The BT-Tree: A Branched and Temporal Access Method

Linan Jiang, Betty Salzberg *
College of Comp. Sc., Northeastern Univ.
Boston, MA 02115
{linan, salzberg}@ccs.neu.edu

David Lomet
Microsoft Research
One Microsoft Way Bldg 9
Redmond, WA 98052
lomet@microsoft.com

Manuel Barrena †
Universidad de Extremadura
Cáceres, Spain
barrena@unex.es

Abstract

Temporal databases assume a single line of time evolution. In other words, they support time-evolving data. However there are applications which require the support of temporal data with *branched* time evolution. With new branches created as time proceeds, branched and temporal data tends to increase in size rapidly, making the need for efficient indexing crucial. We propose a new (*paginated*) access method for branched and temporal data: the BT-tree. The BT-tree is both storage efficient and access efficient. We have implemented the BT-tree and performance results confirm these properties.

1 Introduction

There are many database applications that require the support of time-evolving data. Temporal database systems model explicitly the temporal behavior of data, thus providing the ability to store and query temporal data efficiently [9].

Conventional temporal databases assume a single line of time evolution. As an example, consider an architect's design of a new house (say Joe's house). The house design starts from scratch and evolves over time. Figure 1 shows the design of Joe's house along a single line of time evolution starting from January. A temporal database captures the evolution of this design. Queries such as "Find Joe's house design in February" are supported by the database.

While conventional temporal databases work well for many temporal database applications, they are not sufficient for applications that require the support of temporal data with *branched* time evolution, called **branched-and-temporal data**. Branched and temporal data arises in several important areas, such as software configuration control and engineering design. An example application is given in [4]. In our running example, consider the case where, at some time in May, the architect begins a new house design (say Paul's house). Instead of starting from scratch, the architect may choose to start from Joe's house design in February, which is already stored in the database. Joe's house design in February is modified to suit Paul's requirement later on. As time proceeds, Paul's house design can be viewed as a new time evolution branch which starts in May with Joe's house design in February as its initial design. Figure 2 captures the process of temporal house design with branched time evolution. Graphs as shown in Figure 1 and Figure 2 describing the evolution of the history of different branches are called **version trees**.

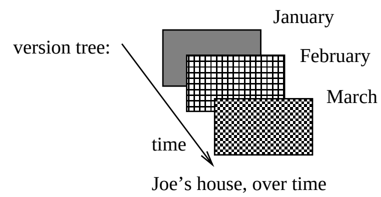

Figure 1: House design with a single line of time evolution.

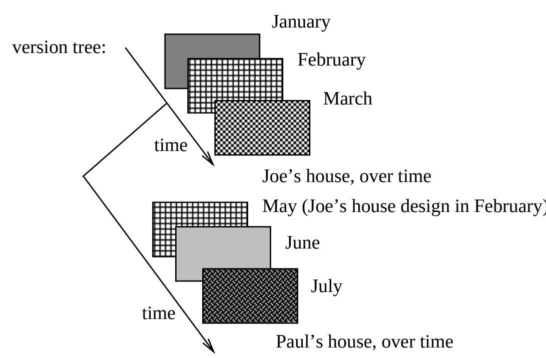

Figure 2: House design with branched time evolution.

A branched and temporal database, such as the house design database, has three dimensions: data space, branch, and time. In our example, the *data space* contains different parts of a house such as the kitchen and the bedroom. The *branch* corresponds to the full design product, say "Joe's house". *Time* reflects the changes made in the design as it

*This work was partially supported by NSF grant IRI-93-03403 and IRI-96-10001 and by a grant for hardware, software and research from Microsoft Corp.

†This work was partially supported by DGES grant PR95-426.

0 **Proceedings of the 26th VLDB Conference, Cairo, Egypt, 2000.**

evolves. This is illustrated in Figure 3. A (Branch, Time) pair, say Joe's house in February, noted as *(Joe's house, Feb.)*, is called a **version**.

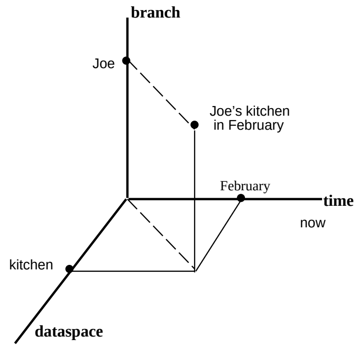

Figure 3: Design database dimensions.

Given a specific branch B, branches that are derived from branch B are called **descendent branches** of B. For example, "Paul's house" is a descendent branch of "Joe's house". Analogously, for a specific version (B, T), versions that are derived from (B, T) are called **descendent versions** of (B, T). For example, *(Paul's house, June)* is descendent version of *(Paul's house, May)*. If a branch $B1$ (version $(B1, T1)$) is a descendent branch (descendent version) of branch $B2$ (version $(B2, T2)$), we say that branch $B2$ (version $(B2, T2)$) is an ancestor branch (ancestor version) of branch $B1$ (version $(B1, T1)$).

With new branches created as time proceeds, branched and temporal data tends to increase in size rapidly, making the need for efficient indexing crucial. A branched-and-temporal index method not only needs to support version slice queries, such as "show me the design for Joe's house in March.", but also needs to support historical queries [4], including horizontal queries and vertical queries, which arise because of branching. A typical horizontal query is "Find all the house designs for a given branch, say "Joe's house", or one of its *descendent* branches, in June". This shows what has evolved from Joe's house. A typical vertical query is "Find all the house designs for a given branch, say "Paul's house", or one of its *ancestor* branches, in July." This shows how Paul's house has evolved differently from its ancestors.

Simply concatenating branch and key, and using temporal access methods for branched-and-temporal data does not consider the ancestor descendent relationship among versions, hence won't be able to support historical queries efficiently. Even for version slice queries the data would not be clustered efficiently with concatenation since ancestor branches contribute to the version slices of their descendent branches. For example, some of Joe's house design is shared by Paul's house in May.

Our perspective on how to index a branched and temporal house design database is motivated by (1) keeping the total amount of disk space small and (2) making the number of disk accesses for typical queries, such as version slice query and historical query, minimal. Therefore in designing the BT-tree, we focus on exploiting the sharing property of data records across different versions to save space, meanwhile clustering data records according to versions (for version slice queries) and versions with ancestor descendent relationship (for historical queries) to achieve query efficiency. Our solution to the problem provides a reasonable trade-off between space and access time.

To save space, we exploit how data is shared between versions. Data records consist of a invariant part (usually called the **key**), which describes the part of the data space they cover, and a varying part which contains the branch identifier, a time stamp, and the rest of the data (in a house design, this might include the type and size of cabinets or the color of the paint.) When Joe's house design in February did not change the kitchen design from its January's version, we say that the data record with key "kitchen" and version *(Joe's house, Jan.)* is shared between two versions *(Joe's house, Jan.)* and *(Joe's house, Feb.)*. Later on if Paul's house design in May created from Joe's house design in February also did not change the kitchen design, the same data record will be shared by version *(Paul's house, May)* as well.

Disk pages which contain data records are called **data pages**. Data records in a data page are shared among different versions as much as possible. Data page splitting policies will cause some copying of data records, requiring some extra space for duplication, but resulting in more efficient search. Other pages, which direct searches, are called **index pages**.

Assuming that our house only consists of a kitchen and a bedroom, an example of the structure capturing the branched evolution in Figure 2 is shown in Figure 4. The index page in Figure 4 indicates that if you are searching for any data derived from version (Paul's house, June) you look in data page 2, otherwise you look in data page 1.

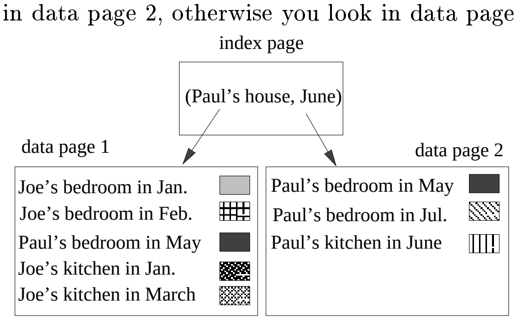

Figure 4: Data pages and index pages in the proposed structure;

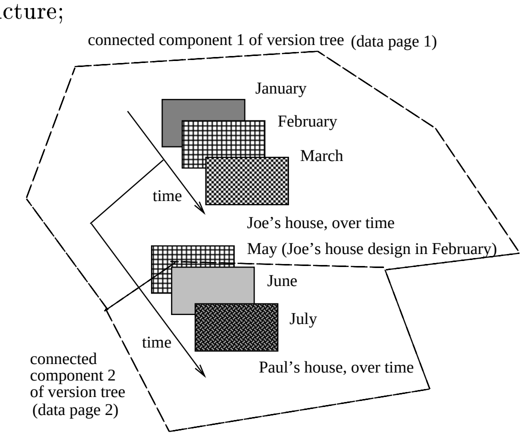

Figure 5: A version tree is divided into subtrees.

To efficiently support typical queries, the version tree is divided into connected components, each of which is rooted at a certain version. For example, Figure 5 shows that the version tree in Figure 2 is divided into two connected components with one rooted at version *(Joe's house, Jan.)* and another one rooted at version *(Paul's house, June)*. The data space is divided into key ranges. Each disk page in our structure, whether an index page or a data page, will corresponds to a connected component of the version tree and a key range. For example, data page 1 in Figure 4 corresponds to the connected component 1 in Figure 5 and the entire data space [bedroom, kitchen], data page 2 in Figure 4 corresponds to the connected component 2 in Figure 5 and the entire data space [bedroom, kitchen], while the index page in Figure 4 corresponds to the entire version tree and the data space.

The index page in Figure 4 only contains one version node *(Paul's house, June)* and two pointers. The right pointer channels all the searching for data records in version *(Paul's house, June)* and its descendent versions. The left pointer channels all the searching for data records in the set of versions that are not descendents of version *(Paul's house, June)*, say *(Joe's house, Jan.)*.

Data records of versions that are not descendents of version *(Paul's house, June)* are all stored in data page 1. From the version tree shown in Figure 2 or Figure 5, we know that these versions include three versions of Joe's house in January, February and March separately, and Paul's house in May. Considering the assumption that each house contains only two keys: bedroom and kitchen, data page 1 should store information about 8 data records in total. However only 5 data records are stored in data page 1. This is because some of the data records are shared among different versions. For example, Joe's kitchen in January is shared among three versions: *(Joe's house, Jan.)*, *(Joe's house, Feb.)* and *(Paul's house, May)*.

Data page 2 contains data records in two versions of Paul's house in June and July. June's version has the bedroom of Paul's house in May which is a record that is copied to data page 2 from data page 1 through data page splitting policies, while data record "Paul's kitchen in June" is shared between two versions *(Paul's house, June)* and *(Paul's house, July)*.

Figure 4 shows a simple case of our structure where only version information is needed inside the index page. We will also need to distinguish the different key ranges covered by data pages.

The access structure with data pages and index pages as outlined above is called a **BT-tree (Branched and Temporal Tree)**. The rest of the paper describes the BT-tree in detail.

1.1 Background and Previous Work

Branched-and-temporal indexing is a relatively unexplored area. However, many access methods (for example, [3], [1], [6], [10] and [7]) have been proposed for temporal data. A survey and comparison of these access methods can be found in [8]. These methods have effectively solved the problem of providing access to versioned record sets where the versioning is actually linear, i.e, no branching.

Other approaches to managing versioned data are less closely related. Work on "version management," for example, in software engineering, does not consider efficient use of disk pages. The paper [4] does not consider pagination and, although the structure is "branched," only a current version can be split into branches. Old versions can not be modified. Driscoll et al. [2] develop techniques for making linked data structures (e.g. binary search trees) fully persistent (all versions can be read and updated).

Perhaps closest to our work is that of Lanka and Mays [5], which is based on ideas from [2]. Lanka and Mays' "fully persistent $B+$-tree" maintains multiple versions of B+-trees. The Fully persistent $B+$-tree is a *branched-only* access method. Branched-only data structures can be used for branched-and-temporal data if the versions are made to correspond to a branch and a timestamp. Lanka and Mays did not suggest this. There is no access method in the literature explicitly proposed for branched-and-temporal data. In addition, fully persistent $B+$-trees have extra "version block" nodes in the search path making them less efficient than our BT-tree. Their data nodes also store some redundant information, making the total space usage greater than ours. Furthermore we provide an ancestor determination method which exploit the lesser amount of branching found in a branched-and-temporal database. The paper [5] maintains a full version tree for ancestor determination. This is too space-and-compute expensive for our case.

1.2 Organization of This Paper

The rest of the paper is organized as follows. Section 2 contains the description of the BT-tree including the ancestor determination method, the structure of data pages and index pages, the splitting algorithm and the consolidation algorithm. Performance results are presented in section 3.

2 The BT-tree

For the purpose of this paper, time is assumed to be discrete, described by a succession of nonnegative integers. Each branch is assigned a unique branch id, which is represented by a positive integer. A combination (B, T) of a branch identifier B and a time stamp T is called a **version**. A branch typically has a large number of versions, one for each time stamp when a change was made in that branch.

The first problem we encounter in designing such a branched and temporal index method is that versions (B, T) in the structure are only partially ordered. The lack of a linear ordering on versions makes navigation through a representation of a fully persistent BT-tree structure problematic. We need to be able to decide whether or not one version is a descendent or ancestor of another.

2.1 Ancestor Determination

Ancestor determination methods used in [2] and [5] for branched data use $O(n)$ space. Their methods are not suitable for branched-and-temporal case where the number of timestamps is large (hence the corresponding total number of the versions is large). Our approach of solving the ancestor problem is designed for the case when the number of branches is small although the number of timestamps may be large.

To add understanding, we define the version tree first. Unlike the version tree in the branched case, where every node corresponds to a branch, every node of the version tree here corresponds to a pair (B, T). An edge from node (B, T) to node (B', T') exists if version (B', T') is obtained by updating version (B, T). Figure 6 shows an example of version tree where only one branch exists.

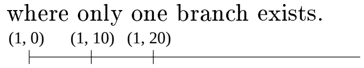

Figure 6: An example of version tree where only one branch exists.

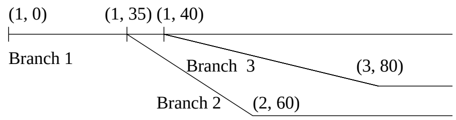

Figure 7: The version tree corresponding to the branch table in Table 1 below.

The main idea is to use a branch table, which contains one entry for each branch. Each entry consists of four items: a branch id, the branch id of its ancestor branch, the start time of this branch and the share time which is the time when this branch shares information with its ancestor branch. For example, the branch table entry for branch 2 is $(2, 1, 60, 35)$. This means that branch 2 is created out of version $(B, T) = (1, 35)$ at time 60. Given the branch table entry for branch 2, a version $(2, T)$ is valid only if $T \geq$ start_time=60. An example of branch table with three branches is shown in Table 1.

Branch id	Ancestor branch id	Start time	Share time
1	1	0	0
2	1	60	35
3	1	80	40

Table 1: Branch table for the running example.

When a new branch i is created out of version (j, T) at time T', a new entry (i, j, T', T) is generated and appended at the end of the branch table. Each branch table defines a subset of a version tree, indicating only the branching. Not all versions are listed as entries, since a version which is created as an update of a current version (with the same branch number, but a new time stamp) is not listed. This is why the large number of versions in a branched-and-temporal structure can be captured in a small table. A version tree containing only branching information corresponding to the branch table in Table 1 can be found in Figure 7.

Definition 2.1.1 *The branch id of the ancestor branch found in the branch table entry of branch i is called* **Direct_ancestor(i)**. *The share_time found in the branch table entry of branch i is called* **Share_time(i)**. *The start_time found in the branch table entry of branch i is called* **Start_time(i)**.

For example, given the branch table shown in Table 1, Direct_ancestor(2)=1, Share_time(2)=35 and Start_time(2)=60.

Definition 2.1.2 *Given two versions (B_1, T_1) and (B_2, T_2) with $B_1 \leq B_2$,* **(B_1, T_1) is an ancestor of (B_2, T_2)** *if ($B_1 = B_2$ and $T_1 \leq T_2$) or (B_1, T_1) is an ancestor of version (Direct_ancestor(B_2), Share_time(B_2)). If (B_1, T_1) is an ancestor of (B_2, T_2), then* **(B_2, T_2) is a descendant of (B_1, T_1)**.

This definition is used to develop the following algorithm **Ancestor(v1,v2)** for ancestor determination. **Ancestor(v1,v2)** returns true if $v1 = (B1, T1)$ is an ancestor of $v2 = (B2, T2)$.

```
If B1=B2 then { if  T2<=T1 then return FALSE
                            else return TRUE }
If B2<B1 then return FALSE
else return(Ancestor(v1,
     (Direct_ancestor(B2), share_time(B2))))
```

This algorithm requires following the direct ancestor path of a version ($B2$, $T2$) upwards in the branch table until it can be determined whether or not ($B1$, $T1$) is an ancestor. Since the total number of branches is small, the number of ancestors of a given branch is small and this search will be fast.

Definition 2.1.3 Ancestor(B,T) *is the collection of versions which are ancestors of version (B, T). i.e, if (B_1, T_1) $\in$ Ancestor(B_2, T_2), then (B_1, T_1) is an ancestor of (B_2, T_2).*

2.2 Overview of the BT-tree

The structure of the BT-tree is a directed acyclic graph of pages, including index pages and data pages. Fully persistent structures are traditionally called "trees" because the restriction to one version is a tree. In addition, there is one distinguished page called *the root* and a set of *leaf pages* which are those pages with no outgoing edges. Leaf pages are data pages and they are the furthest pages from the root. Furthermore, all leaves are the same distance from the root. Hence the BT-tree is balanced.

Data pages contain branched-and-temporal data while index pages contain a small binary tree channeling search to lower level pages. Initially the BT-tree starts from one index page and one data page, where the data page is empty and the index page contains only one node pointing to the data page. As time proceeds, data records are continuously added to the data page. A some point when there is no additional space to insert new data into the data page, a *data page overflow* happens.

Data page overflow needs special handling: a split is performed on the overflowed data page. One or two new data pages will be created and a small portion of data from the overflowed data page will be copied to the new data page(s). Information about the data page split will be posted up to the upper level index page to direct future searches to either the old overflowed data page or the newly created page(s).

As new data is added to data pages, more and more data pages overflow and are therefore split, consequently more and more information is posted to upper level index pages to record the splitting history of data pages. At some point *index page overflow* happens.

Index page overflow needs special handling: a split is performed on the overflowed index page. One or two new index pages will be created from the index page splitting. A tree will be extracted and copied to the new index page(s). The tree copied to the new index page(s) points to old pages as well as new pages therefore making the BT-tree a directed acyclic graph of pages instead of a tree of pages.

An example will be given in the index page splitting section.

As a result of the index split, information will be posted to upper level index pages to record the split history. The split may percolate up as needed. A new root will be created when the root index page is split.

Now we give a detailed explanation of data pages and index pages and their splitting algorithms.

2.3 Data Pages

Data pages contain branched-and-temporal data. Our data consists of **record variants**. For our purpose a **record variant** is characterized by four entries: a time-invariant part called a **key**, a branch id, a time stamp and an information field. For example, $(a, 3, 80, info)$ is a record variant with key = a, branch id = 3, time stamp = 80 and *info* representing the data content of this record variant.

A page is identified with a key range and a connected component of version tree. For example, the data page $D2$ shown in Figure 10 is identified with key range $[a, d]$ and the connected component of version tree enclosed in the bigger dotted boundary, denoted as $V1 - 20$, in the right side of the figure. Version tree components are named with the branch (in this case branch 1) and the time stamp (in this case 20) of their root.

Definition 2.3.1 *A record variant $(k, b, t, info)$ is said to be* **alive in a connected component of version tree and a key range** *if*

- *k is within the key range and*
- *either*
  - *(b,t) is within the connected component of the version tree or*
  - *Let version (B', T') be the root of the connected component of the version tree. Then (b, t) $\in$ Ancestor(B', T'), and $\forall$ record variants $(k', b', t', info)$ with $k' = k$ and $(b', t') \in$ Ancestor(B', T'), $t \geq t'$.*

The second condition in Def 2.3.1 implies that either the version (b, t) is contained in the connected component of the version tree or it is the most recent ancestor of the root of the connected component of the version tree on a record variant with the same key. In our example, the connected component of the version tree is *V1-20*, from Figure 10. Let's consider the only two record variants in the database with key d: $(d, 1, 2, info)$ and $(d, 1, 3, info)$. The root of *V1-20* is *(1,20)*. Record variant $(d, 1, 2, info)$ is *not* alive in the connected component of version tree *V1-20* and key range $[a, d]$ because $(d, 1, 3, info)$ has the same key and has a version which is a more recent ancestor of $(1, 20)$. Record variant $(d, 1, 3, info)$ is alive in *V1-20* and $[a, d]$ because its version *is* the most recent ancestor of $(1, 20)$ for key = d. Record variant $(a, 2, 60, info)$ is alive in *V1-20* and $[a, d]$ because its version is contained in *V1-20*.

Each data page contains copies of all the record variants alive in a connected component of version tree and a key range. For example, data page $D2$ in Figure 10 corresponds to the component *V1-20* and the key range $[a, d]$. $D2$ contains record variants $(d, 1, 3, info)$ and *not* $(d, 1, 2, info)$ because $(d, 1, 3, info)$ is alive in *V1-20* and $[a, d]$ while $(d, 1, 2, info)$ is *not* alive in *V1-20* and $[a, d]$.

Data pages partition the entire version tree and data space for which the database is defined. For example, in Figure 10, both data pages cover the key range $[a, d]$ and the two connected components of the version tree, *V1-0* (corresponding to page D1) and *V1-20* (corresponding to page $D2$) are disjoint and cover the version tree.

Record variants in a data page are ordered by key, branch and time stamp inside a data page. When a new record with key k is added to the data page at time t (= current time) by branch i, a new record variant of the form $(k, i, t, info)$ is created and inserted in a proper position so that the all data variants are in order. When a data page becomes full, a data page splitting occurs.

2.4 Data Page Splitting

We distinguish **updates**, which create a new record variant for an existing key and **inserts**, which create a new record variant with a new key. When a new update or insertion, say $(k, i, T, info)$ (T is current time,) into a data page causes the page to become over-full, it will be split at (i, T) with one or two new data pages allocated. Information about the split will be posted to the parent page that channeled the search to the split data page.

Definition 2.4.1 *A record variant $(k, j, t, info)$ in a data page is said to be* **alive at version** (i, T) *with T = current time if $(j, t) \in Ancestor(i, T)$ and $\forall$ record variant $(k', j', t', info)$ in the data page with $k' = k$, and $(j', t') \in Ancestor(i, T)$, $t \geq t'$.*

For example, in Figure 10, record variant $(a, 3, 83, info)$ in data page $D2$ is alive at $(3, T)$ (say T is current time 90), while the same record variant is not alive at $(2, T)$ because $(3, 83)$ is not an ancestor of $(2, T)$.

To split a data page at (i, T), only alive record variants at (i, T) are copied to the new pages. Copying the alive record variants into *one* new page is called a **version split**.

If there are not many alive record variants in the full page (because in this page, most of the record variants are old variants of alive record variants) only one new page is needed. If there are many alive record variants or (considering variable-length records) if the space occupied by alive record variants is too large, two new pages are allocated and a B^+-tree-like key split is made among the alive record variants being copied. This is called a **version-and-key split**. We define a threshold utilization U for alive record variants. When alive record variant utilization exceeds U, we do a version-and-key split. When alive record variant utilization is less than or equal to U, we do a version split. A typical value for U is about .66, which guarantees that a new data page will be at least one third full of current data.

BT-tree data page splitting is illustrated in Figure 8, Figure 9, Figure 10, and Figure 11. The database starts with an empty data page $D1$. Figure 8 shows the insertion of 6 record variants in data page $D1$ up to current time T = 19. Only branch 1 is created so far and data page $D1$ is full. At time 20, in order to insert record variant $(a, 1, 20, info)$, a version split occurs to data page $D1$. Consequently, data page $D2$ is created, as shown in Figure 9. Only those record variants in data page $D1$ which are alive at $(1, 20)$ are copied to the new page, data page $D2$. The new record variant is also inserted into the new data page $D2$.

Figure 10 shows that branch 2 is created and a new record variant $(a, 2, 60, info)$ is inserted into data page $D2$ at time 60. At time 80, branch 3 is created and a new record variant $(a, 3, 80, info)$ is inserted into data page $D2$. At time 83, The insertion of another record variant into data page $D2$ by branch 3 makes the data page $D2$ a full data page. The version tree grows as new branches are created.

In Figure 11, as we are trying to insert record variant $(b, 3, 85, info)$ into the data page $D2$ which is already full, data page $D2$ has to be split. A version-and-key split at version $(3, 85)$ and key c occurs here generating two new pages $D3$ and $D4$ corresponding to same connected component of the version but different key ranges.

A data page could be split more than once if it is already full and new branches are created from some version in its version-tree component. This cannot happen in temporal (not branched) structures.

2.5 Index Pages

Index pages also represent connected components of version tree and key ranges, with a full partition of the version-key space at each level of the tree. Within each BT-tree index page are index nodes that identify the connected component of version tree and key range for each child page, and hence channel searches to its children pages.

A **split history** tree or **sh-tree** is used within each index page. The sh-tree is a small binary tree. The sh-tree, describing the history of the splits of its children, contains three types of 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.

Initially the BT-tree only has one index page I, with one leaf node as shown in Figure 12(a), referencing the only data page $D1$ in Figure 11. As new data is added into the data page $D1$, $D1$ becomes full and a data page split occurs. Whenever a data page split occurs, the sh-tree in the parent index page is changed so that it reflects the splitting history of its children.

In case of a version split of a data page at (i, T) (T is current time,) the parent's reference to the old page is replaced by a new vsh node (i, T). This node has one child referencing the old page while the other references the new page. Figure 12(b) shows that the vsh node $(1, 20)$ is posted to index page I (which is also root index page of this BT-tree) when data page $D1$ is split and data page $D2$ is generated as shown in Figure 9.

In case of version-and-key split at (i, T) (T is current time) and key k, the parent page's reference to the old page is also replaced by a new sh-tree vsh node (i, T). This node has its left child referencing the old page while the right child is a ksh node with key value k referring to the two new pages. Figure 12(d) shows how posting happens when data page $D2$ is version-and-key split at version $(3, 85)$ and key c generating data pages $D3$ and $D4$ as shown in Figure 11.

A vsh node (b, t) in an index page divides the lower level BT-tree rooted at (b, t) into two parts, with the right subtree of the vsh node containing everything which is a descendent of (b, t) while the left subtree of the vsh node contains everything which is not a descendent of (b, t). Similarly, a ksh node k divides the lower level BT-tree rooted at k into two parts, with the right subtree of the ksh node containing record variants with key value greater than or equal to k while the left subtree of the ksh node contains record variants with key value less than k. The search algorithm for a single point (K, B, T) in a BT-tree is in Figure 13.

1. Start at the root page of the BT-tree.
2. BT-tree index page has been reached: Start at the root node of the sh-tree in this BT-tree page.
3. Sh-tree node has been reached: Test the type of sh-tree node.
   (a) ksh: If $K \geq$ key value in ksh, go right, else go left. Go to step 3.
   (b) vsh (B', T'): If $(B', T') \in Ancestor(B, T)$, go right, else go left. Go to step 3.
   (c) sh-tree leaf: Fetch the disk page P in the sh-tree leaf. If P is another index page, go to step 2. Otherwise continue.
4. BT-tree data page has been reached: Find the record variant in this data page with key value K and alive at version (B, T).
   - If there is none, return indicating the search is unsuccessful.
   - If this is a deletion record variant, the record did not exist in the database at time T for version B. Return indicating the search is unsuccessful.
   - Otherwise, return the record variant and indicate that the search is successful.

Figure 13: Search in the BT-tree for (K, B, T), a record variant with key K alive at version (B, T).

2.6 Index Page Splitting

An index page records the split history of data pages by getting one (in case of version-split) or two (in case of version-and-key split) index node(s) posted whenever a child data page splits. Index page splitting happens when an index page overflows. Only disk addresses of children (leaf nodes) **alive at the splitting version** and the key boundaries separating them are copied.

Definition 2.6.1 *A child (leaf) of an sh-tree in an index page is* **alive at (B, T) (T= current time)** *if (B, T) is in the connected component of the version tree associated with the child page whose disk address recorded in the leaf node of the sh-tree.*

Consider a BT-tree with root index page in Figure 12 (d) and four data pages in Figure 11. Suppose the current time is 90. To find children (leaves) of the sh-tree in Figure 12 alive at $(3, 90)$, we look at the connected components of version trees in Figure 11. Since $(3, 90)$ is in the connected component of the version tree corresponding to $D3$ and

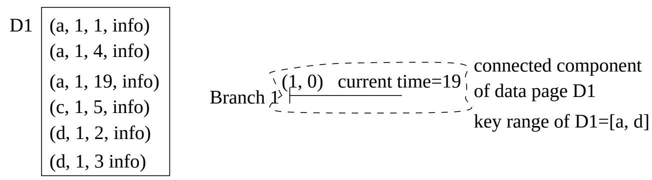

Figure 8: Data page $D1$ is full at this point.

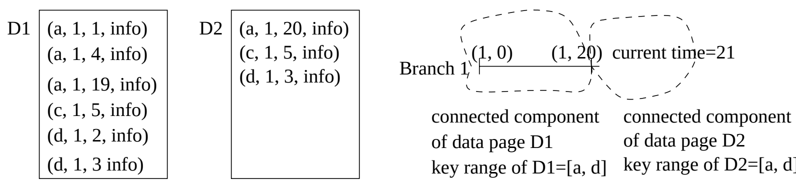

Figure 9: In order to insert a new record variant $(a, 1, 20, info)$, data page $D1$ has to be version split. Data page $D2$ is generated.

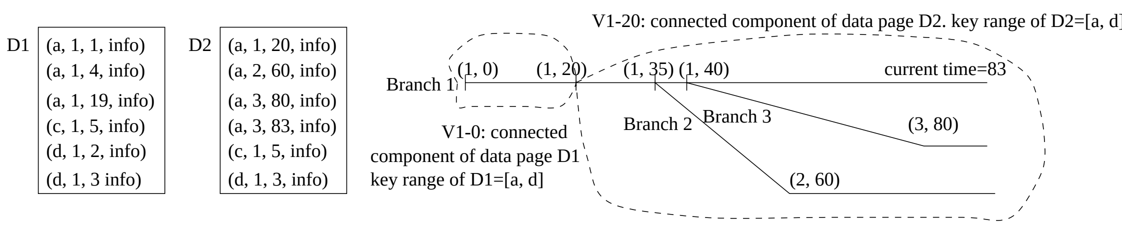

Figure 10: Branch 2 and 3 are created and new record variants are inserted into data page $D2$ causing it to become full at time 83.

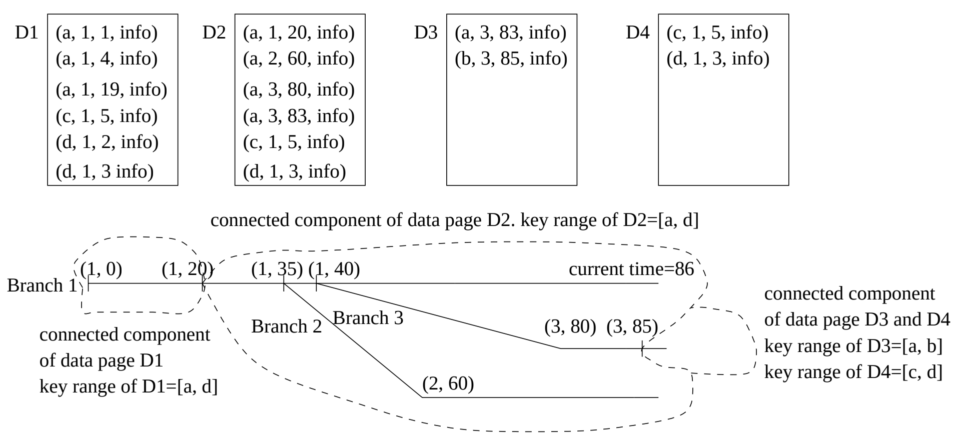

Figure 11: Data page $D2$ is version-and-key split at $(3, 85)$ as new record variant $(b, 3, 85, info)$ is attempted to be inserted into data page $D2$.

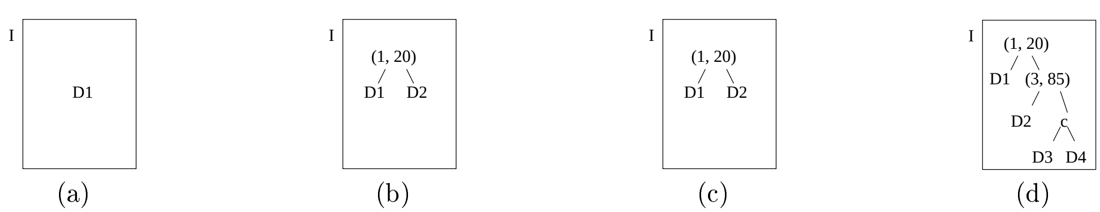

Figure 12: The evolution of index page I corresponding to data pages in (a) Figure 8; (b) Figure 9; (c) Figure 10; (d) Figure 11. The connected component of index page I is the entire version tree and the key range of index page I is [a, d].

$D4$, leaves $D3$ and $D4$ in the sh-tree in Figure 12 (d) are alive at $(3, 90)$. However $(3, 90)$ is not in the connected component of version trees corresponding to $D1$ and $D2$, hence leaves $D1$ and $D2$ are not alive at $(3, 90)$.

If there are too many alive children found when splitting an index page, a key split is also made. Split information is posted to the parent as usual. When a root page is split, a new root page is allocated to hold the split information, as in the B^+-tree. Root-page splits thus increase the height of the BT-tree.

When we version split an index page, the new index page has a sh-tree which only has ksh nodes. The algorithm for BT-tree index-page split (split an index page at version (B, T) (T is current time)) is in Figure 14. The effect of this algorithm is to obtain one (in case of version-split) or two (in case of version-and-key-split) binary search tree(s) on key only referring only to the *alive* children.

1. Start at the root of the sh-tree of the full index page.
2. If a vsh (B', T') is encountered, do not copy the vsh.
   (a) If $(B', T') \in Ancestor(B, T)$, go right.
   (b) Otherwise, go left.
3. If a ksh is encountered, copy the ksh to the new index page and process both subtrees recursively.
4. When a new sh-tree is constructed in the new index page, the key ranges and their corresponding disk addresses are known. Construct a balanced sh-tree with these key ranges. If two such trees are needed, for two new sh-tree index pages, split at the middle key range and construct two balanced sh-trees, one for each new sh-tree index page.

Figure 14: Algorithm for splitting a BT-tree index page at version (B, T) (T is current time.)

Figure 15 illustrates the resulting index page I, evolved from the index page I shown in Figure 12(d), after page $D2$ had a version split at version $(2, 88)$, and page $D3$ had a version-and-key split at version $(3, 95)$ and key b. Assume that index page I is about full at this point. At time 98, branch 3 inserts or updates a record variant in data page $D4$ causing it version-and-key split at version $(3, 98)$ and key d. While attempting to post the sh-tree shown in Figure 15 to index page I, the index page I overflows. Therefore index page I is version split at version $(3, 98)$, as shown in Figure 16, creating a new index page with four children. Vsh node $(3, 98)$ is posted up to the new root page created.

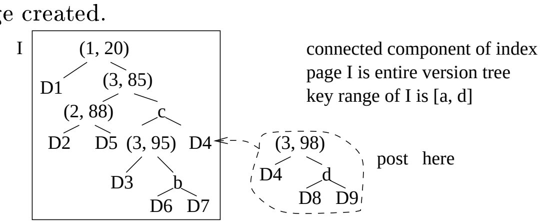

Figure 15: This is the index page I, shown in Figure 12(d), after $D2$ had a version split at version $(2, 88)$ and page $D3$ had a version-and-key split at $(3, 95)$ and key b. Now data page $D4$ are having a version-and-key split at version $(3, 98)$ and key d. When attempting to post the vsh node and the ksh node up to index page I, we find that the page is full.

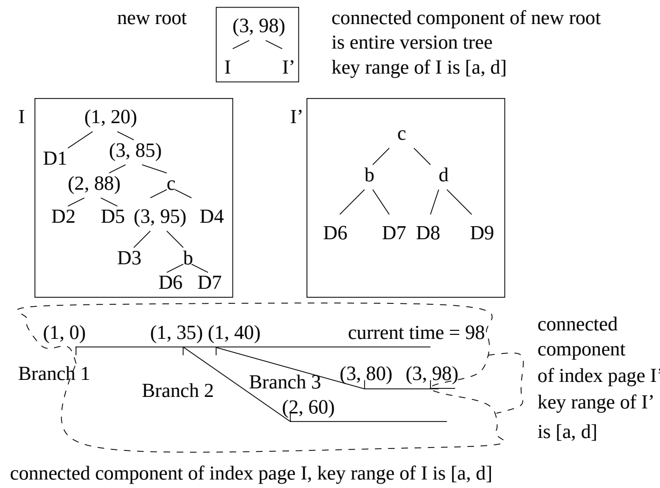

Figure 16: Version split index page I at version $(3, 98)$ creating new page I'. A new root is created and vsh node $(3, 98)$ is posted.

2.7 Discontinued Record Variants

Record variants are never physically deleted. When the most recent record variant with a given key is to be discontinued, a new record variant is inserted with the key, the branch id, say B, of the branch in which the record variant is deleted, the time stamp, say T, of the delete, and a **delete marker** indicating it is marking the end time of the most recent previous variant within certain branch. Delete marker record variants are necessary in data pages which include the key in their key range and include the version (B, T) in the corresponding connected component of version tree. This is the only way to tell that a record variant has been discontinued by branch B at time T, i.e. that its prior record variant is no longer alive. That is, the delete marker record variant bounds the time interval of the preceding record variant existing in a specific branch.

If a page where a delete marker record variant is version-split, we can choose to treat the delete marker record variant like any other record variant and copy it to the new page. This permits us easily to answer queries about the history of record variants with a given key. However, if deletes are common, this dilutes the number of actual record variants that a page can hold. Hence, here we choose not to copy the record variants with delete markers. Version queries will still be correct. Say a record variant with key value k is deleted at version (B, T). We have a data page D that every version within the connected component of version tree corresponded to data page D is a descendent of version (B, T). The absence of a record variant with key value k in such a page indicates that the record is not alive at that time within version B.

2.8 Page Consolidation

When a version query finds only a small number of record variants satisfying the query in each visited data page, query performance is poor. Low density of correct variants for a given version is caused by deletion. Consolidation guarantees good version query performance for the BT-tree.

A data page is **sparse at version** (B, T) when the space occupied by *alive variants* at version (B,T), not including delete marker variants, falls below a threshold. To consolidate a page P sparse at (B,T), both P and a sibling page are version split at (B,T). (A **consolidating sibling** must be a page whose corresponding segment of the version tree contains (B,T) and which has the same parent as P and an adjacent key range.[1]) Copies of the alive record variants at version (B,T) from both the sparse page and its consolidating sibling are combined in a new page or, should that result in an over-full page, we then key-split the new consolidated page. This is similar to consolidation in [5]. Since more copies are made, page consolidation degrades space utilization in order to improve query performance. A page consolidation threshold of t "guarantees" that the space occupied by record variants alive at any given query version in any given data page will not fall below t.[2]

Non-root index pages can also be consolidated. This can be done when posting information about a lower level consolidation at version (B,T) results in the index page having too few alive children in the same version. The index page is version split at (B,T) and combined with a sh-tree copy resulting from a version split of one of its siblings at (B,T).

3 Performance

We present the results of our performance study on the BT-tree. The parameters of the system are described first. Graphs and explanations follow.

3.1 System Parameters

We assume all record variants, including delete marker 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 (in one branch and with current time).

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

Let the number of branches in the system be denoted "B". The maximum number of variants per page is b. In our case, b is 35. R is the total number of non-redundant record variants, including delete marker record variants. R includes different variants of records with the same key. R is 50,000 here. $K(i)$ is the number of record variants alive at version (i,T) (T is current time.) K does not include records that have been deleted. Let N be the total number of data pages in the BT-tree and let $N_c(i)$ be the number of data pages containing record variants alive at version (i,T) (T is current time) in this BT-tree.

We measure total space cost by **multiversion total utilization(MVTU)**. Keeping every distinct record variant (including delete marker record variants) is needed to support arbitrary version slice queries and historical queries. MVTU measures the fraction of the total data space occupied by distinct record variants.

$$MVTU = \frac{R}{N \times b}$$

Every version was once the current version in a branch and every version query will access only those data pages, which were current at that time in that branch. Hence, version query cost is captured by **single version current utilization** for branch i (**SVCU(i)**) (the fraction of a branch's data pages containing record variants alive at version (i,T) (T is current time) occupied by these alive record variants).

$$SVCU(i) = \frac{K(i)}{N_c(i) \times b}$$

The value of SVCU may vary from one branch to another. **Average SVCU ($\overline{\textbf{SVCU}}$)** is the average SVCU value across different branches. $\overline{SVCU}$ measures the average version slice query efficiency.

$$\overline{SVCU} = \frac{\Sigma_{i=1}^{B} SVCU(i)}{B}$$

The **horizontal query current utilization** (**HQCU**) and the **vertical query current utilization** (**VQCU**) are also defined to measure the horizontal query efficiency and the vertical query efficiency. Because of space limitation, the definitions and the performance results on $HQCU$ and $VQCU$ are not detailed in this section.

3.2 Performance Results

In order to study the performance of the system under different circumstances, three sets of experiments were carried out. The first set of experiments measures performance as the number of branches increase. The second set of experiments measures performance as the ratio of updates version insertion varies (no deletes are allowed) The third set of experiments allows deletes and examine the effect of consolidation. Because of space limitation, we only present the second set of experiments.

The number of branches are fixed to be 10. All branches other than the first are randomly created between the 10,000th and 20,000th 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. The tree height for the BT-tree is always 3 in this experiment.

Figure 17 shows the $MVTU$ curves for early branch creation profile. We first remark on the general property of the BT-tree $MVTU$ curves. As the update rate increases, the $MVTU$ value of the BT-tree increases. In other words, the total amount of space occupied by 50,000 records decreases. The reason is as follows. when the update rate increases, we have more transactions update existing record variants instead of inserting new record variants with different key values. Therefore, considering one data page, there will be a smaller number of distinct key values as the update rate

[1] In the case that the key range of the parent equals the key range of the child, a page has no consolidating sibling, making page consolidation impossible. This does not impact search correctness as page consolidation is needed only for performance.

[2] Disregarding the case when page consolidation is impossible.

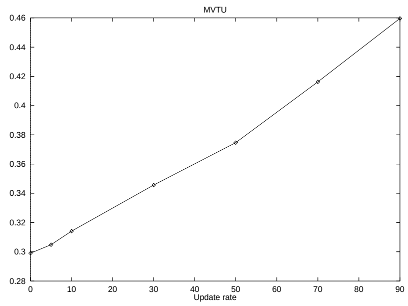

Figure 17: No Deletes: MVTU for early branch creation.

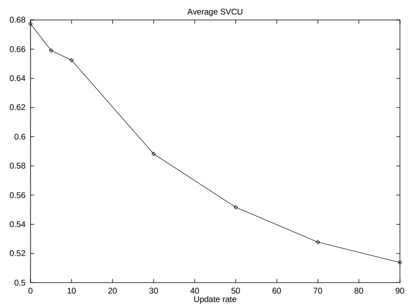

Figure 18: No Deletes: $\overline{SVCU}$ for early branch creation.

increases. Consequently, when this data page is full and split, there will be fewer record variants copied to the new page. When the total number of record variants copied is less, the $MVTU$ value is higher.

The space needed is at worst three times the minimal amount when the update rate is low, i.e. mostly insertions of record variants with new keys, and at best about twice the minimal amount when there are mostly updates of existing record variants.

Figure 18 shows the $\overline{SVCU}$ curve. As the update rate increases, the $\overline{SVCU}$ value decreases. The reason is as follows. When the update rate increases, distinct key values in a data page decrease. Therefore record variants alive at current versions are not compactly clustered, making the version slice query less efficient. The percent of the found data pages which is occupied by answers to the version slice query is at worst near 50% when there is a high update rate. In this case, much of the other space is occupied by record variants which are not alive in the query version. At best, it is around 65% when the update rate is 0.

4 Conclusions

There are many database applications that require the support of branched and temporal data. Since branched and temporal data increases in size as time proceeds, efficient indexing is important.

In this paper, we have presented the BT-tree, a new paginated method for storing and accessing branched and temporal data. Each data page and index page in BT-tree corresponds to a connected component of the version tree and a key range. At each level of the tree, the pages partition the version-data space so that each point (K, B, T) (key, branch, time stamp) is in exactly one page. Performance results show that the BT-tree provides a reasonable trade off between space and access time.

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.

[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.

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

[4] 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.

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

[6] 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.

[7] 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.

[8] Betty Salzberg and Vassilis J. Tsotras. A comparison of access methods for time evolving data. *Computing Surveys*, March 1999.

[9] R. Snodgrass and I. Ahn. Temporal databases. *IEEE computer*, pages Vol. 19, No. 9, pp 35–42, 1986.

[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.