Querying Transaction–Time Databases under Branched Schema Evolution

Wenyu Huo and Vassilis J. Tsotras

Department of Computer Science and Engineering,
University of California, Riverside, USA
{whuo,tsotras}@cs.ucr.edu

Abstract. Transaction-time databases have been proposed for storing and querying the history of a database. While past work concentrated on managing the data evolution assuming a static schema, recent research has considered data changes under a linearly evolving schema. An ordered sequence of schema versions is maintained and the database can restore/query its data under the appropriate past schema. There are however many applications leading to a *branched* schema evolution where data can evolve in parallel, under different concurrent schemas. In this work, we consider the issues involved in managing the history of a database that follows a branched schema evolution. To maintain easy access to any past schema, we use an XML-based approach with an optimized sharing strategy. As for accessing the data, we explore branched temporal indexing techniques and present efficient algorithms for evaluating two important queries made possible by our novel branching environment: the vertical historical query and the horizontal historical query. Moreover, we show that our methods can support branched schema evolution which allows version *merging*. Experimental evaluations show the efficiency of our storing, indexing, and query processing methodologies.

1 Introduction

Due to the collaborative nature of web applications, information systems experience evolution not only on their data content but also under different schema versions. For example, Wikipedia has experienced more than 170 schema changes in its 4.5 years of lifetime [5]. Schema evolution has been addressed for traditional (single-state) database systems and issues on how data is efficiently transferred to the latest schema have been examined [4]. Consider however the case where the application maintains its past data (typically for archiving, auditing reasons etc.) which may have followed different schemas. A temporal database can be facilitated to manage the historical data, but issues related to how data can be queried under different schemas arise. The pioneering work in PRIMA system [8] addresses the issues of maintaining a transaction-time database under schema evolution by introducing: (i) an XML-based model for archiving historical data with evolving schemas, (ii) a language of atomic schema modification operators (SMOs), and (iii) query answering and rewriting algorithms for complex temporal queries spanning over multiple schema versions. Nevertheless,

© Springer-Verlag Berlin Heidelberg 2012

PRIMA considers only a linear evolution: a new schema is derived from the latest schema and at each time there is only one current schema.

In many applications, however, the schema may change in a more complex way. For instance, in a collaborative design environment, an initial schema may be branched into a number of parallel schemas whose data can evolve concurrently. Another common case of non-linear evolution is in software development management. Revision control enables the modifications and developments happening in parallel along multiple branches. The release history of Mozilla Firefox [1] shows that 10 branches of versions have been developed and 4 more branches are on the way.

In this paper we address the issues involved in archiving, managing and querying a branched schema evolution. In particular, we maintain the branched schema versions in an XML-based document (*BMV-document*) using schema sharing. This choice was made because the number of schema changes is relatively smaller than data changes and the hierarchal structure of XML allows for easy schema querying. The data level changes are stored in column-like tables (*BC-Tables*), one table for each temporal attribute, with the support of applicable temporal indexing. To the best of our knowledge, this is the first work to examine both data and schema evolution in a branched environment. Our contributions can be summarized as:

1. We utilize a *sharing* strategy with *lazy-mark* updating, to save space and update time when maintaining the schema branching.
2. We employ branched temporal indexing structures and link-based algorithms to improve temporal query processing over the data. Moreover, we propose various *optimizations* for two novel temporal queries involving multiple branches, the *vertical* and *horizontal* queries.
3. We further examine how to support *version merging* within the branched schema evolution environment.
4. Our experiments show the space effectiveness of our sharing strategy while the optimized query processing algorithms achieve great data access efficiency.

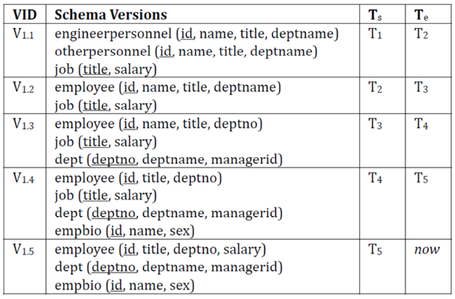
The rest of the paper is organized as follows. Section 2 summarizes work on linear schema evolution (PRIMA). Section 3 introduces branched schema evolution while section 4 presents the BMV-Document for storing schema versions and the BC-Tables for storing the underlying data changes (with the support of branched temporal indexing). Section 5 provides algorithms and optimizations for efficient processing of temporal queries. The merging challenges are discussed in section 6 and the experimental evaluations are presented in section 7. Finally, conclusions appear in section 8.

2 Preliminaries

2.1 A Linear Evolution Example

Consider the linear schema evolution shown in Table 1 and Fig. 1(a), of an employee database, which is used as the basic running example in this paper. When the database was first created at T_1, using schema version $V_{1.1}$, it contains three tables: **engineerpersonnel**, **otherpersonnel** and **job**. As the company seeks to uniformly manage the personnel information, the DBA applies first schema modification at T_2, which merges two tables **engineerpersonnel** and **otherpersonnel**, producing schema $V_{1.2}$. Each schema version is valid for all times between its start-time T_s and its end-time T_e (the time it was updated to a new schema). The rest schema versions and their respective time intervals appear as well until the latest schema $V_{1.5}$. A special value "now" is used to represent the always increasing current time.

Table 1. A linearly evolving employee database



VID	Schema Versions	T_s	T_e
$V_{1.1}$	engineerpersonnel (id, name, title, deptname) otherpersonnel (id, name, title, deptname) job (title, salary)	T_1	T_2
$V_{1.2}$	employee (id, name, title, deptname) job (title, salary)	T_2	T_3
$V_{1.3}$	employee (id, name, title, deptno) job (title, salary) dept (deptno, deptname, managerid)	T_3	T_4
$V_{1.4}$	employee (id, title, deptno) job (title, salary) dept (deptno, deptname, managerid) empbio (id, name, sex)	T_4	T_5
$V_{1.5}$	employee (id, title, deptno, salary) dept (deptno, deptname, managerid) empbio (id, name, sex)	T_5	*now*

Schema changes are represented by Schema Modification Operators (SMOs) [4]; each operator performs an atomic action on both the schema and the underlying data, like CREATE/MERGE/PARTITION TABLE, ADD/DROP/RENAME COLUMN. For example, two tables in $V_{1.1}$ were merged to one table by a MERGE TABLE operation in $V_{1.2}$. In the following discussion we will use the term SMO to denote a change operator applied to one schema without detailing which SMO was actually used.

2.2 XML Representation of a Linear Schema Evolution

The history of the relational database content and its schema evolution can be published in the form of XML, and viewed under a temporally grouped representation whereby complex temporal queries can be easily expressed in standard XQuery [8, 9]. The MV-Document [8] intuitively represents both schema versions and data tuples using XPath notation, as: **/db/table-name/row/column-name**. Each of the nodes, representing respectively: databases, tables, tuples, and attributes, has two more attributes, start-time (ts) and end-time (te), respectively representing the (transaction-) time in which the element was added to and removed from the database.

Consider our running example: when the three-table schema in version $V_{1.1}$ was created, three table nodes with names **engineerpersonnel**, **otherpersonnel** and **job** were created in the MV-Document, each with interval [T_1, "now"). Similarly, the nodes for their attributes etc., were added in the XML document. In $V_{1.2}$ the schema evolved into the two tables **employee** and **job**; these changes were updated in the MV-Document by changing the end-time of **engineerpersonnel** and **otherpersonnel** to T_2 (as well as the intervals of their attribute and tuple nodes). Meanwhile, a new table node for **employee** is added with interval [T_2, "now"). Since the **job** relation continues in the new version, there is no update on that table node.

To make the storage and querying of MV-Documents more scalable, [9] uses relational databases and mappings between the XML views and the underlying database system. This is facilitated by the use of H-Tables, firstly introduced in [12]. Consider the **employee (id, title, deptno, salary)** relation of schema $V_{1.5}$ in Table 1. Its history is stored in four H-Tables, namely: (i) a key table, **employee_key (id, ts, te),** that stores the interval (ts, te) during which tuple with key id was stored in the corresponding relation. (ii) three attribute history tables: **employee_title (id, title, ts, te), employee_deptno (id, deptno, ts, te)** and **employee_salary (id, salary, ts, te)** that maintain how the individual attributes of a tuple (identified by id) changed over time, and (iii) an entry in the global relation table **relations (relationname, ts, te)** which records the time spans covered by the various relations in the database.

3 Branched Schema Evolution

Many modern complex applications need to support schema branching; examples include scientific databases, collaborative design environment, web-based information systems, etc. With branched schema evolution enabled, a new branch can be created by updating the schema of a *parent* version V_p. If version V_p is a current schema version and the data populating the first schema of the new branch is adapted from the currently alive data of V_p, we have a current branching (*c-branching*). An example of c-branching appears in Fig. 1(b) where the most current version of branch B_1 is $V_{1.5}$. At the current time T_6 branch B_2 is created out of $V_{1.5}$ (i.e., the B_2 creation time is T_6) by applying SMOs on the relations that $V_{1.5}$ has at T_6. For example, under branch B_2 a new attribute *status* was added in **empbio** to describe the marital status of employees. As a result, data can start evolving concurrently under two parallel schemas, $V_{1.5}$ and $V_{2.1}$. A real life scenario leading to c-branching is the case when a company establishes a subsidiary. These two companies share the same historical database (branch B_1 from T_1 to T_6) but in the future their schema and data evolve independently. Note that a version can start from any past version (*h-branching*). In this paper however, we concentrate on c-branching due to the challenges of the parallel evolving it imposes.

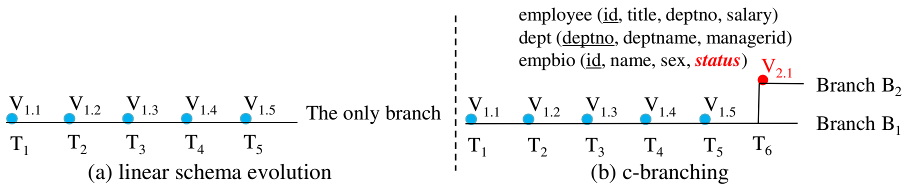

$V_{1.1}$ $V_{1.2}$ $V_{1.3}$ $V_{1.4}$ $V_{1.5}$
The only branch
T_1 T_2 T_3 T_4 T_5
(a) linear schema evolution
employee (id, title, deptno, salary)
dept (deptno, deptname, managerid)
empbio (id, name, sex, *status*)
$V_{2.1}$
Branch B_2
$V_{1.1}$ $V_{1.2}$ $V_{1.3}$ $V_{1.4}$ $V_{1.5}$
Branch B_1
T_1 T_2 T_3 T_4 T_5 T_6
(b) c-branching

Fig. 1. Linear evolution and branching

As more branches occur, effectively the different schema versions create a *Version Tree*; an example (assuming c-branching) with six branches is shown in Fig. 2, which is an extension of the branched employee DB example from Fig. 1(b). Such version tree can easily display the parent-child relationship among versions and branches; this relationship information is very useful for further optimizations.

The novel problems in supporting c-branching are emanated from its sharing of data: the same original data can evolve in parallel under different branches. To provide efficient access and storage in a branched environment, we use different structures to maintain the evolution of schema versions and their underlying data. Since schema changes are much less frequent, we adopt an XML-based model that enables complex querying (BMV-Document). In contrast, the data evolution over time creates large amounts of historical, disk-resident data, so our focus is on branched column tables (BC-Tables) and efficient index methods.

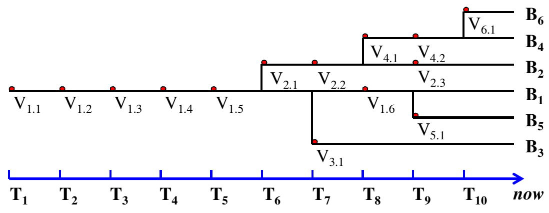


Fig. 2. Example of Version Tree

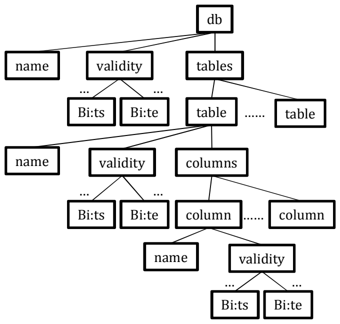


Fig. 3. Illustration of BMV-Document

4 BMV-Document and BC-Tables

4.1 BMV-Document

The BMV-Document is an extension of the MV-Document for storing the branched evolving schema versions in an XML-based representation. The main upgrades are: (i) branch identifier *bid* is needed, because a single timestamp cannot uniquely identify the appropriate schema version. (ii) The BMV-Document refers only to the schema-level storage, and does not detail the data level. (iii) The BMV-Document uses a sharing strategy between versions with various update options and a validity interval (**bid:ts**, **bid:te**) is thus required, as shown in Fig. 3. When a c-branching is created, the child branch may only modify a relatively small part of its parent schema. Simply copying the schemas of all live tables and their columns from the parent version would incur storage overhead.

Schema Sharing. Consider the c-branching on B_1 that creates a new branch B_2 in Fig. 1(b). B_2's creation time is the start time of its first version, namely $V_{2.1}$, which emanated from $V_{1.5}$ by applying some SMOs.

One approach for schema sharing is *full-mark* which adds new (B_2:ts, B_2:te) interval to all corresponding tables and their columns explicitly for the new branch. While this is better than copying all tables and columns, it still requires update work, especially when there are many current tables and columns. To archive better efficiency, we develop a *lazy-mark* approach, which adds a new (B_2:ts, B_2:te) interval to the db node only, and leaves all shared tables and columns unchanged. If the c-branching partially updated the parent schema, besides adding a validity interval on the db node, the lazy-mark approach updates only the modified tables and columns (based on the corresponding table-level and column-level SMOs).

Therefore, the lazy-mark approach can be summarized as: For each update the path to the corresponding level (db, table or column) is visited and the related nodes are updated. Later on, SMOs can update the BMV-Document within a branch as well, and we re-mark those lazy-marked nodes. As a result, the complexity of each schema update for the lazy-mark sharing strategy remains constant per SMO.

Schema Querying. While using schema sharing and lazy-mark to save updating time and storage space, the BMV-Document can still provide efficient access to all branched schema versions. A typical schema query is: "show the schema version at time t for branch B_i". This implies finding the valid tables, as well as their columns, at time t for branch B_i. The procedure of checking whether a table is valid at a given time is shown in Algorithm 1. The interesting case is if table node T does not have a validity interval for B_i; the algorithm should then check whether this table is shared from one of B_i's ancestor branches through lazy marking (line 7-16). For example, consider the case when branch B_2 is created at time T_6 by adding a status attribute in **empbio** table (Fig. 1(b)). Due to lazy-marking, the table **empbio** has only the B_1 branch id in its interval. However, when we check it for branch B_2, following Algorithm 1, we determine that it has been inherited from B_1 and shared by B_2 at time T_6.

Algorithm 1: CheckTable (T, t, B_i)

```
Check whether table node T is valid at time t for branch B_i,
where t is later than B_i's start time.
1    if T has a validity interval for B_i then
2        if B_i:ts = null then return false;
3        else
4            if B_i:ts <= t < B_i:te then return true;
5            else return false;
6    else
7        B_h = B_i's parent; B_g = B_i;
8        while (B_h != null)
9            if T has a validity interval for B_h then
10               if B_h:ts = null then return false;
11               else
12                   tt = B_g's start time;
13                   if B_h:ts<tt<B_h:te then return true;
14                   else return false;
15           B_g = B_h; B_h = B_g's parent;
16       end while
```

4.2 BC-Tables

While the BMV-Document maintains the branched schema versions, the BC-Tables are used to store the underlying evolving data changes. Like H-Table [12], each BC-Table stores the (history of) values for a certain attribute of a base relation.

A BC-Table starts from a particular time and may span over multiple schema versions. However, there are considerable improvements: (i) a BC-Table can be shared by multiple branches; (ii) each data record carries only the start time of its time interval; (iii) suitable branched temporal indexing methods are built on top of BC-Tables.

For indexing a BC-Table we facilitate the branched temporal index ([6, 10]) which is a directed acyclic graph over data and index pages. Data pages (which are at the leaf level) contain temporal data, while index pages contain the searching information to lower level pages. In data pages, due to data sharing, a compact data representation <key, data, ts> is used, where ts corresponds to the record's start time (which will be a bid:time in our BC-Tables) of the original record. In an index page, an entry referencing a child page C is of the form <KR(C), TI(C), address(C)>, where KR is the key-range of the child page, and TI is a list of temporal interval(s) for the shared multiple branches of C.

Splitting occurs when a page becomes full. However, unlike in B+-tree page splitting, when a temporal split happens, the data records currently valid, are copied to a new page. Thus data records are in both the old page and the new page. The motivation for copying valid data from the full page is to make the temporal query efficient. Splits (temporal-split, key-split, and consolidation) cluster data in pages so that when a data page is accessed, a large fraction of its data records will satisfy the query.

Index page splits and consolidations are similar to those of data pages. Since in index page temporal splits, children entries can be copied, this creates multiple parents for these children. As a result, the branched-temporal index is a DAG, not a tree [6].

When the search for a given key *k*, branch B_i and time *t*, is directed to a particular data page P through the index page(s), the algorithm checks all the records in P with key *k*, and finds the record with the largest start time *ts*, such that $ts <= B_i{:}t$.

Nevertheless, page P may have been shared by branch B_i, in which case some of its B_i related entries may not contain the B_i interval. Those entries are inherited from B_i's ancestor branches. Therefore, we need to extend the search algorithm of the branched-temporal index [6,10]. In particular, we extend the meaning of the "<" comparison when comparing bid:time tokens. Given two tokens $B_i{:}T_i$ and $B_j{:}T_j$ the comparison $B_i{:}T_i < B_j{:}T_j$ is satisfied whether ($B_i=B_j \wedge T_i<T_j$) or ($B_i{:}T_i < Par(B_j){:}Ts(B_j)$"), where $Par(B_j)$ is the parent branch of B_j in the version tree, and $Ts(B_j)$ is the start time of B_j.

For example, assume that a data page is shared by branch B_1 and B_2, having entries: <a, v_1, $B_1{:}t_1$>, <b, v_2, $B_1{:}t_2$>, <c, v_3, $B_1{:}t_3$>, <b, v_4, $B_2{:}t_{14}$>, <c, v_5, $B_1{:}t_{15}$>, and let branch B_2 be created from B_1 at time t_{10}. So the valid data entries for B_1 at time t_{15} are <a, v_1, $B_1{:}t_1$>, <b, v_2, $B_1{:}t_2$>, <c, v_5, $B_1{:}t_5$>; while the valid data entries for B_2 at time t_{15} are <a, v_1, $B_1{:}t_1$>, <b, v_4, $B_2{:}t_{14}$>, <c, v_3, $B_1{:}t_3$>.

5 Query Processing

Data queries are temporal queries on the data records (stored in the BC-Tables and indexed by the branched temporal index). As with traditional temporal queries [11], a user may ask for: (i) a *snapshot* query, or (ii) a time *interval* query. In a linear schema evolution, snapshot or interval queries deal with a single branch. In a branched schema evolution, the following *multiple*-branch queries (first introduced in [7]) are also of interest: (i) *vertical* query and (ii) *horizontal* query. We first discuss how to process temporal snapshot and interval queries within one branch, and then proceed to vertical and horizontal queries over multiple branches.

5.1 Queries within a Single Branch

In this case, the temporal constraint (time snapshot or interval) falls within the lifetime of branch B_i. For a snapshot query, the target schema version that stores the queried data is unique and can be identified easily (from the BMV-Document). The corresponding BC-Tables are then accessed through their branched temporal indices.

Processing a time interval query is more complicated because of two challenges: (i) the time interval may have multiple target schema versions (thus even for a single attribute, multiple BC-Tables may be accessed); (ii) in one BC-Table, many data pages may intersect with the time interval, so the search algorithm needs to avoid duplications. The first challenge also appeared in PRIMA [8]: the original temporal query should be reformulated by query rewriting into different sub temporal interval queries for each related BC-Table and the final results are merged from those BC-Tables.

For the second challenge, even in one BC-Table with branched temporal indexing, the naïve depth-first traversal strategy leads to two problems: first, the response set can contain duplicates (due to page splitting copies); second, the same directory entry can be accessed more than once while a query is evaluated. This effect is illustrated in Fig. 4 where the gray-colored rectangles display the pages of the branched temporal index visited for a time-interval query. The naïve algorithm would visit pages 1, 2, 5 once, pages 3, 4, 7 twice, page 8 thrice and page 6 four times.

Traditional duplicate elimination methods such as hashing or sorting may require storage/time overhead, and they are not easy to solve index entry duplication. Therefore, we adopt the $\text{Link}_{\text{based}}$ algorithm proposed in [3] for (linear) multi-version index structures. The BC-Tables' data pages are equipped with external links pointing to their temporal predecessors.

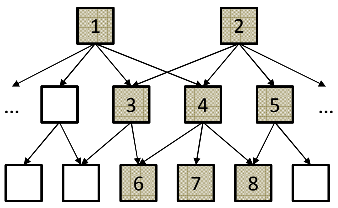


Fig. 4. Visited pages

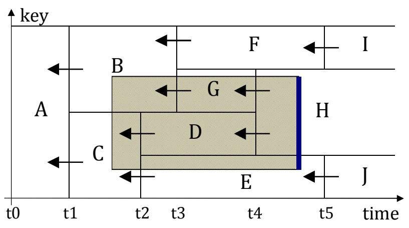


Fig. 5. Data pages with links

An example is presented in Fig. 5 where each page is viewed as the time-key rectangle of the records it contains. A key-range time-interval query (the grey rectangle) intersects pages B, C, D, E, G and H. The $\text{Link}_{\text{based}}$ algorithm consists of two steps.

First, the right border of the query rectangle is used to perform a key-range snapshot query. In Fig 5, this snapshot query will access data pages H and E. Second, for each qualifying page obtained in step 1, its temporal predecessor pages are checked to see whether they contain an answer. If they do, the corresponding pages are put into the buffer, answers are reported and the process is repeated. If the left border of the page is already earlier than the left border of the query rectangle, then we do not proceed further. The worst-case performance of $\text{Link}_{\text{Based}}$ is $O(log_B n + a/B + u/B)$ where B is the page capacity, n is the number of records at right-border time t, a is the number of answers, and u denotes the number of updates in the query time period.

5.2 Data Queries Over Multiple Branches

Vertical Query. The vertical query is an extension of a single branch query, seeking information for a given branch and its ancestors. An example of a vertical query is: "find the data within a key range KR for a given branch B_i and its ancestor branches, at a time stamp t" (or "during a time interval I"). The time stamp t or interval I must be no later than the end time of branch B_i.

For a *vertical snapshot query* of branch B_i and at time t, if t is earlier than the start time of B_i, then the result conceptually lies in one of B_i's ancestors B_j, whose lifetime covers time t. For a *vertical interval query*, the time interval may span multiple branches along a path in the version tree. For example, in Fig. 6, to find titles of employees within a range KR for branch B_4 and its ancestors in a time interval $[T_5, T_{10})$, we need to access data from branches B_4, B_2 and B_1.

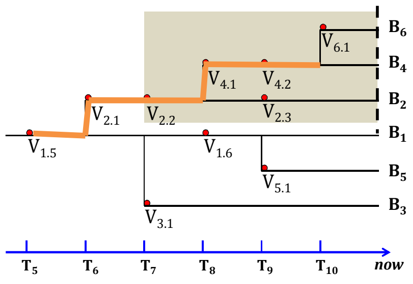


Fig. 6. A part of Version Tree fron Fig. 2.

To process a vertical interval query, we first divide the whole query interval I for branch Bi into multiple smaller adjacent sub-intervals $\{I_1, I_2,..., I_k\}$, one for each ancestor branch along the path $\{B_{i1}, B_{i2},..., B_{ik}\}$ (where $B_{i1} = B_i$, $B_{i2} = B_i$'s parent and so on). In the above example, querying for B_4 with a time interval $I = [T_5, T_{10})$, I should be divided to $[T_8, T_{10})$ for B_4, $[T_6, T_8)$ for B_2 and $[T_5, T_6)$ for B_1 (depicted as the thick orange line in Fig. 6). Then we process the vertical interval query by answering multiple interval queries for each branch and merge the results together. □

However, certain sub-intervals from different branches may be sharing the same BC-Tables, hence a BC-Table could be processed multiple times by different sub-queries. Notice that the sub-intervals are adjacent and the shared data pages are connected by backward links ($Link_{based}$ approach). Therefore, an optimized processing on vertical interval query is to unite the multiple adjacent sub-queries for the same BC-Table into one "super-query". This optimization, called *reunion*, can guarantee that each BC-Table is processed only once for any vertical interval query.

In the above query example, "find the title of employees within a KR for B_4 and its ancestors during $[T_5, T_{10})$", we assume that the **employee_title** table schema is never changed by any branches after it was created at T_5. With the naïve method, we need to process this table three times for three branches with three time intervals as $[B_4:T_8, B_4:T_{10})$, $[B_2:T_6, B_2:T_8)$ and $[B_1:T_5, B_1:T_6)$. When utilizing the optimized method, the three sub-queries are united into one super-query with an interval $[B_1:T_5, B_4:T_{10})$.

Horizontal Query. The Horizontal query accesses temporal information for a given branch and its descendants. An example is: "find data within a key range KR for a given branch B_i and its descendants, at time point t" (or during "a time period I"). The time stamp t or interval I must be no earlier than the start time of branch B_i.

A *horizontal snapshot query* can be visualized as a snapshot of multiple relevant branches from a sub-tree of the version tree. For example, the query: "find data for branch B_2 and its descendants at time *now*", corresponds to the vertical dash line in Fig 6, involving branches B_2, B_4 and B_6. To process a horizontal snapshot query on time t, we first determine which descendants of branch B_i (including itself) are valid at t, and then issue multiple vertical snapshot queries, one for each branch.

A *horizontal interval query* can be visualized as a branch-time rectangle on a sub-tree of the version tree. For example, the query: "find data for branch B_2 and its descendants during time interval $[T_7$, now)", corresponds to the grey rectangle in Fig 6, involving branches B_2, B_4 and B_6. To process a horizontal snapshot query on time t, we again first issue multiple vertical interval queries, one for each descendant branch.

However, this naïve processing method for the horizontal interval query will not be efficient if the multiple vertical interval queries have common parts. In the above example, the vertical interval queries for B_2, B_4 and B_6 during interval $[T_7$, now) have common parts: $[B_2:T_7, B_2:T_8)$ and $[B_4:T_8, B_4:T_{10})$, as depicted in Fig. 6 by the thick orange line inside the grey rectangle.

As a result, for the multiple vertical interval queries, instead of using the same original query time interval I, we should use different intervals for those descendant branches. For each descendant branch B_j, the new query time interval I_j is the intersection of $[ST_j, SE_j)$ with I, where ST_j and SE_j is the start time and end time of branch B_i. For the above example, the optimized vertical interval queries are: $[B_6:T_{10}$, B_6:now), $[B_4:T_8$, B_4:now), and $[B_2:T_7$, B_2:now). This *rearrange* optimization can improve horizontal interval querying by preventing multiple visits of common parts.

6 Merging of Branches

Since branching is allowed for schema evolution, it is quite natural for us to consider the possibility of merging multiple branches. Branching and merging are two key aspects in many modern environments, such as web-based information systems, collaborative framework, and software development managing tools. Branching provides isolation and parallelism, while merging provides subsequent integration. In this section, we consider how to support current version merging (c-merging).

With c-branching, any currently alive version can create a branch; for a c-merging, the currently alive version of branch B_i can merge to another currently alive version from a different branch B_j by creating a new common schema version. In the example shown in Fig. 7, both branching and merging are applied. Such schema evolution will form a *Version Graph* instead of a version tree.

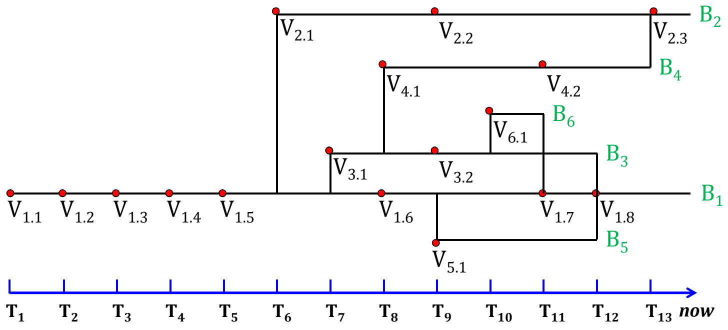


Fig. 7. Schema evolution with branching and merging

6.1 Merging in BMV-Documents

When branch B_i's latest version $B_{i.x}$ merges to branch B_j's latest version $B_{j.y}$ at time t, the branch B_i and version $B_{i.x}$ should be ended and a new version $B_{j.y+1}$ should be created for branch B_j. The branch and version termination can be achieved by updating the end time for corresponding nodes and the lazy-mark process can be utilized for only updating the db and table nodes without reaching to column nodes. After figuring out which elements are discarded from $B_{j.y}$ to $B_{j.y+1}$, and which are added from $B_{i.x}$ to $B_{j.y+1}$, we apply the updates for the corresponding tables and columns. Suitable schema duplication elimination and conflict resolution are applied.

6.2 Merging in BC-Tables

When merging is applied in BC-Tables at the data level records, we still can use the same sharing strategy with the branched temporal index but with special extensions. Assume branch B_i merges to B_j at time t. For both branches, some data records have remained while others are removed (especially when there are conflicts). In BC-Tables, we only delete the removed records by adding null values and keep the remained records unchanged, which is consistent with our sharing method in section 5. Data duplication elimination and conflict resolution are applied as well.

For data accessing, certain extensions should be implemented for merging, since merging integrates data records from two branches into one. Exploring of a branch's ancestors due to lazy mark is extended from one single path to multiple paths with depth-first or breath-first search along the version graph. Meanwhile, the branched temporal indexing can be adapted for merging with certain modifications.

6.3 Query Processing

Here we concentrate on data querying within multiple branches. For vertical queries seeking temporal information for a given branch and its ancestor branches, the ancestors include not only the ones formed by branching but also those by merging. So even for a snapshot querying, the vertical query may need to traverse multiple paths along the version graph by DFS or BFS. For example, assume that in the example of Fig. 8, we want to find some records for branch B_1 and its ancestors at time T_{10}. Traversing the version graph backward for B_1 from *now* to T_{10}, we meet two merging points at time T_{12} and T_{11}. Hence the final result unites the response records from not only branch B_1 but also B_5, B_6 and B_3 at time T_{10}.

To process a vertical interval query we access data from multiple parallel paths which may have common parts. The *rearrange* optimization proposed for horizontal querying under branching can be used here. For example, as shown in Fig. 7, assume we want to find some data for branch B_1 and its ancestors during time interval [T_9, T_{13}). From the version graph, we know that B_3 and B_5 merged to B_1 at time T_{12} and B_6 merged to B_1 at time T_{11}. We can avoid visiting the common paths [B_1:T_{12}, B_1:T_{13}) four times and [B_1:T_{11}, B_1:T_{12}) twice by utilizing *rearrange* to make querying intervals as [B_1:T_9, B_1:T_{13}), [B_3:T_9, B_3:T_{12}), [B5:T_9, B_5:T_{12}), and [B_6:T_9, B_6:T_{11}).

7 Experimental Evaluation

To illustrate the efficiency of our framework we present several experiments based on the running example of the employee DB in Fig 2. First, we extend it with more schema versions and branches. The first ten schema changing points (from T_1 to T_{10}) are shown in Fig 2. After that, we make another ten schema changing points (from T_{11} to T_{20}) in two rounds. In each round, there are five schema changes: the first two are linear schema evolutions followed by one schema version branching and two linear schema evolutions. For each linear schema evolution, we choose 50% of the existing branches and make new schema versions for them updating 20% tables and 20% columns in those tables. For each schema branching, we chose all existing branches and make a new branch for each by updating 20% tables and columns. In the end, we have 20 schema changing points with 24 branches of 104 schema versions.

In addition to linear and branched schema evolution, we also create content-level data changes. From T_1 to T20, after each schema changing point, we update the record-level data value 500 times. For each time, we update all existing branches, and for each branch we update 0.2% of all employees for salary, title, and some other randomly chosen attributes. In the end, we have 10,000 time instants of content-level data updates. The Employee DB schema is initialized with 1,000 tables and average 5 columns in each table. We also produce 10,000 employees with 100 titles and other relevant information. For both schema changes and data changes, the tables, attributes and tuples are chosen randomly with a uniform distribution. The page size of our system is 4KB and we set the data page capacity as B = 100 records.

7.1 BMV-Documents

The sharing strategies among multiple branches and the lazy-mark approach are advantageous in space saving for the BMV-Document without sacrificing querying efficiency. We store the branched schema versions, in XML-based BMV-Documents with three different options when branching occurs: (i) copy the schema without any sharing (Non-Shared); (ii) use the sharing strategy and full-mark approach (Shared); (iii) use the sharing strategy and lazy-mark approach (Lazy-mark). Fig. 8 depicts the size per branch (total size / number of versions) of the documents under certain schema changing points: T_{10} (6 branches), T_{15} (12 branches), and T_{20} (24 branches). The options using sharing strategies use much less space than the non-shared option. Compared to the full-mark, the lazy-mark approach is more efficient.

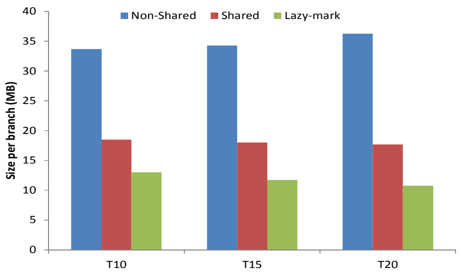


Fig. 8. Space saving in BMV-Documents

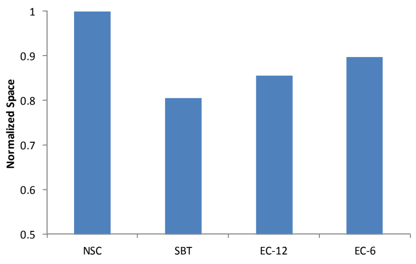


Fig. 9. Space saving in employee_title table

7.2 BC-Tables

Space Saving. In addition to the shared BC-Tables (SBT), we use a non-shared method which simply copies alive records from the parent branch when a c-branching happens. The non-shared copying method (NSC) utilizes the MVBT ([2]) to store data in each branch separately, so that each single branch has its own data pages and index structure. The total sizes of data pages and index pages for all tables of all 24 branches are: NSC 71.4 GB and SBT 54.9 GB; clearly, the shared BC-Tables provide significant space saving. Nevertheless, the querying performance of the non-shared method will be better than the fully shared BC-Tables since data has been fully materialized at each branch. Therefore, we consider a trade-off between space and querying performance by applying an enforced copying method (EC), which only allows at most p branches that can be shared in one BC-Table. If a shared BC-Table already reaches this number p, then for a later c-branching, we enforce copying (make a new BC-Table for the newly branch) instead of sharing. The fully shared BC-Tables and non-shared method are two extreme situations for this enforced copying (p = 1 corresponds to the non-shared method). In our experiments we implemented an enforced copying method EC with p = 12 (EC-12) and p = 6 (EC-6).

In order to factor out the query reformulating, we choose one particular BC-Table **employee_title**, whose schema never changes from the beginning and is shared by all branches. To compare space usage of the **employee_title** table by the four methods (NSC, SBT, EC-12 and EC-6), we depict a normalized space usage. Since NSC has the largest storage usage (data pages + index pages), the normalized space is computed by (method_i's space) / (NSC's space). As shown in the Fig. 9, the shared **employee_title** BC-Table provides the best space savings followed by EC-12 and EC-6.

Snapshot querying. We use the following query: "find titles of all employees whose ids are within a key range of size 100, for branch B_i at time t" and test on all 24 branches. For each branch, we randomly pick 100 time instants which are in the lifespan of that branch and measure the average snapshot querying time. The average results of all 24 branches are calculated and depicted as normalized page I/O (Fig. 10). The SBT method has the largest I/O usage, so the normalized page I/O is computed by (method_i's I/O) / (SBT's I/O). The non-shared copying method has a better snapshot querying performance because data records are stored separately for each branch. However, considering the space saved, shared BC-Tables are performing relatively well on query time. The trade-off methods (EC) gain better querying performance while controlling the space overhead.

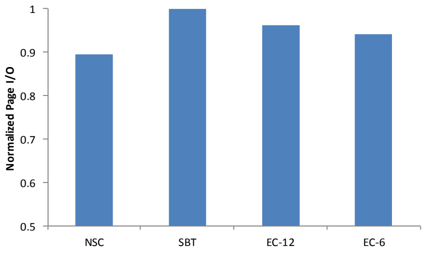


Fig. 10. Snapshot Querying

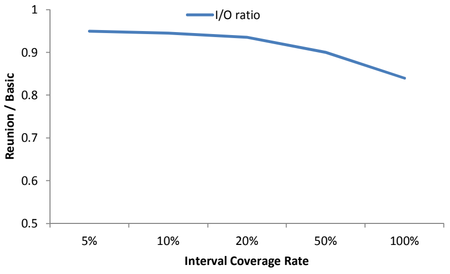


Fig. 11. Vertical interval querying

Interval Query Processing. For interval query processing we implement the $Link_{Based}$ algorithm along with the *reunion* and *rearrange* optimizations in shared BC-Tables. First, we test vertical interval queries involving multiple branches: "find titles of employees whose ids are within a key range of size 100 for branch B_{24} and its ancestors in the time interval I". Five different time intervals are used and their coverage rates with respect to the whole temporal data lifetime are 5%, 10%, 20%, 50%, and 100% correspondingly. Two methods are implemented here: one is the basic solution (Basic) which divides the query interval into multiple sub-intervals for each branch. The other is the optimized *reunion* method (Reunion) that unites the sub-intervals into one super-interval if they are sharing the same BC-Table. The I/O ratio of these two methods (Reunion's I/O) / (Basic's I/O) is shown in Fig. 11. Clearly the *reunion* optimization can improve the vertical interval querying, and the improvements are more significant when the query interval covers more ancestor branches.

Then we consider horizontal interval queries involving multiple branches: "find titles of employees whose ids are within a key range of size 100 for branch B_1 and its descendants in the time interval I". The different interval I coverage rates are used as same as above. We again implement two methods: one is the basic solution (Basic) that issues multiple vertical queries with the same query interval for each descendant branch, and the other is the optimized *rearrange* method (Rearrange) that arranges different query intervals for each descendant branch to achieve querying efficiency. The I/O ratio of these two methods (Reunion's I/O) / (Basic's I/O) is shown in Fig. 12. As seen, the *rearrange* optimization can effectively improve the horizontal interval querying especially when the query interval covers more common parts.

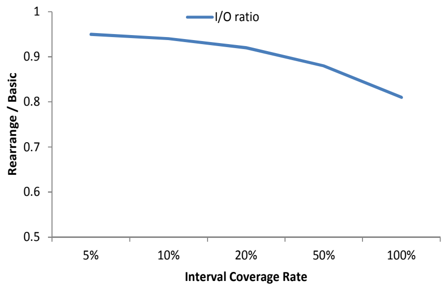


Fig. 12. Horizontal interval querying

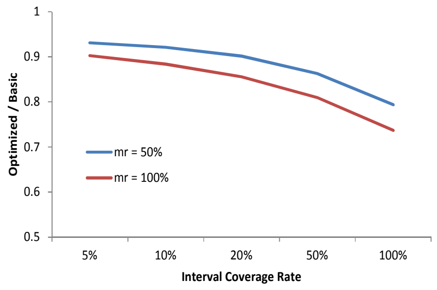


Fig. 13. Querying with merging added

7.3 Branched Schema Evolution with Merging

Finally, we employ schema merging into the branched system as well. The branched schema versions and datasets are extended as follows: We randomly insert 5 schema merging points into the 20 schema changing points, and for each such schema merging point, we randomly pick some existed branches to do the merges. A parameter mr (0 ~ 1) is used to control the merging rate. For example, if mr = 50%, we randomly pick half of existed branches to do the merges. The content-level data changes are generated as before: the data is updated 500 times after each schema changing point (evolving, branching and merging). The total number of time instants with data updates is increased from 10,000 to 12,500.

Below we only show results for the horizontal interval querying for branch B_1. We set up five different querying interval coverage rates as same as above with two different merging rates as mr = 50% and mr = 100%. The methods we test include (i) the basic method (Basic) without avoiding the common sub-paths and (ii) the optimized method (Optimized) with both *reunion* and *rearrange* implemented. The I/O ratio of these two methods (Optimized's I/O) / (Basic's I/O) is shown in Fig 13 for the two mr rates. The optimized method has an advantage in interval querying processing, and this becomes more apparent for larger merging rates and longer query intervals.

8 Conclusion

We addressed branched schema evolution for transaction-time databases. To the best of our knowledge, this is the first attempt to examine both data and schema evolution in a branched environment. Efficient schema sharing strategies with smart lazy-mark updates are used. Schema versions are stored in XML-based documents for ease of querying. Data records are stored in relational column tables with branched and temporal indexing. We also explored temporal querying optimizations, especially for vertical and horizontal interval queries. The feasibility of supporting schema merging was also examined. In future research, we will investigate temporal joins and aggregations under schema evolution with branching and merging.

Acknowledgements. This work is partially supported by NSF grant IIS-0910859.

References

[1] `http://en.wikipedia.org/wiki/History_of_Firefox`

[2] Becker, B., Gschwind, S., Ohler, T., Seeger, B., Widmayer, P.: An asymptotically optimal multiversion b-tree. VLDB Journal (1996)

[3] Bercken, J., Seeger, B.: Query Processing Techniques for Multiversion Access Methods. In: VLDB 1996 (1996)

[4] Curino, C.A., Moon, H.J., Zaniolo, C.: Graceful Database Schema Evolution. In: VLDB (2008)

[5] Curino, C.A., Moon, H.J., Zaniolo, C.: Managing the history of metadata in support for db archiving and schema evolution. In: ECDM (2008)

[6] Jiang, L., Salzberg, B., Lomet, D., Barrena, M.: The BT-Tree: A branched and temporal access method. In: VLDB (2000)

[7] Landau, G.M., Schmidt, J.P., Tsotras, V.J.: Historical Queries along Multiple Lines of Time Evolution. VLDB Journal (1995)

[8] Moon, H.J., Curino, C.A., Deutsch, A., Hou, C.-Y., Zaniolo, C.: Managing and Querying Transaction-time Databases under Schema Evolution. In: VLDB (2008)

[9] Moon, H.J., Curino, C.A., Zaniolo, C.: Scalable Architecture and Query Optimization for Transaction-time DBs with Evolving Schemas. In: SIGMOD (2010)

[10] Salzberg, B., Jiang, L., Lomet, D., Barrena, M., Shan, J., Kanoulas, E.: A Framework for Access Methods for Versioned Data. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K. (eds.) EDBT 2004. LNCS, vol. 2992, pp. 730–747. Springer, Heidelberg (2004)

[11] Tsotras, V.J., Jensen, C.S., Snodgrass, R.T.: An Extensible Notation for Spatiotemporal Index Queries. SIGMOD Record 27(1) (1998)

[12] Wang, F., Zaniolo, C., Zhou, X.: Archis: An xml-based approach to transaction-time temporal database systems. VLDB Journal (2008)