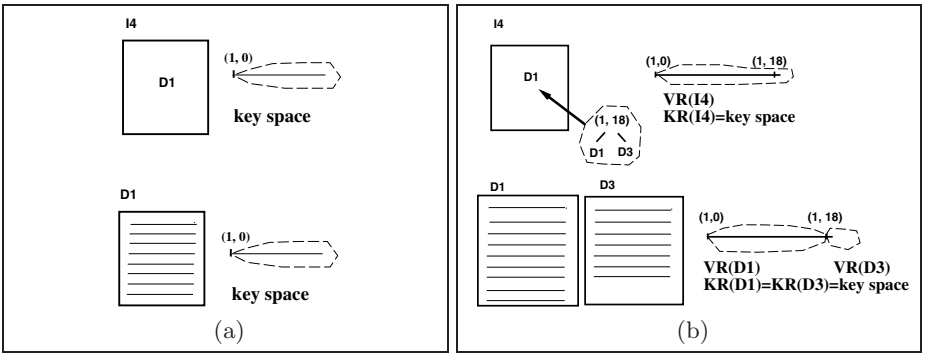

Fig. 2. (a) This is how the BT-Tree gets started. (b) Version-split page $D1$ at $(1, 18)$. We use $VR(I)$ to represent the version range corresponding to page I and $KR(D)$ to represent the key range corresponding to page D

represent all the data corresponding to versions in $VR2$ and key k, we say the record variant (v_2, k, d) is *valid* at versions in $VR2$.

BT-Tree Splits. Each page in the BT-Tree corresponds to a key range and version range and only contains information related to the key range and version range. As shown in Figure 2 (a), when the BT-Tree gets started, there is only one index page and one data page with both pages corresponding to the key space and the entire version space. New record variants are directed to the sole data page $D1$ in the BT-Tree. (Data is represented by horizontal line segments in the figure.)

When an insertion of a new record variant (B, T, k, d) finds insufficient space in a data page, we will split the data page at version (B, T) ((B, T) is always *current* since (B, T, k, d) is a new record variant.) This is called a *version-split*. The BT-Tree only performs version splits at current versions, not previous versions. For example, in Figure 2 (b) at time 18, in order to insert a new record variant we have to version-split data page $D1$ at current version $(1, 18)$. A new page $D3$ corresponding to a new version range $((1, 18), \emptyset)$ and the same key range as the old page $D1$ is created. The old page $D1$ corresponds to the version range $((1, 0), \{(1, 18)\})$. The version tree to the right of the figure shows the corresponding version range and key range of each page.

Any record variants of $D1$ which are still valid at version $(1, 18)$ are *copied* to $D3$. *In general, when an overflowing data page is split at a current version, only the record variant that caused the overflow is* moved *to the new page, although many existing record variants in the original page are* copied *to the new page. Therefore the original overflowing page is still full.*

Posting is implemented by replacing the original reference to the old page with a *split history* tree (A similar mechanism has been exploited in previous index structures, such as [4].) The split history tree or *sh-tree* is used within each index page. The sh-tree is a small binary tree containing three types of

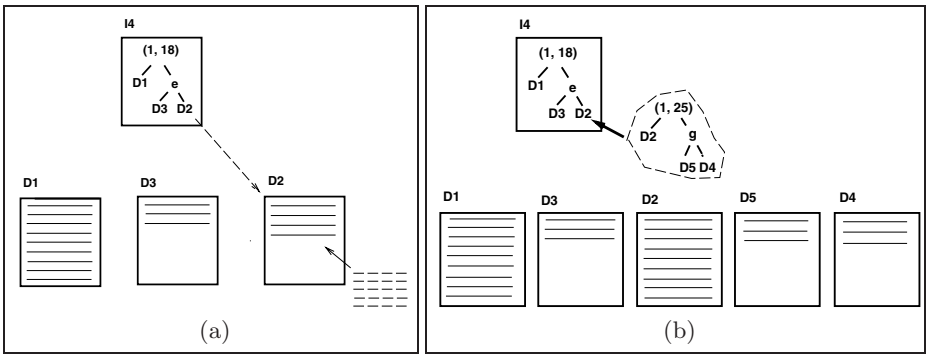

Fig. 3. (a) Version-and-key split data page $D1$ at $(1, 18)$ and e. (b) Version-and-key split data page $D2$. When we try to post to the root page $I4$, we found inefficient space

nodes: *vsh* nodes, *ksh* nodes and *leaf* nodes. A vsh node contains a branch id and a time stamp (indicating a *version*), a ksh node contains a key value while a leaf node contains a disk page address of a child page in the next lower level of the BT-Tree. In Figure 2 (b), the sh-tree that is going to be posted to index page $I4$ (to replace the pointer to page $D1$) contains only one vsh node and two leaf nodes.

We will be using the following information about BT-Trees in designing the new splitting algorithm. *A vsh node* (B, T) *in an index page divides the lower level BT-Tree rooted at* (B, T) *into two parts. The right subtree of the vsh node refers to lower level pages whose version range contains descendants of* (B, T)*, while the left subtree of the vsh node refers to pages whose version range contains versions which are not descendants of* (B, T). For example, after we post the sh-tree to page $I4$ in Figure 2 (b), page $D3$ contains all information whose versions are descendant of version $(1, 18)$, and page $D1$ contains all information whose versions are not descendant of version $(1, 18)$.

When version-splitting a data page results in a new data page with the number of record variants exceeding some threshold, a *version-and-key-split* occurs. Specifically, in the BT-Tree, a version-and-key-split is a current version-split followed by a key split. No pure key split (a key split without an immediate preceding current version split) is allowed in BT-Tree. In our running example, since the resulting new page $D3$ after the version-split has too many record variants in it, as shown in Figure 2 (b), we do a version-and-key-split instead, creating two new data pages $D3$ and $D2$, as shown in Figure 3 (a). The sh-tree posted after the version-and-key-split contains an additional ksh node e. A ksh node e divides the lower level BT-Tree rooted at e into two parts. The right subtree of the ksh node refers to lower level pages whose key range contains key values greater than or equal to e while the left subtree of the ksh node contains key values less than e.

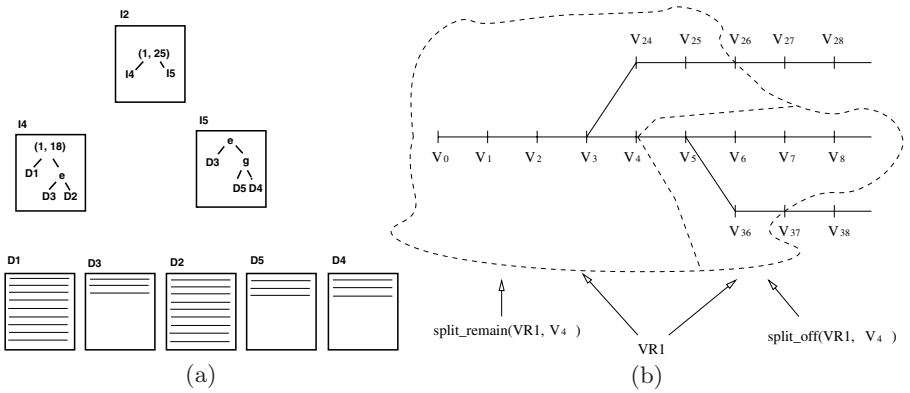

Fig. 4. (a) Version-split the current root page $I4$, creating a new index page $I5$ and a new root $I2$. (b) An example of version range splitting

Now that we've introduced vsh nodes and ksh nodes, the simple *searching algorithm* of the BT-Tree can be easily understood. *Suppose we are searching for a record with version v and key k. When we meet a vsh node (B, T), if v is a descendant of version (B, T), we go right, otherwise we go left. When we meet a ksh node $k1$, if k is greater than or equal to $k1$, we go right, otherwise we go left.*

Notice that in both Figure 2 (b) (version-split page $D1$) and Figure 3 (a) (version-and-key-split page $D1$), the original overflowing page $D1$ remains full after the split. In Figure 3 (a) we also show a lot of new record variants being inserted into data page $D2$ after the version-and-key-split. In Figure 3 (b) we can see that at time 25 we have to version-and-key split page $D2$ creating new data pages $D5$ and $D4$. But when we try to post to the root page $I4$, we find insufficient space. Therefore we version-split the current root page $I4$, creating a new index page $I5$ and a new root $I2$. This is shown in Figure 4 (a). Generally speaking, if a page in the lower level is split at version v and posting to an upper level index page P causes P to overflow, we will split P at version v too. Therefore *both index pages and data pages split at current versions only.* When version-splitting an index page, we will extract an sh-tree which only has ksh nodes. When the size of the resulting sh-tree exceeds a threshold, we do a version-and-key-split as well.

Although we do not intend to repeat the index page split algorithm from [5]) here, we do want to emphasize that version-splitting an index page sometimes creates multiple parents for lower level pages. For example, in Figure 4 (a), after version-splitting index page $I4$, generating page $I5$, data page $D3$ has two parents $I4$ and $I5$. In case of version-and-key splitting an index page (version split the index page first, followed by a key split on the new page created), only the version split, not the key split, might create multiple parents for lower level pages. For example, if $I4$ was version-and-key-split creating two new pages $I5$ and $I5'$, it is possible that both pages $I4$ and $I5$ are the multiple parents of

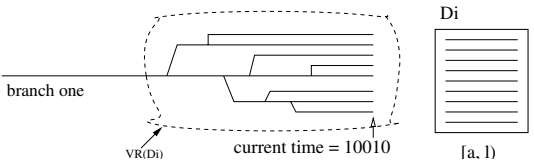

Fig. 5. An extreme case where the version range of the page D_i contains 7 start versions from 7 different branches

a data page. It is also possible that pages $I4$ and $I5'$ are the multiple parents of an data page, but it is not possible that pages $I5$ and $I5'$ are the multiple parents of an data page. This is because the version-split algorithm (see [5]) copies some leaf nodes (a leaf node contains the disk address of a child page) from the old page to the newly created page, hence creating multiple parents for the child page. The key-split followed by the version-split only divides the leaf nodes into two non-intersecting sets with one set of leaf nodes in one page (stored as a tree separated by some keys) and the other set in the other page. The property that *only version-splits not the key splits made after version splits could generate multiple parents* is important in creating our new splitting method for the BTR-Tree.

2.3 Motivation for the BTR-Tree

The performance of the BT-Tree is sensitive to the way branches are created. This is mainly because the BT-Tree always splits at a current version, no matter how branches are created.

Branch creation affects the performance of the BT-Tree through those pages whose version range contains the start version of a branch. An extreme case is shown in Figure 5 where the version range of the page D_i (also denoted as $VR(D_i)$) contains branch one and 7 other different branches whose start versions are in the version range of the page D_i. Assume the key range of page D_i is $[a, l)$. For each of these eight different branches having a new insertion or update involving a record variant with a key within the key range $[a, l)$, the page D_i is going to be accessed. Now suppose that the page D_i is full.

When inserting or updating a full page, we need to split the page no matter which branch the new record variant comes from. After splitting at current version (B, T), later record variants coming from the same branch B will be directed to the newly created page, not the old overflowing page. However, record variants coming from other branches will still be directed to the original overflowing page. But the overflowing page D_i remains full after splitting D_i at a current version. Therefore, any new insertion or update to page D_i will immediately cause another split of page D_i.

Assuming no new branches are created in the version range, the maximum number of times D_i may be split is eight (7 different branches with the start version in $VR(D_i)$ plus branch one whose start version is not in $VR(D_i)$.) Since

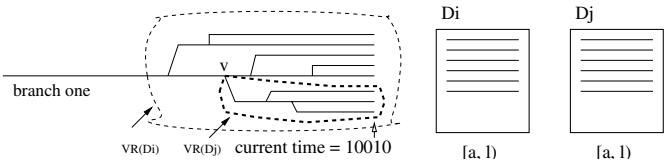

Fig. 6. An example of R-splitting for page D_i

many of the shared record variants may be copied to the new page when splitting a data page, shared record variants could have maximally eight copies in eight different pages, leading to expensive space cost.

Our method of decreasing the number of copies of shared records in cases such as we have shown in Figure 5 (hence ultimately increasing the space efficiency of the indexing structure) is to split the page at a previous version v (not a current version v_c) to decrease the number of branches contained in a page. To describe it more clearly, we define the split operation on the version range.

Definition 2.1 *Let v be a version and vr be a version range with $v \in vr$. Split(vr, v) generates two version ranges:*
split-off(vr, v): The part of the version range that is "split-off" is defined by

- *start(split-off $(vr, v)) = v$,*
- *end(split-off $(vr, v)) = \{v' | v' \in end(vr)$ and $v' \in dec(v)\}$.*

Split-off is the subset of vr whose members are descendants of v.
split-remain(vr, v): The part of the version range that "remains" is defined by

- *start(split-remain $(vr, v)) = start(vr)$,*
- *end(split-remain $(vr, v)) = (end(vr) \cup \{start($ split-off $(vr, v))\} - end($ split-off $(vr, v))$.*

Split-remain is the subset of vr whose members are not descendants of v.

For example, let's consider the version range $VR1$ in Figure 4 (b). $VR1 = (v_0, \{v_{26}, v_{37}\})$. *Split* $(VR1, v_4)$ generates two version ranges: *split-remain*$(VR1, v_4) = (v_0, \{v_{26}, v_4\})$ and *split-off*$(VR1, v_4) = (v_4, \{v_{37}\})$.

With the definition of *split_off* and *split_remain*, we can illustrate the effect of splitting a page at previous version more clearly. Let's use the page D_i in Figure 5 as an example. After splitting at a previous version v, the version range of the new page created will be *split_off*$(VR(D_i), v)$, while the version range of the old page after the split will be *split_remain*$(VR(D_i), v)$. The number of branches in the original page D_i will be decreased if the split-off contains some of the branch start versions.

Splitting a page at a previous version is different from splitting a page at a current version in that some of the records in the original overflowing page will be *moved* (not just *copied*) to the version range of the newly created page. This

is mainly because splitting at a previous version really divides the version range of the original overflowing page in the middle and moves many existing versions (non-current) to the newly created page. Hence the version range of the new page contains not just the current version v_c and those versions derived from v_c in the future, as in the case of splitting at current version v_c. Splitting a page at a non-current version v is called *R-splitting*. We will refine the definition of R-splitting in the next section.

Figure 6 shows R-splitting data page D_i from Figure 5 at version v. The new page will corresponds to the version range enclosed in the bold dotted boundary, which is also the split-off. The starting versions of some branches are included in the split-off. By doing so, all records created in those branches are removed from the original page and some records that are created in ancestor branches of those branches but are inherited by those branches are copied, leaving empty space in both page Di and the newly created page D_j, allowing further insertions and updates to both pages without further page splitting. By including start versions of some branches in the split-off, the number of branches in the version range of the original page D_i is decreased, hence the situation we described in Figure 5 where a full page contains information from many different branches and the page has to be split many times is mitigated.

3 Splitting Using Path-Defined Version Ranges

We call a BT-Tree which is extended to allow R-splitting a *BTR-Tree*. Since a page in a BT-Tree might have multiple parents, R-splitting needs to guarantee that one only needs to post to the parent page that leads the search to the splitting page. This is called *local* splitting.

In order to guarantee a local R-splitting, we explore paths and version ranges associated with paths in the following subsections. We will define a *path* and a *path to a page*. We will give a refined definition of R-splitting. The data page and index page splitting strategy follows. To illustrate these concepts, we use the BT-tree in Figure 7 and the corresponding version tree in Figure 8.

3.1 Definition of Path

Definition 3.1 Path: *A path r is a sequence of the vsh nodes and ksh nodes that channel the search from the root of the BTR-Tree to a data page along with the directions (left or right) to move after encountering these vsh nodes and ksh nodes.*

Given the version tree shown in Figure 8 and a corresponding example shown in Figure 7, the *path* $r6$ is "$(3, 80)L(1, 25)R(1, 50)ReRgR$". This means we met vsh node $(3, 80)$ first, then turn left, then we met vsh node $(1, 25)$ and turn right, after that we met vsh node $(1, 50)$ and turn right, then we met ksh node e and turn right, finally we met ksh node g and turn right. Several other paths are also indicated in Figure 7. These paths all lead to data pages. But paths may also lead to index pages. We make the following definition.

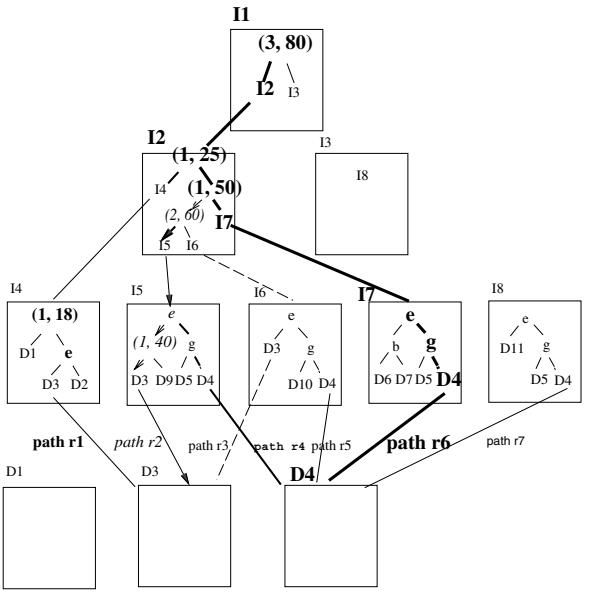

Fig. 7. Examples of A BT-Tree and paths

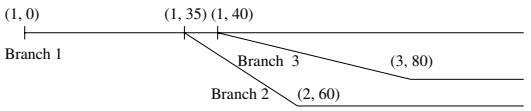

Fig. 8. The version tree of the running example

Definition 3.2 Path to a Page: *If path r goes through an index page or data page P, $path(r, P)$ is defined to be the sequence of the vsh nodes and ksh nodes that channel the search from the root of the BT-Tree to the page P along with the directions (left or right) to move.*

For the same example shown in Figure 7, the $path(r6, I7)$ is "$(3, 80)L$ $(1, 25)R$ $(1, 50)R$".

3.2 The Version Range Associated with a Path

Paths are associated with version ranges. In this section we define the version range of a path and design an algorithm to calculate it.

Definition 3.3 Path Defined Version Range: *For each path r and page P (P could be an index page or a data page) that path r went through, $path(r, P)$ identifies a version range vr, denoted as $VR(path(r, P))$. $VR(path(r, P))$ is defined as follows: a version (B, T) is in $VR(path(r, P))$ if and only if there ex-*

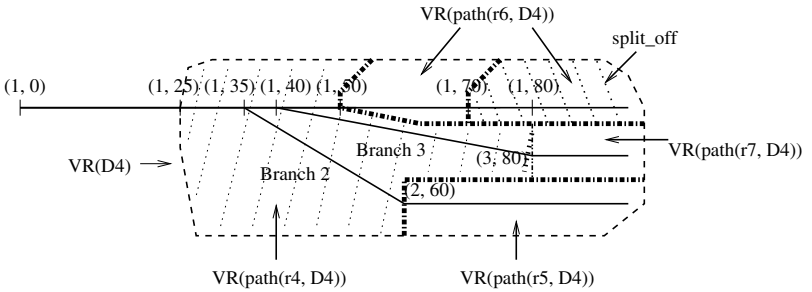

Fig. 9. Version ranges of paths from root page $I1$ to data page $D4$ (path $r4$, $r5$, $r6$ and $r7$ in Figure 7) are disjoint. $VR(path(r4, D4)) = ((1, 25), \{(1, 50), (3, 80), (2, 60)\})$. $VR(path(r5, D4)) = ((2, 60), \emptyset)$. $VR(path(r6, D4)) = ((1, 50), \emptyset)$. $VR(path(r7, D4)) = ((3, 80), \emptyset)$

ists a key k such that the search for (B, T, k) through the BTR-Tree follows $path(r, P)$.

Given a $path(r, P)$, an algorithm can be designed to calculate $VR(path(r, P))$. The intuition of the algorithm is based on the fact that a path is actually a search path and $VR(path(r, P))$ is in fact the search space defined by $path(r, P)$. From the search algorithm, we know that the searching space for version v starts with the whole version space as we start from the root index page. A $(B, T)R$ in the path means that version v is a descendant of (B, T), therefore the search space changes with (B, T) as its start version. A $(B, T)L$ in the path means that version v is not a descendant of (B, T), therefore the search space further shrinks with (B, T) as one of its end versions. The detailed algorithm returning the *Start_version* and *End_set* which define $VR(path(r, P))$ is as follows.

1. Initialization
   Start_version = (B_{start}, T_{start}) = (1, 0) (B_{start}=1, T_{start}=0)
   End_set = $\emptyset$.
2. Read the vsh nodes in $path(r, P)$ in order. If the current vsh node is (B', T'), do the following:
   - If (B', T')R:
     - If $T' > T_{start}$, then set Start_version = (B', T'), and remove from the End_set all versions that are not descendants of Start_version.
     - Otherwise do nothing.
   - If (B', T')L:
     - If (B', T') is not a descendant of the current Start_version (B_{start}, T_{start}), ignore (B', T'), else
     - If $\exists$ $(B_i, T_i) \in$ End_set such that (B_i, T_i) is an ancestor of (B', T'), ignore (B', T'), else
     - If $\exists$ $(B_i, T_i) \in$ End_set such that (B', T') is an ancestor of (B_i, T_i), then remove from the End_set all versions that are descendants of (B', T') and add (B', T') to End_set.

- Otherwise, add (B', T') to End_set.

In the example shown in Figure 7, $path(r6, I7)$ = "(3, 80) L (1, 25) R (1, 50) R". According to the previous algorithm, $VR(path(r6, I7))$ is (Start_version=(1, 50), End_set = $\emptyset$).

We make two observations here. First, in the process of computing $VR(path(r, P))$, the version range $VR(path(r, P))$ is initialized as the whole version space and it gets smaller and smaller as we go through the algorithm. Hence if page I is a parent of page D and path r goes through both page I and page D, then $VR(path(r, I)) \supseteq VR(path(r, D))$.

Secondly, since $(B, T)L$ in the path leads to non-descendants of (B, T) and $(B, T)R$ in the path leads to descendants of (B, T), the intersection of $VR(path\ r)$ and $VR(path\ s)$ is empty if path r contains $(B, T)L$ and path s contains $(B, T)R$.

3.3 Properties of Paths

In order to develop our new splitting strategy, we are concerned only when paths to a page P visit two different parents of P. If P has only one parent, splitting can occur anywhere without causing posting to more than one parent page. Thus, we will only be interested in paths which diverge at a vsh node. (Paths which diverge at a ksh node will end in different data pages since key splits of index pages split the children into disjoint key ranges.)

Definition 3.4 *path r* **v-diverges** *path s*

Path r v-diverges path s if $\exists$ *(B, T) such that path r contains (B, T)L and path s contains (B, T)R.*

For example, path $r5$ in Figure 7 v-diverges with path $r6$ because we have $(1, 50)L$ in path $r5$ and $(1, 50)R$ in path $r6$. Our goal in developing a method based on paths will be to assure that, even when a page P has multiple parents, only one of the parents, corresponding to one of the paths to P, will be involved when information about a split of P must be posted.

Consider how a page (such as page $D3$ in Figure 4 (a)) starts to get multiple parent pages. Originally page $D3$ has only one parent (page $I4$ in Figure 3 (a)). Version-splitting page $I4$ at version $(1, 25)$ generates the second parent page (page $I5$) of $D3$. When the version-split creates multiple parents, the two paths to the same page (in our case, the two paths are (see Figure 4 (a)) : one path from root page $I2$ to page $D3$ going through page $I4$, the other path from root page $I2$ to page $D3$ going through page $I5$) are separated by the vsh node that is posted after the version-split (in our case, the vsh node is $(1, 25)$ in the root page $I2$). Since different paths to a same page v-diverge, (again, if the paths diverge at a ksh node, they do not end at the same page) using the induction method, we can show that VR($path(r, P)$) $\cap$ VR($path(s, P)$) = $\emptyset$ if r and s are different paths to the same page P. An example is shown in Figure 9 where the version range of the four different paths to the same page $D4$ from Figure 7 are disjoint.

If we split page $D4$ in such a way that the split-off (the version range that the new page is responsible for) is contained completely in $VR(path(r6, D4))$, as shown in Figure 9, it is not necessary to post to the other three index pages that the other paths to the same page $D4$ (path $r4$, path $r5$ and path $r7$) are lying in. The intuition is that posting will split one path into two paths and also split the original version range of the path into two disjoint version ranges. Neither of these two disjoint version ranges intersects any version range corresponding to other paths. Now let's refine the definition of R-splitting as follows.

Definition 3.5 *R-splitting: Consider an index page or a data page P and a $path(r, P)$. Let vr satisfy $vr \subseteq VR(path(r, P))$ and the end set of vr is empty. We call splitting page P in such a way that the split off is vr R-splitting. After the split, page P corresponds to version range $(VR(BEFORE(P)) - vr)$ where $BEFORE(P)$ represents the page P before splitting. The new page created out of the split corresponds to the version range vr.*

We will explain later in Section 3.4 why the end set is required to be empty in the definition of R-splitting. Many version ranges vr may satisfy the definition for R-splitting. The next section considers which split-off version range to choose for an R-split.

3.4 The Splitting Strategy

Our motivation in choosing a split-off among several possiblities is to decrease the branching in index and data pages.

We first define the *owning branch of a page*. When the BTR-Tree is initialized, we have one index page and one data page. They contain information about the first branch in the branched-and-temporal database, branch one. The owning branch of the index page and the data page is branch one. New data pages and index pages are created from page splitting. If a new data page or a new index page is created by splitting an existing page at version (B, T) ((B, T) is the start version of the split-off when the page is R-splitting), the owning branch of the data page or the index page is B.

The Data Page Splitting Strategy. When we try to insert a new record variant (B, T, k, d), the search algorithm directs us following *path* r to a data page D. We need to split the data page D if there is not enough space in the data page D for the new record variant. Since (B, T, k, d) is a new record variant, we know that (B, T) is current. We denote the start version of a branch B as (B, *Start-time* (B)). The following algorithm is followed when splitting the data page D.

- If B is the owning branch of the data page D, we version-split at the current version (B, T).
- Otherwise, we look at $VR(path(r, D))$. We consider the two cases where the start version of B is in and not in $VR(path(r, D))$.

- If (B, *Start-time* (B)) $\in VR(path(r, D))$, we compute *VR_splitting* = *split_off* (VR (*path* (r, D)), (B, *Start-time*(B))) and look at the End_set S of *VR_splitting*:
  - $S = \emptyset$: R-split with the split-off to be *VR_splitting*.
  - $S \neq \emptyset$: split at the current version (B, T).
- Otherwise, we look at the End_set S of VR(*path* (r, D)):
  - $S = \emptyset$: R-split with the split-off to be $VR(path(r, D))$.
  - $S \neq \emptyset$: split at the current version (B, T).

The reason that we check the End_set of *VR_splitting* before deciding whether to R-split or not is to guarantee that the version space of the data page D after splitting is still a version range (not a union of disjoint version ranges). This enables us to retain the simple searching scheme of the BT-Tree.

The Index Page Splitting Strategy. Assume that the path r is followed to get to a data page D while trying to insert a new record variant. Subsequently, the data page D is split. The parent page of the data page D lying in the path r needs to be updated to reflect the split. As in the BT-Tree, this is achived by posting. If the page D is split at the current version (B, T), we post (B, T). If the page D is R-split, we post (B_{start}, T_{start}), the start version of the split-off. When the posting make the index page (say, I) full, we split the index page I by looking at how the data page D was split:

- If the data page D is split at the current version (B, T), we split the index page I at the current version (B, T), and post (B, T).
- If the data page D is R-split with the split-off to be *VR_splitting*, we compute a new split-off *VR_splitting_I* = *split_off* ($VR(path(r, I))$, (B_{start}, T_{start})), with (B_{start}, T_{start}) the start version of *VR_splitting*, and R-Split the index page I with the new split-off *VR_splitting_I*. We then post (B_{start}, T_{start}).

For index page R-splitting, we no longer need to check whether the End_set of the split-off *VR_splitting_I* is empty, like we did for the data page splitting case. This is because we can deduce the fact that the End_set of *VR_splitting_I* must be empty from the property of $VR(path(r, I)) \supseteq VR(path(r, D))$ and the fact that the End_set of the split-off for the data page D *VR_splitting* is empty.

4 Performance Results

We present some results of our performance comparison between the BTR-Tree and the BT-tree.

We assume all record variants, including *null* record variants, have the same size. A transaction is either an insertion of a record variant with a new key, an update of an existing record variant or a delete of an old record variant.

The database system starts up with only one branch. Other branches are created gradually after a number of transactions occurred in the first branch.

Table 1. The total number of data pages

Update Rate (%)	BT-Tree (A)	BTR-Tree (B)	Improvement ((A-B)/B)
0	7679	6438	19.3%
5	7329	6190	18.4%
10	7025	6052	16.1%
30	6010	5454	10.2%
50	5228	4902	6.7%
70	4161	4007	3.8%

Table 2. The total number of index pages

Update Rate (%)	BT-Tree (A)	BTR-Tree (B)	Improvement ((A-B)/B)
0	474	420	12.9%
5	480	439	9.3%
10	469	416	12.7%
30	378	366	3.3%
50	332	293	6.7%
70	280	273	2.6%

Transactions are randomly assigned to existing branches. We use the intuitive measurement of data page and index page numbers to measure the space cost and the query efficiency.

The number of branches is fixed to be 100. All branches other than the first are randomly created between the 3,000th and 30,000th transaction with a randomly selected ancestor version (other branch creation profiles were also implemented, but since their results were similar, they are not presented here.) The key range of the first branch is [0, 800,000). All other branches are allowed to modify versioned records in key range [0, 600,000). We vary the fraction of updates versus insertions. No deletes are allowed. In these experiments, the height of the BTR-tree never rose above 3. (Consolidation algorithms (see [5]) have been designed to guarantee good version query performance in case of deletes. Experiments show that the effect of consolidation on the BTR-Tree is as same as the BT-Tree.)

We first comment on the space utilization. Table 1 (Table 2) shows that comparison of total number of data pages (index pages) in each index structure. With the new split method, the number of data pages and index pages needed is smaller This is anticipated. The number of times that record variants in a page are going to be copied is less, making the total space cost less.

When the update rate increases, the differences on the space cost get smaller. The reason is as follows. The number of distinctive key values in a page is smaller when the update rate is higher. When a full page is split at a current version, the number of record variants that gets copied to the new page is smaller, making the saving on the number of times that record variants in the page are copied

Table 3. The number of data pages that need to be accessed for the current version query in branch one

Update Rate	BT-Tree	BTR-Tree
0	222	216
5	213	210
10	202	200
30	153	152
50	111	112
70	64	65

in case of R-splitting over the number of times that record variants in the page are copied in case of splitting at a current version less.

Now we comment on the version query efficiency. With 100 branches, the version query efficiency may vary from one branch to another. We compare the version query efficiency of the BT-Tree and the BTR-Tree for branch one. Branch one is considered as a respresentative because it has the maximum number of record variants among the 100 branches. As time proceeds, all other branches may reach a state having as many record variants as branch one now. (The analysis for other branches are similar). We consider the current version query efficiency measured by the number of data pages containing record variants valid at current version. Table 3 shows the number of data pages that needs to be accessed for both the BT-Tree and the BTR-Tree for the current version query in branch one. With low update rate, the query efficiency of the BTR-Tree is slightly better than the BT-Tree. With high update rate, it is almost the same.

5 Conclusion

Based on the BT-tree [5], we have proposed a new splitting algorithm, namely R-splitting, thus creating a new branched-and-temporal index structure, the BTR-tree. Information gathered from the search path enables expanded splitting choices without requiring posting to more than one parent. We have proposed a detailed splitting strategy for the BTR-tree. This strategy aims to decrease the amount of branching in pages while maintaining simple posting and searching algorithms. The performance results show that the BTR-Tree improves the space performance significantly when the space requirement is the most (relatively low update rate). The new splitting strategy improves query performance slightly for the same low update case.

References

[1] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. On optimal multiversion access structures. In *Porc. Symp. on Large Spatial Databases, in Lecture Notes in Computer Science, Vol. 692*, pages 123–141, Singapore, 1993. 29

[2] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences, 38*, pages 86–124, 1989. 29

[3] M. C. Easton. Key-sequence data sets on indelible storage. *IBM J. Res. Development*, 30(3):230–241, 1986. 29, 30

[4] G. Evangelidis, David Lomet, and B. Salzberg. The hB-$^{\Pi}$-tree: A Multiattribute Index Supporting Concurrency, Recovery and Node Consolidation. *VLDB Journal*, 6(1), January 1997. 32

[5] Linan Jiang, Betty Salzberg, David Lomet, and Manuel Barrena. The BT-tree: A branched and temporal access method. In *International Conference on Very Large Data Bases*, pages 451–460, Cairo, Egypt, September 2000. 28, 34, 35, 43, 44

[6] Gad M. Landau, Jeanette P. Schmidt, and Vassilis J. Tsotras. Historical queries along multiple lines of time evolution. *VLDB Journal, 4*, pages 703–726, 1995. 28

[7] Sitaram Lanka and Eric Mays. Fully persistent B+-trees. In *Proceedings of the ACM SIGMOD conference on Management of Data*, Denver, CO, 1991. 29

[8] David Lomet and Betty Salzberg. The performance of a multiversion access method. In *Proceedings of the ACM SIGMOD conference on Management of Data*, pages 354–363, 1990. 29, 30

[9] Peter Muth, Patrick O'neil, Achim Pick, and Gerhard Weikum. Design, implementation, and performance on the LHAM log-structured history data access method. In *Proceedings of the 24th VLDB Conference*, pages 452–463, New York, 1998. 29

[10] Vassilis J. Tsotras and Nickolas Kangelaris. The snapshot index: An I/O-optimal access method for timeslice queries. *Information Systems 20(3)*, pages 237–260, 1995. 29