

Seminar Report - Information Systems Seminar FS2010

“Archiving Scientific Data”

Michael Haspra

Seminar: Algorithms for Database Systems, Information Systems Seminar, ETH Zürich, Spring Semester 2010

Paper: Peter Buneman and Sanjeev Khanna and Keishi Tajima and Wang-Chiew Tan. Archiving Scientific Data. Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'02)

Professor: Prof. Donald Kossmann

Sheperd: Corina Bassi

Abstract

This seminar report describes the main ideas of the paper “Archiving scientific data” by P. Buneman et. al. This paper is about a key based hierarchical archiving technique for scientific data which suits this hierarchical structure and supports an according key structure. The archiving approach is explained by an example application. The emphasis

on the descriptions is on understandability. So only concepts needed to understand the key features are explained. For more details please refer to the paper or the other given references.

1 Introduction

Scientific research often makes excessive use of databases. While it is clearly important to have access to the current content of the database, it is also often necessary to be able to retrieve earlier versions of the data. Being able to retrieve earlier states one can reproduce scientific processes and verify the basis of possible findings. Disregarding this functionality in a scientific database would automatically mean to neglect all evidence of the research findings. So a producer of scientific data is bound to take that into account and therefore to maintain such an archive of his data. Peter Buneman et al. propose an archiving technique for hierarchical data with key structure in their paper “Archiving Scientific Data” [1].

Common approaches to archive different versions of data are line based diff techniques. P. Buneman et al. claim that these techniques do not work well for a variety of scientific data, so they suggest using a hierarchical archiving technique instead. The proposed approach is to store the data in a tree structure, enriched with a hierarchical key system for object identification and timestamps, for distinction of different versions of the stored objects. In addition to the description of the storage technique, the authors also present a storage and retrieval algorithm realized in an example application. This example application is used to archive data stored in XML documents. This is an obvious application since XML documents come with a hierarchical structure and in general support a hierarchical key system.

This report is organized as follows. The next section describes some existing archiving approaches and states why there was still a need for the technique presented in [1] by the time it was published. Section 3 describes the archiving technique proposed in [1] by answering the three questions which naturally come up if someone presents an archiving technique. How to store the data? How to save changes of data? How to retrieve data? Section 4 describes some performance aspects followed by further thoughts and a conclusion about the proposed approach.

One note on terminology has to be made here, to prevent confusion. Although this is an archiving approach and not a method to track objects over time, the authors use the term *timestamp* to indicate different versions of an object. The only relation between the timestamps used in [1] and actual timestamps is, that if a timestamp t is smaller to a timestamp t' according to some order relation (the natural numbers in all examples), that implies that t indicates a version earlier (in time) than t' . So, the term *timestamp* here actually means *version*. In particular nothing can be said about the time elapsed between two versions. For consistency with [1] the term *timestamp* is used in the same manner in this report.

2 Related Work

The authors mention different archiving techniques for various data, such as CVS [2] or SCCS [3], as related work. Most of them are so called diff based techniques. In a diff based technique one stores one or several complete versions of the data together with so called edit scripts, these edit scripts describe the differences between one of the complete versions and a particular version. There are several variations of this technique such as incremental diff or cumulative diff which then again can be implemented with a backward or a forward diff strategy. The details of these techniques are not of interest. Important are two problems they all have in common.

The first problem is a performance issue. To retrieve a particular version of some data such a diff based archiving tool first has to find a complete version of it (this can be an earlier or later version than the requested

one). Then it has to modify this complete version, using the edit scripts, to finally get the requested version. As a document goes through many versions, and therefore many edit scripts are stored, it can be quite costly to do this reconstruction.

The second problem is a semantic one. When only edits to the data are stored one may violate some notion of “object” continuity. This issue can be explained by the following example. Suppose you have a database to store persons as `Person(name, address)`. If two persons now exchange their addresses, a diff algorithm, which has to store this change in the archive, may come to the conclusion that these two persons actually exchanged their names. The following example explains this problem. The diff algorithm cannot know if Ann and Bob exchanged their names or if they exchanged their addresses and now appear in a different order.

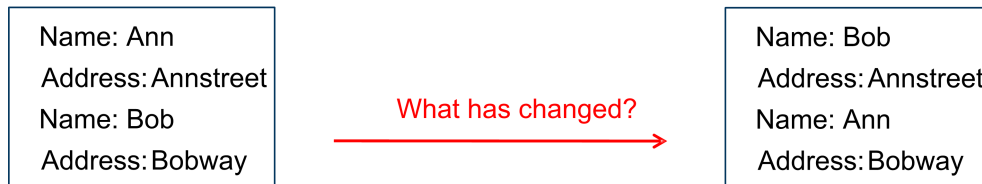


Fig. 1: An example where a diff algorithm may fail.

If one wants to obtain the version history of an object stored in the database, here of a person, this will cause some problems. Clearly there are some applications where the notion of an “object” does matter and the archiving system has to support this, by introducing a temporal invariant key for every object.

3 The proposed archiving technique

3.1 Problem statement

The problem which P. Buneman et al. want to address is the following. One has some hierarchical data that contains different “objects”, which can be identified with a key. These objects can change over time and one wants to store the different versions of these objects in an archive. It must also be possible to retrieve a particular version of such an object (or several objects).

So, one needs an archiving technique that preserves the hierarchical structure of the data and allows the notion of objects and object continuity. It is helpful to keep in mind that the main requirement for the archiving technique is this notion of objects. The technique must also be efficient in terms of space requirement and runtime for saving and retrieving objects. Since scientific databases can be very large with millions of objects.

3.2 Contributions of [1]

In [1] an archiving technique is presented, which addresses the previously given problem statement.

To describe an archiving technique one has to answer three questions:

- How to store the data?
- How to save changes of data?
- How to retrieve data?

The following three sections answer these questions. In [1] this is mainly done by describing an example application to archive data in XML format. However, the archiving technique itself does not only apply for XML data. For example one could also think about JSON Objects, but it might be helpful to keep this possible application in mind during the next sections. XML documents are inherently hierarchical and therefore convenient if one wants to comprehend the examples or construct own ones. Also the tree representation which will be used for the examples is a common way to look at XML. Nevertheless, the following descriptions are not meant to be XML specific.

3.2.1 How to store the data?

In order to efficiently store scientific data, the authors first identify some properties which hold for a variety of scientific data. First they indicate that most scientific data is stored in some kind of a hierarchical database. Second they claim that most databases have a hierarchical key structure. That means each node in the hierarchy can be uniquely identified by a hierarchical key. Hierarchical means that the key holds some information about the position of the node in the hierarchical structure (maybe together with some properties of the node itself).

Because of these findings about the hierarchical nature of a variety of scientific data the authors propose to build a hierarchical archive based on two main ideas.

First, all versions of the database are merged into one archive based on the keys of the objects occurring in the different versions. A timestamp will then describe the occurrences of an object in different versions.

Second, despite a timestamp exists for every node in the conceptual view, in practice it is only stored if it differs from its parent. This idea saves space.

To give an example, assume one wants to again have a database of persons, the different versions of this database should be archived, so one can keep track of the changes. Figure 2 illustrates this example, but now in a hierarchical view (remember this could be XML documents). Again Ann and Bob exchanged their address between version 1 and version 2.

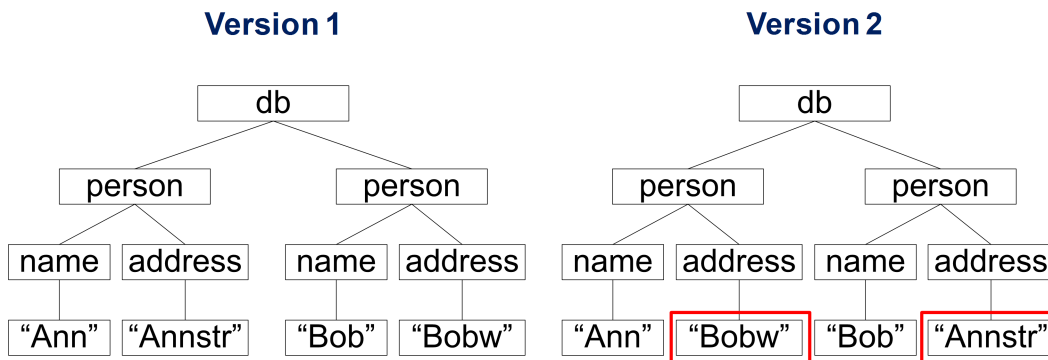


Fig. 2: Two versions of a persons database.

Figure 3 shows how the data can be stored in a single archive (by merging the two versions together). A node which occurs in different versions is only stored once in this archive. The timestamp (given by the attribute 't') indicates in which versions this particular node is valid. So for example if one is interested in the left person sub tree, at version 1, the child node of address would be "Annstr" and at version 2 it would be "Bobw". As already described, nodes which have no timestamp inherit the timestamp from their parent.

How this merging of two versions is done will be explained in section 3.2.2

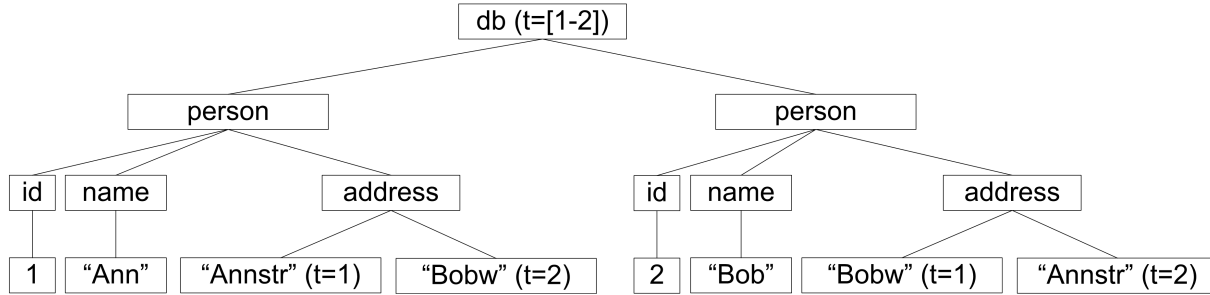


Fig. 3: Both versions of the database in a single archive.

The proposed technique now changes the way one stores information about changes of the data. In the diff based approach the information of changes is grouped by time, where in the proposed approach this information is grouped by objects. That immediately brings up an important precondition for the data one wants to archive. The data has to have a key system to make this approach work. Otherwise one cannot group information about change by object, since one cannot identify such an object.

Defining keys for hierarchical data is straight forward and very common nowadays. For the example of XML one could do it like in [4] or in XML Schema. The key system used in [1] is actually the one defined earlier by P. Buneman et al. in [4]. The details about the key system are not important for this report but some explanations have to be made here, to prepare for the description of the inserting and retrieving algorithms.

1. A key in [4] always consists of two parts (remember that the conceptual view is a tree). A path Q , beginning at the root node r and ending at some descendant d . A set of paths P beginning at d and ending in some descendant of d . More precise, Q defines the location of an object in the archive (a sub tree) and P defines the descendants of d which identify this object (this sub tree). As an example, for the archive given in figure 3, a key could be $(/db/person, \{id\})$ where $Q = db/person$ and $P = \{id\}$. That means: Every person can be uniquely identified by its id.
2. What keys do exist is application specific. That means, that this has to be defined by the user and has to be provided as input to the archiving system. For every sub tree in the archive one wants to look at as an object one has to specify a key. In the example, if the persons are objects one has to specify a key for them (like in the previous section). This key specification does not necessarily have to be made manually, in the case of XML one could also generate it from a Schema specification.
3. In later descriptions, a node for which a key has been defined will be called **keyed**. And a keyed node which has no keyed descendants will be called a **frontier** node. It should be clear that the keyed nodes and their sub trees represent the actual objects.

3.2.2 How to save changes of data?

To describe the storage mechanism it is probably good to first give an overview about what should happen in general during such a “save” operation and what components of the archiving technique play a role during

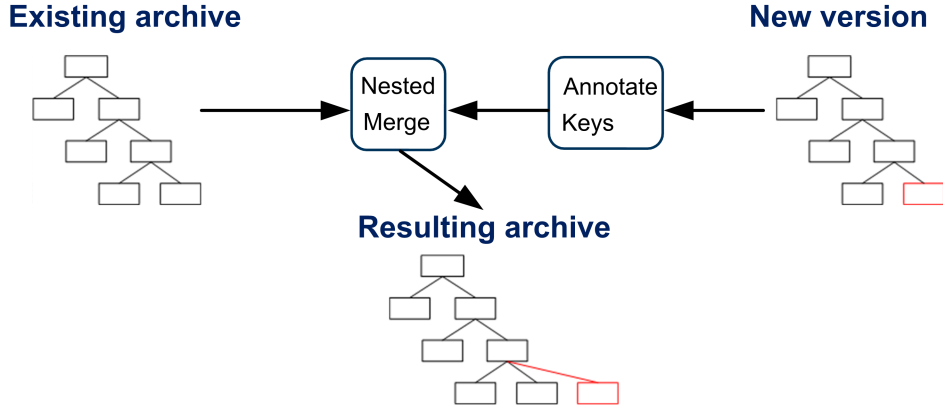


Fig. 4: The storage process.

this operation. The basic situation when a save operation is applied is that there exists a current version of a database, an archive containing previous versions of this database and a key specification, which is provided to the system (this has already been described in the previous section). The result of the “save” operation is then an archive containing all states of the database including the current one. So the purpose of this “save” operation is not to save changes of the objects into the database (this is left to the actual implementation), it is to save the current state of the database into the archive. So it’s better to think of it as a merging of two big trees. This general situation is illustrated in figure 4.

The two algorithms which are involved are *annotate keys* and *nested merge*. *Annotate keys* is a preprocessing step, that has to be applied only to the current version and *nested merge* is then applied to both, to do the actual merging. It is important to see that these algorithms are always applied to the whole archive and the whole current version. Even if only one object has changed between two versions, the whole tree structure is processed during the merging process.

Of course this hasn’t to be true for every change in the current version. Because changes in the current version and saving these changes into the archive are two different things. When to apply the merging is completely left to the user. It could be that many objects in the current version are being changed before it is merged into the archive. But if the same object changes twice and one wants to keep track of this, one also has to do the merging twice.

Annotate keys *Annotate keys* simply annotates every keyed node (remember this is a node for which a key in the key specification exists) in the tree structure with its key value. This has only to be done for the current version because for every version in the archive it has already been done during inserting. The way *annotate keys* works is pretty straight forward. It just scans the tree in preorder and collects all relevant descendants (specified in the key spec) for every keyed node. When the preorder scan leaves a keyed node the collected key is added to it.

As an example consider the current version to be that one given in figure 3. Also assume that there exists a key specification which contains the key $(/db/person, \{id\})$. As already states this means that every person can be uniquely identified by its id. So *annotate key* collects the value of the *id* node for each *person* node and annotates this to the *person* node itself. The result of this is given in figure 5.

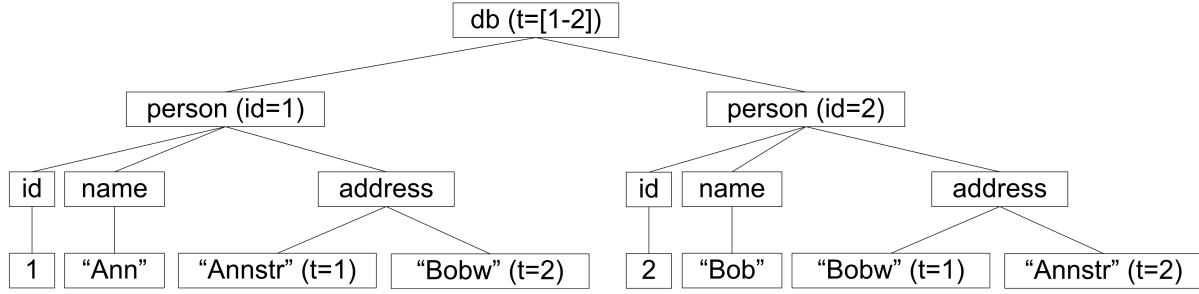


Fig. 5: The annotated archive.

The reason why this is done is that the following merging algorithm now doesn't have to collect key values during the merging process.

Nested merge The second algorithm which takes part in the merging process is *nested merge*. The two main ideas of *nested merge* are, first to identify elements or nodes that correspond to each other in the archive and the current version through keys and then to merge these together. The algorithm is given as pseudo code in listing 1. The pseudo code assumes for simplicity that the archive is not empty and that no element contains an attribute besides the id and the timestamps. The following list describes the used variables and after that the algorithm is described and the important steps are explained with an example.

- x denotes a node in the archive
- y denotes a node in current version to add
- $t(x)$ denotes the timestamps annotated on node x
- $l(x)$ denotes the label and key value of node x
- $v(x)$ denotes the sub tree rooted at x
- $v^-(x)$ denotes $v(x)$ without x
- i denotes the version of the database which is currently merged with the archive (So the archive contains nodes with versions in $\{1, \dots, i-1\}$)

Algorithm 1 The merging algorithm: Nested Merge

```

Nested_Merge( $x, y, T$ )
    if  $t(x)$  exists then //We're looking at a node that has a timestamp that timestamp has to be extended.
        add  $i$  to  $t(x)$  and let  $T$  be  $t(x)$ 
    Let  $X$  and  $Y$  denote the set of all children nodes of  $x$  and  $y$  respectively
    if  $y$  is a frontier node then //if we're looking at a frontier node the recursion stops because no descendant is keyed.
        if every node in  $X$  is not a timestamp node then
            if  $v^-(x)$  and  $v^-(y)$  are different then
                add  $\langle T \text{ } t=i \text{ } T - \{i\} \text{ } v^-(x) \text{ } \rangle / T \text{ } \rangle$ 
                and  $\langle T \text{ } t=i \text{ } v^-(y) \text{ } \rangle / T \text{ } \rangle$  as subtrees of  $x$ 
            else
                if there exists a node  $x'$  in  $X$  such that  $v^-(x)$  and  $v^-(y)$  are the same then
                    add  $i$  to  $t(x')$ 
                else
                    add  $\langle T \text{ } t=i \text{ } v(y) \text{ } \rangle / T \text{ } \rangle$  as subtree of  $x$ 
        else
            Let  $XY = \{(x', y') \mid x' \in X, y' \in Y, l(x') \text{ and } l(y') \text{ are the same}\}$ 
            Let  $X' = \{x \in X \mid (\forall y \in Y) (x, y) \notin XY\}$ 
            Let  $Y' = \{y \in Y \mid (\forall x \in X) (x, y) \notin XY\}$ 
            for every pair  $(x', y') \in XY$ 
                Nested_Merge( $x', y', T$ )
            for every  $x' \in X'$ 
                if  $t(x')$  does not exist then let  $t(x')$  be  $T - \{i\}$ 
            for every  $y' \in Y'$ 
                Let  $t(y') = \{i\}$  and place  $v(y')$  as child node of  $x$ 

```

To illustrate the descriptions consider the example in figure 6. It shows a very simple archive with only one person which has three values. Also assume that *person* is a keyed node and *address* is a frontier node. In a real scenario these two nodes would be annotated with their key values. This is omitted because of simplicity.

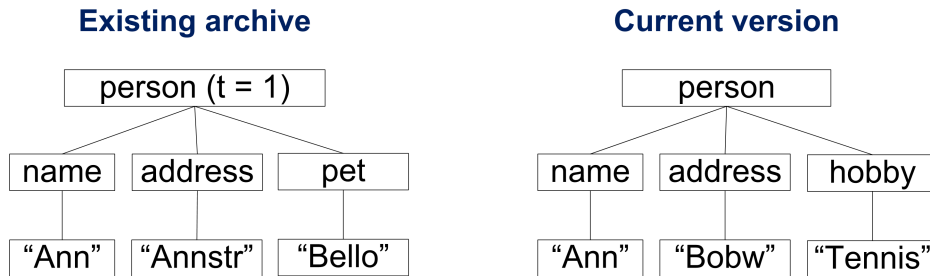


Fig. 6: An archive and the current version to be merged.

The algorithm is started with $\text{Nested_Merge}(r_A, r_D, \{\})$ where r_A is the root node of the archive and r_D is the root node of the current version. In the example the first call to *nested merge* would be: $\text{Nested_Merge}(\text{person}, \text{person}, \{\})$, so x is set to *person* (that one from the archive) and y is set to *person* (that one from the current version). Note that it is a property of the algorithm that $l(x)$ and $l(y)$ are always the same when it is applied to two nodes.

First it's checked if the node in the archive has a timestamp attribute. If this is the case it is extended by the current version i (line 2 & 3). If it has no timestamp, they are inherited from the parent which already has been processed. In the example x has a timestamp, so the timestamp for the node *person* it is extended to $(t = [1 - 2])$.

Then the children of both nodes x and y are collected (line 4). In the example these sets are: $X = \{name, address, per\}$, $Y = \{name, address, hobby\}$.

Now it is checked whether y is a frontier node or not (line 5). In the example this is not the case (because *person* has a keyed descendant), so the else branch has to be executed.

There the children of x and y are partitioned in three sets (lines 16-18):

- XY contains all node pairs from the archive (children of x) and the new version (children of y) that correspond to one another. That means they have the same key and label value.
- X' contains all children of x where there exists no according node in the set of children of y . That means that these nodes are no longer there in the new version.
- Y' contains all children of y where there exists no according node in the set of children of x . That means that these nodes are new.

For the example this would lead to the following three sets:

- $XY = \{name, name, address, address\}$
- $X' = \{pet\}$
- $Y' = \{hobby\}$

For all nodes in XY with the same label *nested merge* is now applied again. For *name*, this is not interesting because `Nested_Merge(name,name,[1-2])` does nothing. For *address* the following now happens.

It has no timestamp so the sets X and Y are constructed immediately to $X = \{Annstr\}$ and $Y = \{Bobw\}$. Now, because y is now a frontier node x and y are merged together (lines 6-9). After that the recursion ends because the only frontier node has been processed.

Finally, what happens to deleted and new nodes: All nodes in X' and Y' will now have their timestamps updated and those in Y' are inserted as children of x . In the example the node *pet* gets a timestamp ($t = 1$) and the node *hobby* is inserted with the timestamp ($t = 2$).

The resulting archive is showed in figure 7.

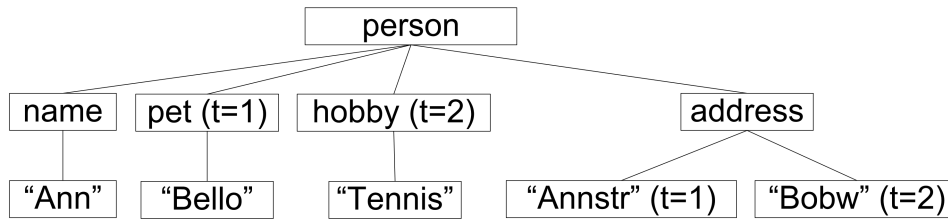


Fig. 7: The resulting archive after nested merge has been applied.

3.2.3 How to retrieve data?

Finally, the question “How to retrieve data?” is left. Just scanning the whole archive for retrieving a version would be inefficient. So the authors suggest building auxiliary data structures to direct the search for relevant nodes of a version in the archive. This auxiliary data structures are so called *timestamp trees*. They are built for every non-timestamp node in the archive.

Timestamp trees

Structure: A timestamp tree for a node x with k children is a binary tree with k leaves (one for every child node). In each leaf, the timestamps and a pointer to the corresponding node in the archive are stored. Internal nodes contain the union of the timestamps of its children.

Construction: To construct a timestamp tree for a node x one simply collects all children of x along with their timestamps and the pointers into the archive file (Those could simply be paths). The binary tree is then build bottom-up wise and stored to disk. To make use of the tree during retrieval, the location of this tree is written into x . This construction has to be done each time a new version of the database is archived.

Figure 8 shows such a *timestamp tree* for the some node db including all pointers.

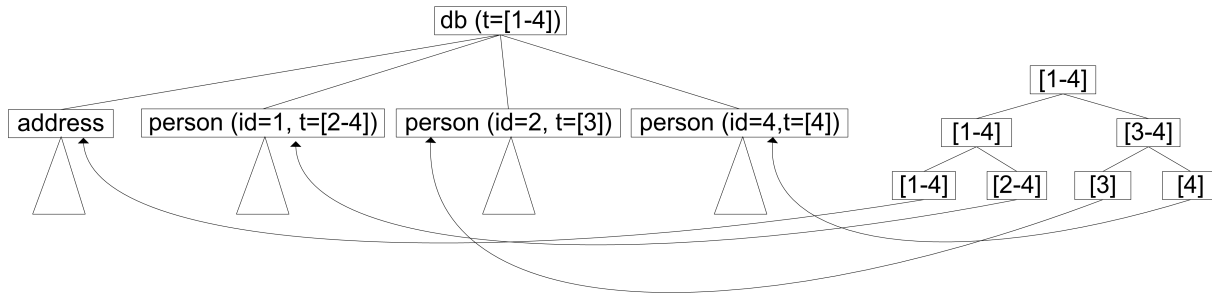


Fig. 8: Timestamp tree for some node db

Retrieval of a version i To get the version i of a node x one simply has to scan its timestamp tree. During this scan one collects all pointers that correspond to the timestamp i and then simply retrieves all nodes in the archive according to the collected pointers.

Retrieving the temporal history of an element In [1] there is also a scheme sketched to retrieve the temporal history of an element. That means, given the key of an element, one would like to retrieve the times at which this element exists. The approach is pretty straight forward. For every node in the archive a list containing all its children is saved. This list contains the timestamps and a pointer to the child nodes and is sorted by timestamp. To now retrieve the temporal history of an element, an algorithm would go down the path, given by the key of the requested element, and performs a binary search in every children list.

4 Performance

4.1 Space requirements

The authors claim that their approach does not need much more space as the mentioned diff based techniques. To support this claim they provide some performance tests and the results show that this is in fact true for this test. Unfortunately it is not indicated if the tested archive is measured together with the auxiliary timestamp trees. So the result is a bit questionable. Also Nested Merge needs an extra hashtable to be fast, so the actual space requirements are probably larger.

4.2 Running time

4.2.1 Nested Merge

The authors claim that nested merge runs in $O(N \log N)$ where $N = \max(N_A, N_D)$ and N_A is the number of nodes in the archive and N_D the number of nodes in the new version. A proof for that is given in [1], but it is not discussed what to do if the sets do not fit in main memory. Also they do not consider the time it needs to recompute the timestamp trees. A running time of $O(N \log N)$ is not bad if one thinks about merging a new version, with a lot of changes, into an existing archive. But if one considers that this can also be the case if only one single node changed, this is pretty bad.

4.2.2 Retrieving the temporal history of an element

It is easy to see, that this algorithm needs $O(l \log d)$ time where l is the length of the path in the key and d is the maximum degree of the archive. If the key has length l , at l positions, during the search in the archive, a list of maximum length d has to be searched. This can be done with a binary search. So the running time is l times a binary search in a list of maximum length of d , this leads to: $O(l \log d)$.

5 Further thoughts

The presented approach seems to be convincing and may be a good choice for many data archives. Especially for XML data. But there are some flaws and further thoughts which are worth being mentioned:

5.1 Requirements to the data

The presented approach only works for data that has a hierarchical structure. It also has to support a key system which is not always the case. Also if one in facts wants a document oriented approach one had to model this somehow in the data since everything is stored in one big tree.

5.2 Storage

The presented approach does not account for any changes other than adding and removing an object. In particular one cannot save changes about relationships between nodes or consider document order. The paper does not discuss what happens when the data does not fit in main memory. From a today's point of view this is clearly a flaw.

5.3 Nested Merge

Nested merge has two main flaws. First it is always applied to the whole archive and a whole new version. There is no possibility to just add a single object. Also the timestamp trees have to be recomputed every time nested merge is applied. The authors themselves mention that such a scientific data archive has adding of new data as the most common operation. If one wants to immediately store this change in the archive every time a new object is saved every saved node is processed every time. Second to identify if a child of a node is present in the archive and the document nested merge may have to do multiple passes over the same data. This could be solved with sorting. The authors account that in [3] 6 years later by introducing a sorted list of children for each node.

5.4 XArch

The presented approach was realized in an actual application 6 years later by Heiko Müller, Peter Bunemann and Ioannis Koltsidas. It is an application to archive XML documents in the presented way. They also made two improvements. First, the already mentioned sorting of child nodes to support nested merge and second a query language. The development of this project still goes on. It can be downloaded at [7].

6 Conclusion

The methods published in the paper have many flaws and do not serve their purpose very well, from a today's point of view. But nevertheless it has its eligibility. By the time the paper was published there haven't been many methods for archiving hierarchical data. So it should be seen as some kind of introduction which says, "Hey there is a lot of data for which we do not have suitable archiving techniques" and then gives some ideas about how to do it. In fact since 2002 many others tried to do improve that and build their own methods building on the ideas of [1]. One of these examples is a paper about Temporal XML [8] which, among other things, addresses the problems of missing reverences and the fact that the merging algorithm is always applied to the whole archive.

7 References

- [1] Peter Buneman and Sanjeev Khanna and Keishi Tajima and Wang-Chiew Tan. Archiving Scientific Data. Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'02)
- [2] Concurrent Version Systems. Unix man pages - cvs
- [3] Source Code Control System. Unix man pages - sccs.
- [4] P. Buneman and S. Davidson and W. Fan and C. Hara and W. Tan. Keys for XML. *In Proc. of Int'l World Wide Web Conf., 2001*
- [5] P. Buneman, S. Khanna, K. Tajima, and W. Tan. Archiving Scientific Data. Technical report, University of Pennsylvania, 2002.
- [6] Heiko Müller, Peter Buneman, Ioannis Koltsidas, XArch: Archiving Scientific and Reference Data, SIGMOD'08, 2008
- [7] <http://xarch.sourceforge.net/>
- [8] Flavio Rizzolo and Alejandro A. Vaisman Temporal XML: modeling, indexing, and query processing, The VLDB Journal (2008).