Archiving Scientific Data

Peter Buneman[*] Sanjeev Khanna[†] Keishi Tajima[‡] Wang-Chiew Tan[§]

ABSTRACT

We present an archiving technique for hierarchical data with key structure. Our approach is based on the notion of timestamps whereby an element appearing in multiple versions of the database is stored only once along with a compact description of versions in which it appears. The basic idea of timestamping was discovered by Driscoll *et. al.* in the context of persistent data structures where one wishes to track the sequences of changes made to a data structure. We extend this idea to develop an archiving tool for XML data that is capable of providing meaningful change descriptions and can also efficiently support a variety of basic functions concerning the evolution of data such as retrieval of any specific version from the archive and querying the temporal history of any element. This is in contrast to diff-based approaches where such operations may require undoing a large number of changes or significant reasoning with the deltas. Surprisingly, our archiving technique does not incur any significant space overhead when contrasted with other approaches. Our experimental results support this and also show that the compacted archive file interacts well with other compression techniques. Finally, another useful property of our approach is that the resulting archive is also in XML and hence can directly leverage existing XML tools.

[*]University of Edinburgh and University of Pennsylvania. Supported by NSF IIS 99-77408 and NSF DL-2 IIS 98-17444. Currently at University of Edinburgh. Email:peter@cis.upenn.edu

[†]University of Pennsylvania. Supported by Alfred P. Sloan Research Fellowship and by an NSF Career Award CCR-0093117. Email:sanjeev@cis.upenn.edu

[‡]Japan Advanced Institute of Science and Technology. Part of this work was done while the author was visiting University of Pennsylvania. Email:tajima@jaist.ac.jp

[§]University of Pennsylvania. Supported by NSF IIS 99-77408 and NSF DL-2 IIS 98-17444. Email:wctan@saul.cis.upenn.edu

ACM SIGMOD '2002 June 4-6, Madison, Wisconsin, USA

1. INTRODUCTION

Scientific databases exist to disseminate the latest research in some area. However, since other research is based on the content of the databases, it is also important to create archives containing all previous states of the data. Failure to do this means that scientific evidence may be lost and the basis of findings cannot be verified. The onus of keeping archives typically falls on the producers of the data, but there appear to be no general techniques for keeping long-term archives or for efficient retrieval from those archives. As an example of the issues involved, consider two widely used data in genetic research: SWISS-PROT [1], a protein sequence database, and On-line Mendielian Inheritance on Man [14] (OMIM), a database of descriptions of human genes and genetic disorders. Both databases have a similar hierarchical structure and both are heavily *curated*, i.e. they are maintained with extensive manual input from experts in the field. In the case of SWISS-PROT, a new version is produced approximately every four months, and all old versions are archived. In the case of OMIM, a new version is produced every day or more often, but only occasionally is a (printed) archive produced. In OMIM, not enough editing information is maintained to reconstruct the exact state of the database at an arbitrary time in its history. These examples illustrate the obvious trade-off between the frequency with which the database is "published" and the space required for complete archiving. Even if the space issue is not critical, there is the issue of the efficiency with which one can query the temporal history of some part of the database. For example, in looking at the history of a component of one of these genomic databases, one might well want to know when a given observation first appeared or when some attribute last changed. SWISS-PROT and OMIM are two contrasting examples of archiving practices. A search of scientific data available on the Web (e.g. [3]) shows that archiving is a ubiquitous problem. Even databases of physical constants [16] are less "constant" than one might naively imagine.

A popular approach to keep all versions of data is to use diff-based technique [24, 5, 10, 19, 20, 6]. A sequence of edit scripts is stored so that one can roll back to any version. There are two problems with this approach. First, as a document goes through many versions, it becomes increasingly costly to recover an old version by undoing the sequence of edit scripts. The second issue is semantic: the minimal edit may violate some notion of "object" continuity. As an example, suppose that the versions of the database are instances of Person(Name, DateOfBirth, Address, Zip). If two individuals exchange houses, a diff algorithm might explain the change as two individuals changing their names and dates of birth. This does not matter if we are only interested in obtaining instances of the entire database, but it does matter if we want to recover the temporal history of components of the database. This example indicates that there is a temporal invariance of keys that should be captured by an archiving system in order to arrive at meaningful change descriptions of "objects". In the archiving technique we are going to describe shortly, we are able to match "objects" across versions before computing the difference. Hence for the above example, we are able to detect that each individual has changed his address.

We are going to describe an archiving technique that works well for a variety of scientific data. Many such databases are kept in hierarchical data formats and they typically have two other properties that we shall capitalize on. First they are *accretive*. Most changes are addition of data. Existing data may be modified or deleted, but such changes are relatively infrequent. Second, they have a hierarchical *key structure*. There is a constraint on the data that allows each node in the hierarchy to be uniquely identified by the path in which it occurs and the values of some of its subelements. This is analogous to the key systems in relational databases where every tuple in a database can be uniquely identified by the name of the relation it belongs and the values of its key attributes. We have found that well-organized scientific data and various domain-specific hierarchical data formats usually support such a structure. Our archiver leverages these properties and effectively stores multiple versions of hierarchical data in a *compacted archive* using the following techniques:

- **Merging versions based on keys.** In contrast to the diff approach which stores edit scripts, we merge all the versions into one hierarchy. An element may appear in many versions, but we identify those occurrences of the same element based on keys, and store it only once in the merged hierarchy. A timestamp describing the sequence of versions in which an element appears are stored with that element. Since changes to our database are largely accretive and an element is likely to exist for a long time, we can compactly represent its timestamps using *time intervals* rather than a sequence of version numbers.

- **Inheritance of timestamps.** Conceptually, a timestamp is stored with every element to indicate a set of version in which it existed. In reality, a timestamp is stored at a child element only when it is different from the timestamp of its parent element.

Example. A simple example illustrates how the archiver works. Figure 1 shows a sequence of versions of a company database. Each version of the database consists of information about its employees and the company address. Every employee can be uniquely identified by his employee id, i.e. id is the key for employees. Each employee also has one name, one salary value, and optionally one telephone number. In addition, a version number is assigned to each version (in this example, 1, 2, 3), and the root node of each version is annotated with a timestamp including that version number.

Figure 2 shows how those versions can be merged into one compact archive by "pushing down" timestamps. We first start at the top-level, and determine the nodes that correspond to one another across all versions according to their key values. (At top-level, each version has only one root node and they correspond to one another.) We merge corresponding nodes together, annotate the resulting node with timestamps of all merged nodes and push the timestamp of each merged node down their respective subtrees. We recursively invoke this procedure for the children of merged nodes until we reach the leaves.

Observe that in the resulting archive in Figure 2, nodes appearing in many versions are stored only once in the archive. If a node occurs in version i, then the timestamp of the corresponding node in the archive contain i. We use time intervals to describe the sequence of versions for which a node exists. For example, the time intervals [1-3,5,7-9] denotes the set 1,2,3,5,7,9. A node that does not have a timestamp is assumed to inherit the timestamp of its parent. For example, the name node under the emp (t=[2-3]) inherits the timestamp t=[2-3].

Observe that it is a property of the archive that the timestamp of a node is always a superset of timestamps of any descendant. This archive can be represented in XML, as shown in Figure 2. For example, employee Joe has a timestamp tag <T t="2-3"> around it, indicating that the entire subtree within exists in versions 2 and 3. Furthermore, during these times, Joe has salary 22k at version 2 and 30k at version 3.

We may assume that the tag T is in a different namespace [26]. It is also easy to see that to reconstruct any previous version, all we need is a simple scan through this document. Notice that information on changes is grouped by elements in this structure while information on changes is grouped by time in the diff approach. Also, our archive ignores the order among elements with keys. If Ann occurs again in version 4 after Bob, it will still be represented in the archive before Bob.

There are two immediate caveats about this approach. First, what happens if the data does not have a key system, i.e. there are nodes that cannot be uniquely identified by their paths and any subelements? We have found that most scientific data sources have a well-organized key system. However, there are also cases in which we cannot define appropriate keys for all the nodes down

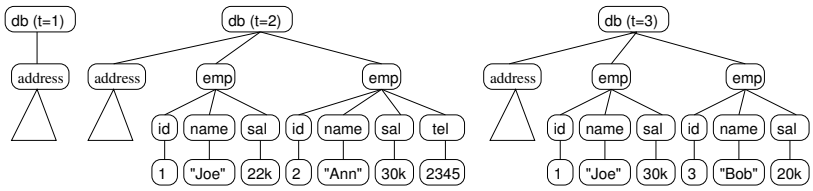


Figure 1: A sequence of versions

```
<T t="1-3">
  <db>
    <address>...</address>
    <T t="2-3">
      <emp><id>1</id>
           <name>Joe</name>
           <sal><T t="2">22k</T><T t="3">30k</T></sal> </emp> </T>
    <T t="2">
      <emp><id>2</id>
           <name>Ann</name>
           <sal>20k</sal>
           <tel>2345</tel> </emp> </T>
    <T t="3">
      <emp><id>3</id>
           <name>Bob</name>
           <sal>25k</sal> </emp> </T>
  </db>
</T>
```

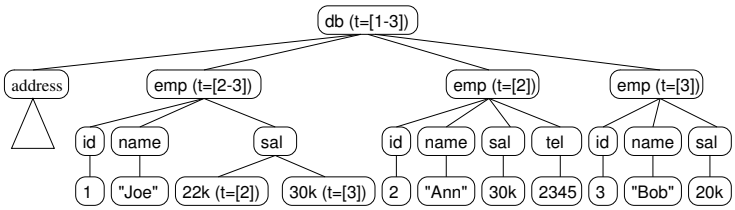


Figure 2: "Pushing" time down. Example of an archive.

to the leaves. For example, some data may be free text represented as a sequence of `<line>` elements and some `<line>` elements may have same text value. Even in such a case, provided the upper nodes in the tree have keys, we can still "push down" the timestamps in the upper part, and stop, or start to use a conventional diff, when we reach elements without keys, such as `<line>` elements in the example above. In fact, if the entire document does not have appropriate keys, our archiving technique is the same as the SCCS approach [21] (see also Section 6). The second caveat is whether the new structure really is smaller than the accumulated past versions. Here we capitalize on the fact that the time sequences associated with elements can be compactly represented as a small number of intervals and are often inherited from a parent element.

Contributions. We extend the technique of Driscoll *et. al.* [9] (See Section 6) and develop an archiving tool for XML data, which compacts a sequence of versions into a single XML document. We show that our approach is viable and comes with several benefits:

- We show that the compacted archive can be constructed efficiently, i.e. a new version can be efficiently merged with an existing archive.
- We show, experimentally, that the space overhead for the compacted archive is comparable to the traditional incremental diff approach.
- We also establish experimentally that the compacted archive works well – in fact better than compressing diffs – with XML compression [12].
- We show that since the structure of our archive is conceptually meaningful, we can easily define index structures on top of the archive to efficiently support various basic operations such as retrieving a past version and finding the temporal history of an element.

For example, on the basis of our experimental findings for OMIM, which is not adequately archived (we recorded 100 versions over 100 days) we predict that we should be able to construct a compacted archive for a year in less than 1.12 times the space of the last version. Moreover the archive, under a XML compression tool, will compress to 40% of the size of the last version.

Organization. This paper is organized as follows. The next section describes keys for XML, which we use to identify "objects" in an XML document. Section 3 describes the main components in our basic archiver and experimental results follow in Section 4. Section 5 describes how one can efficiently retrieve a version or the temporal history of an element using our archive representation. Related work and conclusions follow.

2. KEYS FOR XML

Keys for an XML document allow one to identify "objects" in the document. We use keys to identify the same "object" across all versions and the same "object" is stored only once in the archive.

Various forms of keys specification have been proposed for XML, for example, in XML standard [25] and XML Schema [27]. We use the system of key specification developed in [15] mainly because it provides a generic method for specifying *relative keys*. Relative keys allow one to define keys in a specified scope and hence one

can define a hierarchical set of keys, some of which are relative to others. Here, we briefly review the concept of keys developed in [15].

XML Model. We model an XML document as a tree whose nodes are labeled with (1) tag name (E-nodes), (2) attribute name, value pair (A-nodes), or (3) data values only (T-nodes). Only E-nodes can be internal nodes.

Value Equality. The value of a T-node is its data value. The value of an A-node is a pair consisting of its attribute name and attribute value. The value of an E-node consists of its tag name and two things: (1) a possibly empty list of values of its E and T children nodes according to the document order, and (2) a possibly empty set of values of its A children nodes. Two nodes are *value equal* if they agree on their value, i.e, the trees rooted at the nodes are isomorphic by an isomorphism that is identity on string values. Finally, $=_v$ denotes value equality.

Path Expression. A path expression is a sequence of node names – tag or attribute names. Our path language consists of the following: (1) the empty path "ϵ", (2) a node name, and (3) the concatenation of paths P/Q where P and Q are paths defined by these rules. We use "/" as the path concatenator just as XPath [8] uses it as a location step separator. In XPath, "/" alone or at the beginning of an XPath expression selects an element above the document root. We disallow concatenations where Q begins with "/".

Key. A *key* is a pair $(Q, \{P_1, ..., P_k\})$ where Q and P_i, $i \in [1, k]$ are path expressions. Informally, Q (*target path*) identifies a *target set* of nodes reachable from some context node and this target set of nodes satisfy the key constraints given by the key paths, P_i, $i \in [1, k]$. This is analogous to relational database where Q is a relation name and P_i form the key for that relation. A formal definition is given below. We denote by $n[\![P]\!]$ the set of nodes reachable from node n via path P.

Definition. A node n *satisfies* a key $(Q, \{P_1, \ldots, P_k\})$ iff each P_i is required to exist and is unique for any node in $n[\![Q]\!]$ and for any n_1, n_2 in $n[\![Q]\!]$, if for all $i \in [1, k]$ $n_1[\![P_i]\!] =_v n_2[\![P_i]\!]$ then $n_1 = n_2$. That is,

• $\forall n' \in n[\![Q]\!], n'[\![P_i]\!]$ exists and is unique for every $i \in [1, k]$.

• $\forall\, n_1,\, n_2 \in n[\![Q]\!]\ (\bigwedge_{1 \le i \le k} n_1[\![P_i]\!] =_v n_2[\![P_i]\!]) \rightarrow n_1 = n_2$

In the definition above, $=_v$ denotes value equality and $=$ denotes node equality (whether two nodes are the exact same node).

Example. The node <DB> below does not satisfy the key (A,{B}) but satisfies the key (A,{C}).

```
<DB>
   <A><B>1</B><C>1</C></A> <A><B>1</B><C>2</C></A>
</DB>
```

Relative Key. Keys as discussed above are defined with respect to some node. We often would like to describe the key of some node dependent on the key of an ancestor node much like *weak entities* in relational databases [17]. For example, the key of a weak entity consists of its parent's key and its key (e.g. course Math120, section B). Such dependent keys can be expressed through relative keys as defined below.

Definition. A document satisfies a *relative key* $(Q,(Q',S))$ iff for all nodes n in $[\![Q]\!]$, n satisfies the key (Q', S).

Q' identifies the target set relative to the *context path* Q. Q is always defined with respect to the document root. In other words, "/" is always a prefix of Q.

The key $(Q, (Q',S))$ is different from $(\epsilon,(Q/Q', S))$. The former defines the key (Q',S) with respect to a node reachable by path Q from the root. In other words, within each node reachable by path Q from the root, the set of nodes reachable by path Q' from that node must have distinct S values. The latter defines the key $(Q/Q', S)$ with respect to the root node. All nodes, reachable by path Q/Q' from the root, must have distinct S values. Observe, however, that if a document satisfies the key $(\epsilon,(Q/Q', S))$, it must also satisfy the key $(Q,(Q',S))$. However, the converse is not always true. Observe also that whenever we have a key $(Q,(Q',\{P_1, ..., P_k\}))$, the keys $(Q/Q',(P_i,\{\}))$, $i \in [1, k]$ are implied.

For the rest of the paper, we use the word key to mean a relative key.

Keys for Company Database. The company database shown before satisfies the following constraints which can be expressed as keys.

- (/,(db,{})). There is at most one db element at the root.
- (/db,(address,{})). There is at most one address under db node.
- (/db,(emp,{id})). Every employee within a db element can be uniquely identified by his id subelement.
- (/db/emp,(name,{})), (/db/emp,(sal,{})), (/db/emp,(tel,{})). There can be at most one name node for each employee. Similarly for sal and tel.

The example here consists of mostly keys with empty key paths. In the keys that hold for our experimental scientific data, keys are more complex. We remark that for documents which are standard and consistent representations of relations in XML, the set of keys can be automatically generated from the relational schema.

Some Terminology. We say a node is *keyed* if it has a key. In other words the sequence of tag names from root to this node is equal to the concatenation of context and target path of some key. Given a key $(Q, (Q', \{P_1, ..., P_k\}))$ and a keyed node with path Q/Q', the value v_i rooted under the path P_i, $i \in [1, k]$ of the keyed node is a *key path value* and the keyed node has the *key value* $\{P_1 = v_1, ..., P_k = v_k\}$. Given a set of keys, we consider the paths given by the concatenation of context and target paths for every key in the key specification. We say a path in this set is a *frontier path* if and only if it is not a proper prefix of some other path in the set. A node is a *frontier node* if and only if the sequence of tag names from root to that node equals to some frontier path. In other words, a frontier

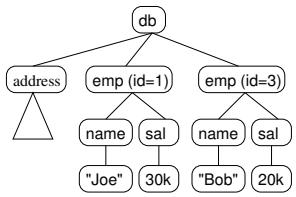


Figure 4: Annotated representation of version 3

node is the deepest possible keyed node. For example, the key specification given above has frontier paths /db/address, /db/emp/id, /db/emp/name, /db/emp/sal and /db/emp/tel. name is a frontier node but emp is not. Obviously, there can be unkeyed nodes beyond frontier nodes. For example, there may be firstname and lastname nodes under name nodes. Observe that frontier paths correspond to keys with empty key paths.

Assumptions about Key Structure. We make three assumptions about keys. First, we assume our keys are defined level-by-level. In other words, all keys are defined relative to its parent (not grandparent, etc.). For example, to identify a name node which lies on the path /db/emp/name requires one to first identify the correct db and emp nodes. Our second assumption is that given a set of keys which a document satisfies, any node that does not occur beneath frontier nodes is keyed. For example, there cannot be an email subelement directly under emp nodes. In other words, we assume that our keys "cover" all nodes that do not occur beneath frontier nodes. Our last assumption is that nodes beneath some key path cannot be keyed. That is, we assume that there cannot be a key $(Q_1, (Q'_1, \{P_1, ..., P_k\}))$ where $k \geq 1$ and another key $(Q_2, (Q'_2, \{...\}))$ such that $Q_1/Q'_1/P_i$ for some $i \in [1, k]$ is a proper prefix of Q_2/Q'_2. The last assumption is necessary to ensure that key values do not change as a result of reordering keyed nodes (When nodes are merged into the archive, they may occur in a different order from that in the version). Although these assumptions may seem restrictive, we note that our experimental data naturally adhere to these assumptions.

3. MAIN MODULES

In this section, we will describe our main modules, Annotate_Keys and Nested_Merge. Our archiver takes a new version, an archive, and a key specification as input. The new version and archive are assumed to obey the same key specification. Annotate_Keys annotates all the keyed elements in the version and the archive with their key values so that every element can be distinguished by its annotation. Then, Nested_Merge merges the annotated version and archive together into a new archive. Figure 3 illustrates this architecture.

3.1 Annotate Keys

As a preprocessing step for Nested_Merge, we first annotate each keyed node in an XML document with its key value so that each node can be uniquely identified by its path and key value. For example, Annotate_Keys will transform version 3 in Figure 1 into the representation shown in Figure 4. Observe that emp nodes are annotated with their key values, for e.g. emp(id=1). We omit the id subelement of emp because they are already stored as annotations. The resulting tree is such that every keyed node can be identified by the path from the root to that node by taking into account the annotations as well. Nested Merge, described in the next section, can then easily identify nodes in the archive and new version that correspond to each other based on those paths.

Keyed nodes can be annotated with its key values with a simple scan through the document. As we scan through the document in document order (preorder), we memorize the value under a key path whenever we encounter a node containing a key path value. By the time, we exit a keyed node, we would have its key value (all the key path values) and hence, we could store the key value at the keyed node. The procedure for annotating keys for an archive is similar to that for a version except that one has to also handle the timestamps in an archive. Since copying key values to a keyed node can increase the size of the document, especially when key values are large XML values, we have a technique for computing a small *fingerprint* of a key value and keyed nodes can be annotated with its fingerprints instead of actual key values. The full algorithm is described in detail in [2] and we omit the discussion here.

Data	Size	No. of Nodes	Height
OMIM	27.0MB	206466	5
SWISS-PROT	436.2MB	10903568	6
XMark	11.4MB	184974	12

Table 1: Various statistics of our data

3.2 Nested Merge

The core module in our archiving system is the Nested Merge module. Nested_Merge takes as input the annotated archive and the new version, and produces as output another archive which now contains information about the new version. The main idea behind nested merge is to first identify elements or nodes that correspond to each other in the archive and incoming version through keys. A node in the incoming version is merged into the corresponding node in the archive during the merge process. Whenever a node is merged into the archive, the set of timestamps associated with that node in the archive is augmented with the new version number. Nodes in the version that do not have corresponding nodes in the archive are simply added to the archive with the new version number as timestamp. Nodes in the archive that do not have corresponding nodes in the incoming version will not have the new version number in their timestamp.

Example. Figure 5 shows an example of nested merge when version 3 of Figure 1 is merged into the archive containing versions 1 and 2. The arrows indicate the correspondences between elements, identified through the set of keys given in Section 2. Notice that the node

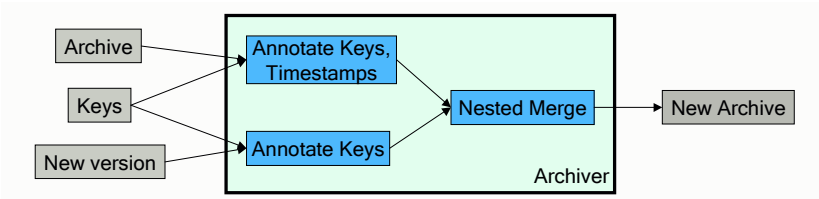


Figure 3: Main modules in our archiver

emp(id=3) in the new archive contains only the timestamp t=3 since it does not have a corresponding node in the old archive.

The pseudocode of the algorithm is described below. We assume the annotated archive contains versions 1 through $i-1$ and the annotated incoming version is a document containing version i. To simplify discussion, we assume that elements, except timestamp elements, do not contain attributes. The pseudocode below assumes that the archive is non-empty. When an archive is first created with one version, we simply add a timestamp to the root of that version. For any node x in the archive, let $t(x)$ denote the timestamps annotated on node x. Let $l(x)$ denote the label and key value of node x. Let $v(x)$ denote the XML value rooted at node x and including x and $v^-(x)$ denote the XML value rooted at node x but not including x. We call the algorithm Nested_Merge(r_A,r_D,{}), where r_A and r_D are the root node of the archive and the new version, respectively. The last argument contains an inherited timestamp, initially empty.

Algorithm Nested_Merge(x,y,T)

If $t(x)$ exists then 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 every node in X is not a timestamp node then
    If $v^-(x)$ and $v^-(y)$ are different then
      add <T t="$T-\{i\}$">$v^-(x)$</T>
      and <T t="i">$v^-(y)$</T> as a 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 <T t="i">$v(y)$</T> as a subtree of x.
Else
  Let $XY = \{(x', y') \mid x' \in X, y' \in Y, l(x')$ and $l(y')$ 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 $(x', y') \in XY$
    (a) Nested_Merge(x',y',T)
  For every $x' \in X'$
    (b) If $t(x')$ does not exist then let $t(x')$ be $T - \{i\}$.
  For every $y' \in Y'$
    (c) Let $t(y') = \{i\}$ and place $v(y')$ as a child node of x.

The algorithm first determines the current set of timestamps. It is i added to $t(x)$ if $t(x)$ exists. Otherwise, timestamps are inherited from its parent. Observe that $t(r_A)$ always exists. It is a property of the algorithm that $l(x)$ and $l(y)$ are the same whenever Nested_Merge(x,y,T) is invoked. The algorithm then proceeds by checking if y is a frontier node.

Frontier nodes are handled specially because there are no keyed nodes beyond frontier nodes. Thus the recursive merging of identical nodes no longer applies for the descendents of frontier nodes. The children nodes of any frontier node have the property that either they are all timestamp nodes or none of them is a timestamp node. In Figure 2, sal of Joe is an example of a frontier node whose children are all timestamp nodes, tel of Joe is a frontier node none of whose children are timestamp nodes. The transition from having no timestamp children nodes to all timestamp children nodes occurs when a merged version is such that contents of the frontier nodes to be merged differs from that in the archive. In Figure 5, when version 3 is merged, the value of sal of Joe differs from that in the archive. Hence timestamps are used to enclose the salary values at the respective times in the new archive.

If y is not a frontier node, we partition nodes in X and Y into three sets: XY contains pairs of nodes from X and Y respectively that corresponds to one another, i.e. with the same label and key value. Observe that by the property of keys, for every node in X, there can be at most one node in Y with the same label and key value. X' (resp. Y') consists of nodes in X (resp. Y) where there does not exist any node in Y (resp. X) that corresponds to one another. Nested merge is recursively called on pairs of nodes in XY, inheriting the current timestamp T. To ensure that nodes of X' no longer exist at time i, timestamp T excluding i is annotated on nodes of X' if they do not already contain timestamps that terminate earlier than i. Subtrees rooted at nodes of Y' are attached as a subtrees of x and they are annotated with a timestamp i since they only begin to exist at time i.

Analysis. We analyze the running time of Nested_Merge to show that it is $O(N \log N)$ where $N = \max(N_A, N_D)$. Here, N_A and N_D are the number of nodes in the archive (A) and version (D) respectively. We assume the set of frontier paths are kept in a hash table. This will allow us to determine if a given path (or node) is a frontier path in constant time. Let d_A and d_D be the maximum degree of A and D respectively. Hence X and Y can be determined in d_A and d_D time. To compare if two XML

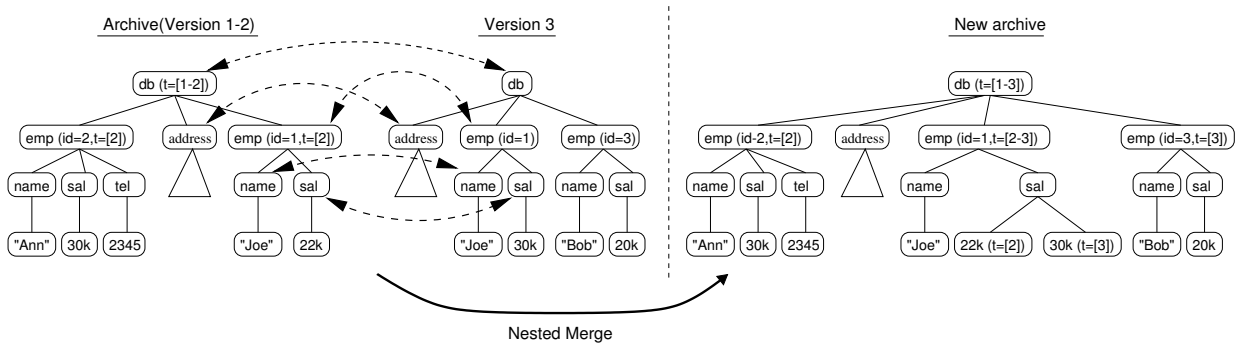


Figure 5: Merging version 3 into the archive containing versions 1 and 2

values are the same, we compare their *fingerprints* (or signatures). We assume that the fingerprints of these values are taken during the Annotate_Key phase. The details are described in [2]. Since fingerprints are small, we will assume for convenience that the comparison of two fingerprint values take constant time. Hence the statements, condition on y being a frontier node, executes in $O(d_A + d_D)$ time.

In case y is not a frontier node, the algorithm proceeds to determine XY, X' and Y' and appropriate actions, (a), (b) and (c), are taken for nodes in each set. Our implementation assumes that X and Y are sorted in ascending order according to their key values and a merge-sort is done on the sorted nodes: Start with the first node of each sorted sequence X and Y. Call them x' and y'. (1) If $l(x')$ and $l(y')$ are the same then perform action (a). (2) If $l(x') < l(y')$ then perform the action (b) on x' and let x' be the next node in the sorted sequence X. (3) Otherwise, perform action (c) on y' and let y' be the next node in the sorted sequence Y. We repeat this procedure until we run out of nodes on either sequence. If X (or Y) is empty first, we perform action (c) (or (b)) on the rest of the nodes in Y (or X). In the worst case, every node in A and D is sorted at some point. Hence Nested_Merge takes $O(N \log N)$ time.

Further Compaction. To obtain a further compact archive representation, one can apply diff-based techniques on XML values beneath frontier nodes. Within the frontier node, we represent the contents that remain the same across versions only once and mark the parts that differ by timestamps. This is much like the approach that SCCS [21] adopts. In this way, instead of representing XML values of these nodes under the respective timestamps, we represent only their difference. The advantage of this technique arises when XML values differ only slightly across versions.

4. EXPERIMENTAL RESULTS

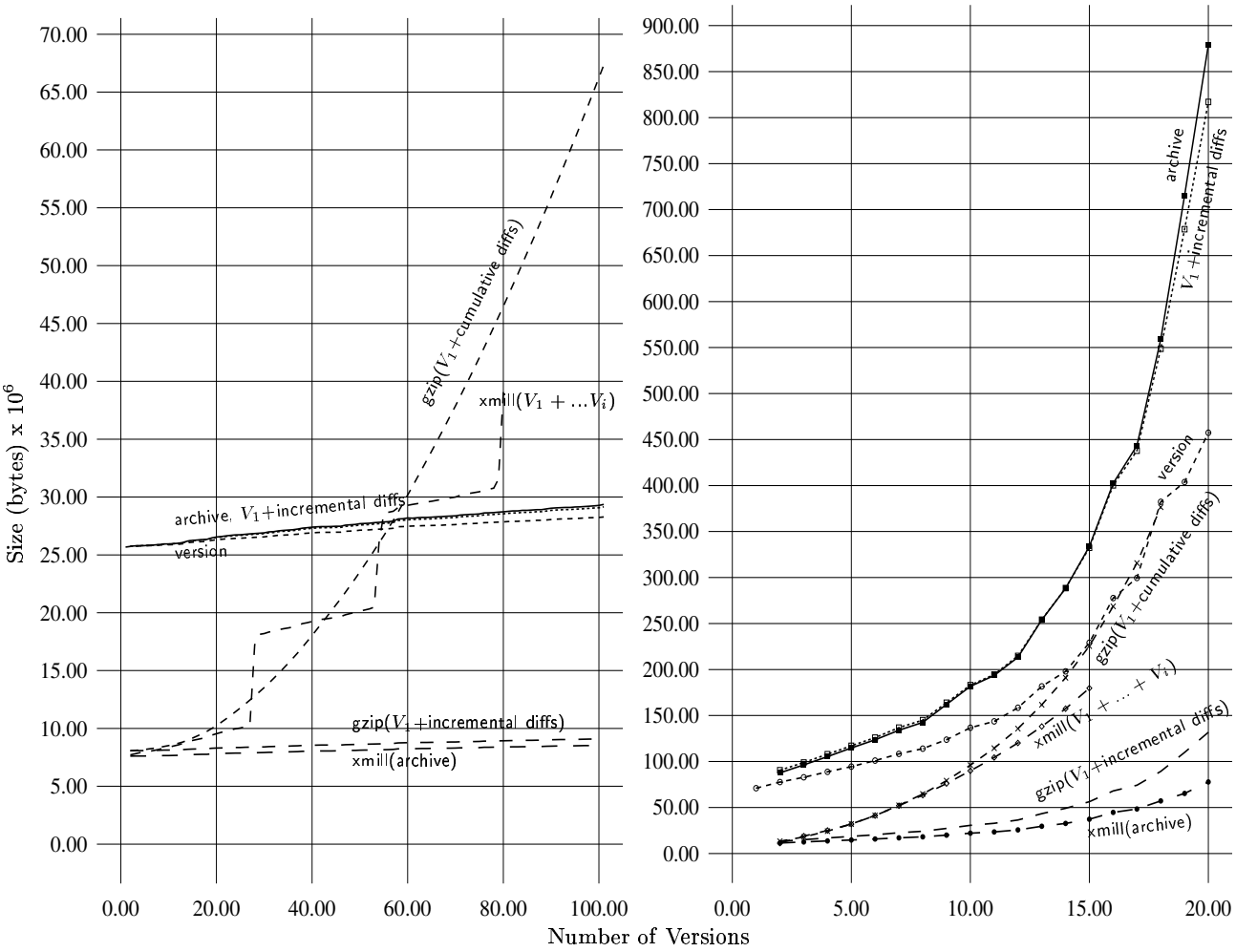
The main finding of our experiments is that our archive requires only slightly more space than the diff-based approach on real scientific data. Moreover, our results also show that our compressed archive is better than any other compressed repository that keeps all versions in terms of space efficiency.

Data. We tested our archiver on three datasets: OMIM [14], SWISS-PROT [1], and XMark [22]. OMIM and SWISS-PROT are real scientific data as described in Section 1. XMark is a benchmark database containing synthetic auction data. We wrote a change simulator for XMark and artificially generated versions of data with various change rates. In the experimental result we show in Figure 7, the versions used in the experiment are generated from the previous version by deleting 10% of the elements in the data, inserting the same number of new elements, and modifying text data for 10% of the elements in the data. As a remark, the deletion/insertion/modification ratios between two versions of OMIM and SWISS-PROT are roughly 0.02%/0.2%/0.03% and 14%/26%/1.2% respectively. Various other statistics of these data can be found in Table 1.

Experiments. The main purpose of our experiments is to answer the following question: How does the storage space performance of our technique compare with diff-based techniques? There are many variants of diff-based approaches, but we identified two likely competitive candidates: (1) *incremental diff approach* which stores the first version and diffs of every successive pairs of versions in a repository, and (2) *cumulative diff approach* which stores the first version and diffs of every version from the first version.

Cumulative diffs is obviously worse than incremental diffs in storage efficiency. However it has the advantage of being fast in retrieving any version. Hence if the storage space required by cumulative diffs is not far from that required by incremental diffs, cumulative diffs would be a tougher competitor for us. Our experiments, however, showed that cumulative diffs quickly suffers from the large storage space when the number of versions gets larger. Henceforth, we concentrate on comparisons between our archive and incremental diff approach. The experimental results with cumulative diffs are shown in [2].

In addition to the choice of incremental diffs or cumulative diffs, we also have a choice of tree diffs or line diffs. Tree diffs have been extensively studied [10, 11, 19, 20, 6] and we used XML-Diff [7], which is implemented for XML and is downloadable from the Web. However, when compared with line diff, XML-Diff in-



(a) OMIM (over $3-4$ months)

(b) SWISS-PROT (over ≈ 5 yrs)

Figure 6: Storage performance on OMIM and SWISS-PROT data.

curred a significantly higher space overhead (and it was also not robust: we could only run it on some of our data). Since our XML data is formatted in such a way that each element is represented by one or more consecutive lines separate from other elements, line diff gives a compact representation for small changes. For this reason, we chose line diff for the comparison. In all our experiments, we used the standard unix diff command with "-d" option to compute the smallest edit scripts. Hence, the sizes of our diff repositories are always the smallest possible.

We also examined how our approach and the diff-based approach perform in combination with compression techniques. We compress the diff-based repositories with gzip (a standard file compression tool) and we compress our archives with XMill (an XML compression tool) [12]. We also experimented with compressing cumulative diffs with gzip since compressed cumulative diffs may be small. As another possible competitor, we also put all the versions side by side into one XML tree and compress it with XMill. Both gzip and XMill were run with "-9" option to give the best possible compression.

In all the graphs, line version shows the size of each version, and line archive shows the size of our archives storing up to each version. Line V_1+incremental diffs shows the size of incremental diff repositories, i.e. the total size of the first version and all the incremental diffs. Line xmill(V_1+...+V_i) refers to the size of the concatenation of all the versions with XMill applied. Line gzip(V_1+incremental diffs) and line gzip(V_1+cumulative diffs) refers to the size of incremental diff repository and cumulative diff repository with gzip applied, respectively. Finally, line xmill(archive) shows the size of our archives with XMill applied.

4.1 Comparison with Diff

The graphs in Figure 6 show the results of the experiments with OMIM and SWISS-PROT. In those graphs, we see that the incremental diff approach marginally

outperforms our approach for most cases. For OMIM, the archive size is never 1% more than the size of incremental diffs, and for SWISS-PROT, the archive size is never 8% more than the size of incremental diffs.

Observe that the data stored in our approach and the diff approach are the same (although they are organized in different ways, i.e. grouped by elements v.s. grouped by time). The small difference between the storage space in the two approaches mainly comes from the difference between the size of timestamps in our archives and the size of "markers" in diff scripts. If we have a huge number of changes of very small data, such as a text data of length 1, the ratio of that difference to the size of the archive can be considerably large. Such an extreme situation, however, rarely occurs in the experimental data we used.

The graph in Figure 7 shows the results of the experiments with XMark. In this case, our approach marginally outperforms the diff approach. The main reason for this is that our change simulator modifies text data to randomly generated text, and text data sometimes happen to be modified to their old values. While incremental diff approach is forced to store those values multiple times in different deltas, our approach stores the value only once but timestamps are augmented to include multiple intervals. Clearly, the former incurs a higher overhead than the latter in most cases.

The scenario just described shows that, in the extreme case where the same data is repeatedly deleted and inserted, our approach is likely to outperform incremental diff approach by a big margin. On the other extreme, there may be insertions and deletions of highly similar data (though semantically different because they have different key values) over the versions. In this case diffs usually have a compact representation (by storing only the difference between them) while our approach is forced to store the highly similar elements separately. Both such extreme cases, however, rarely happens in scientific data.

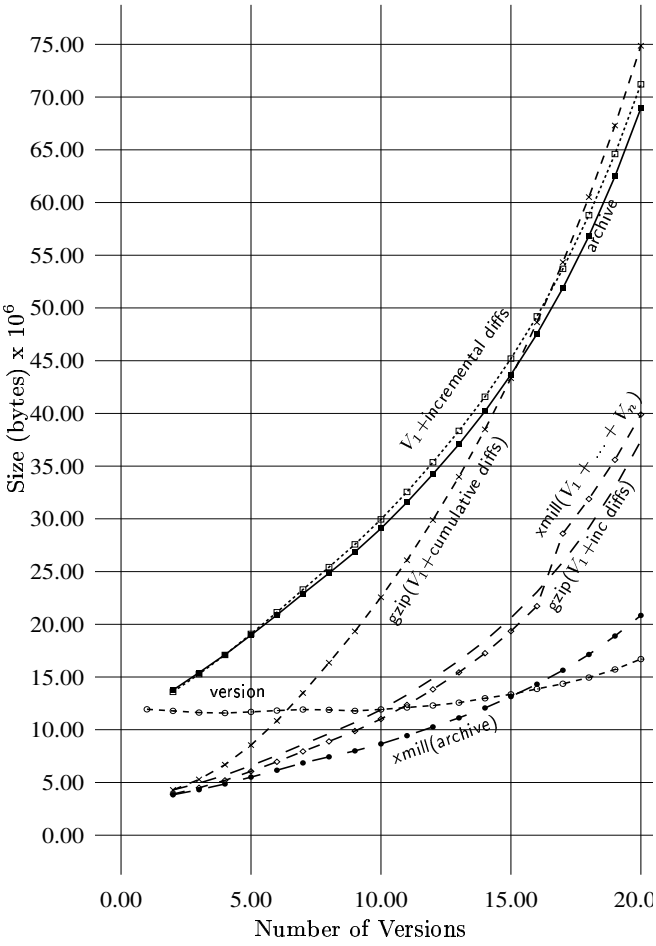


Figure 7: Storage performance on XMark data under 10% insertion + 10% deletion + 10% modification.

4.2 Interaction with Compression

We have seen that incremental diff technique may outperform our technique in terms of storage space. However, when compression is applied to both our archives and incremental diff repositories, our archives consistently uses less storage space in all our experiments. For both OMIM and SWISS-PROT (in Figure 6), we see that XMill applied on our archive (xmill(archive) line) outperforms any other approach even during the times when the incremental diff repository is clearly smaller than our archive without compressions. For instance, in Figure 6(b), as our archive starts to perform worse than the incremental diff repository after the version 17, our compressed archive continues to outperform the compressed diff repository by a growing margin. By the time version 20 of SWISS-PROT is added, the storage space used by our archive is clearly more than incremental-diff. However, gzip(incremental-diff) uses clearly more storage space than xmill(archive) by about the same magnitude.

This reversal of storage space efficiency occurs because our archive representation groups data according to elements in XML format, and XMill can exploit this specific structure to obtain a better compression ratio than general compression tools like gzip. XMill groups text data according to the names of the elements in which they occur and compresses each group separately. Since text data that belong to elements of the same name tend to be fairly similar, high compression ratios can usually be achieved. More surprising is the fact that this reversal of storage space performance occurs even when there are many insertions and deletions of similar but semantically different elements (the extreme case described earlier). Even when the size of incremental diffs is far smaller than the size of our archive, our compressed archive is still smaller than compressed incremental diffs. The results of these experiments are shown in [2].

Our graphs also show that a simple application of XMill to the concatenation of all the versions does not work as well as the our archive compressed with XMill (see lines xmill($V_1 + ... + V_i$) and xmill(archive)). Note that the stepping in line xmill($V_1 + ... + V_i$) in Figure 6(a) is due

to the fact that XMill with "dictionary compression" allocates new space whenever the dictionary at hand becomes full. Finally, the combination of gzip and cumulative diffs achieves rather high compression ratio. However, it still does not perform as well as the XMill applied to our archive.

As a remark, in Figure 6(b), we reached the system file size limit of 2^{31} when trying to add version 19 to V_1+cumulative-diffs repository before applying gzip. The limit is also reached when trying to add version 16 to the $V_1 + ... + V_{15}$ repository before XMill can be applied and hence the discontinuity in lines gzip(V_1+cumulative-diffs) and xmill($V_1 + ... + V_i$) in this graph.

5. EFFICIENT RETRIEVALS

So far, we have seen that a new version can be efficiently merged into an existing archive. Our experiments also show that the space overhead of our archive is comparable to traditional incremental diff approach for scientific data. Moreover, our compressed archive is always better than compressed diff repository in terms of space efficiency. We now show that because the structure of our archive is conceptually meaningful, it is easy to retrieve a version and the temporal history of an element efficiently from the archive by building simple index structures on top of the archive. A version can be retrieved in time roughly proportional to the size of the version and the temporal history of an element can be retrieved in time roughly proportional to the size of its key.

5.1 Retrieving a Version

The structure of the archive is such that a simple scan through the archive can retrieve any version. Whenever a timestamp tag is encountered, we output the contents of the timestamp element only if the required version number is in the range of the timestamps. However this takes time proportional to the size of the archive. To retrieve a version using incremental diff-based technique, this may require one to "roll back/forward" several times depending on how far back/forward the version is. We describe here a simple scheme that allows a version to be retrieved in time roughly proportional to the size of the version, regardless of how far back/forward the version may be. We achieve this by building simple auxiliary timestamp binary trees on top of the archive.

Timestamp Trees. The purpose of a timestamp tree is to direct the search for the relevant nodes of a version in the archive. We will assume a timestamp tree for each non-timestamp node (a node which is not representing the timestamp element) in the archive. Given a node x in the archive with k children, the corresponding timestamp tree is a binary tree with k leaves, one for each child of x. Every leaf of the tree has two kinds of information: the timestamps and the offset to the corresponding child node in the archive. Each internal node of the tree consists of the union of timestamps of its children nodes. We will describe how to construct this binary tree later.

Retrieve. We wish to retrieve a version i from the archive in time roughly proportional to the size of the version. Consider a node x in the archive and suppose it has k children of which only α belong to version i. The naive approach checks for each node if it is relevant to version i, i.e, has timestamp i. To look for the relevant children nodes, we first look for the timestamp tree corresponding to node x through the offset in x. Then we search down the the timestamp binary tree by starting at the root r of the binary tree and check if $i \in t(r)$ (where $t(x)$ denotes the timestamps of node x). If $i \notin t(r)$, we stop. This means that neither of the two children nodes of r is relevant for version i. If yes, we recurse on the two child nodes to continue the search. We repeat this search down the tree. Assuming that we have a complete binary tree and k and α are powers of 2, at most α nodes will be searched in each level after depth $\log \alpha$. Once we reach the leaf nodes, we will know the relevant α nodes in the archive through the offset values stored at the leaves in the binary tree. We also keep track of the number of nodes we have searched in the binary tree so far and stop at the point when we have searched a total of k nodes. If the threshold of k is reached before reaching the leaf nodes, we simply search all k leaf nodes. Observe that if $i \in t(r)$ then $\alpha \geq 1$. With this strategy, we will either probe $2\alpha - 1 + 2\alpha \log(k/\alpha)$ nodes or $2k$ nodes of the binary tree. We search $2\alpha - 1 + 2\alpha \log(k/\alpha)$ nodes when $\alpha \leq k/8$ and $2k$ nodes when $\alpha > k/8$. In the former case, we are at worst factor of $\log k$ away from the optimal number of probes while in the latter case, we are a constant factor away from the optimal. In largely accretive data where data changes infrequently, it is usually the case that $\alpha > k/8$ when a version is retrieved. In the calculations so far, we have ignored the time to test if i is in the set of time intervals of each node. Let m be the maximum number of time intervals among the k nodes. The root node of the binary tree can have at worst mk number of time intervals. Hence the membership test will take $O(\max(\log m, \log k))$ time at each node.

Constructing Timestamp Trees. Given an archive in XML, one can construct a timestamp tree for each non-timestamp node with a single scan through the archive. The idea is to first collect the k children nodes of a node x along with their respective timestamps and offsets into the archive file. We build on these k leaf nodes into a binary tree by pairing nodes repeatedly in a bottom-up manner and taking the union of timestamps of the children nodes for each internal node. Finally we append the binary tree in another file and write the offset to this binary tree in node x of the archive. Internal nodes of the binary tree written to file will contain offsets to their child nodes in order for one to skip to the irrelevant nodes of the binary tree during the search.

The timestamp trees are created each time a new version arrives and after nested merge is applied. The time needed for creating this data structure is easily offset by the efficiency of answering retrieval queries. We have the following correctness property for our algorithms in the sense that we always retrieve what we archived, assuming that the order among keyed nodes does not matter. Let Retrieve(A,i) be the function that retrieves version i from archive A.

THEOREM 5.1. *If a document D is merged into an archive A as version i then Retrieve(A,i) returns D.*

5.2 Retrieving the Temporal History of an Element

Given the key of an element, one would like to retrieve the temporal history of this element, i.e, the times at which this element exists. This would allow us to answer queries such as when an element first/last exists and how it has evolved through history. For example, the history of employee Joe in the company database, given by the path /db/emp[id=1], is 2,3 and 5.

We first observe that to retrieve the temporal history of an element with incremental diff approach would probably require significant reasoning and tracing through all the deltas. In our case, a simple scan through the archive can already retrieve the history of any element although this can be inefficient. We sketch a naive scheme that would allow one to answer this query in time $O(l \log d)$ where l is the length of the key and d is the maximum degree of the archive. The idea is to maintain for each keyed node x in the archive, a sorted list of children keyed nodes as the auxillary structure. Each node in the sorted list contains its key, an offset to the respective node in the archive and an offset to the timestamp node which it inherits the timestamp from. A binary search can be performed on the sorted list to find the element with the desired key. We assume that these information are kept in a fixed size record. To retrieve the correct child node of x with the desired key, we first find the offset to this sorted list of children nodes from x. Perform a binary search to locate the desired child node x' in this list. If not found, the required element does not exist in the archive. Otherwise, locate the corresponding node in the archive using the offset in the record containing x'. We repeat this procedure for every key in the path until we find desired element for the given path. If a node has a large number of children nodes, one can consider building more sophisticated index structures for these children nodes such as a B+ tree.

One can construct a sorted list of children keyed nodes for each keyed node with a single scan through the archive. The idea is to first collect the set of keyed children nodes of each keyed node x. We keep track for each node, the offset in the archive and the offset of the timestamp node which it inherits from. Before we exit a node x, we sort the list and keep the offset of this sorted list in node x.

6. RELATED WORK

There has been considerable research on version management systems and we highlight here the recent ones. The approach taken by Xyleme system [13] is diff-based [6]. They store the latest version together with all forward completed deltas – changes between successive versions which can also allow one to get to an earlier version by inverting deltas on the latest version. Another approach taken by Chien *et. al.* [18] is a reference-based versioning scheme. The first version is always fully stored. As subsequent versions arrive in the form of updates, only newly inserted objects are stored. Objects that remain unchanged are stored as references to the corresponding object in the previous version. It is argued that this approach provides better support for (temporal) queries since it preserves the logical structure of every version as opposed to diff. However, their approach assumes that the edits are known, i.e, the elements that has been modified, deleted or inserted are known. Our technique does not assume the existence of edit scripts, preserves the logical structure of every version and still makes it easy to discover the evolutionary history of any element.

Diff-based approaches for tree-like structures have been substantially studied in [10, 11, 19, 20, 6]. The general problem of finding the minimum cost edit distance between two ordered trees is studied by Zhang *et. al.* in [10, 11]. In [19], Chawathe *et. al.* made further restrictions to obtain a faster algorithm for the minimum cost edit distance. A heuristic change detection algorithm for unordered trees is also proposed in [20] with a richer set of edit operations. Another diff technique is proposed in [6] which uses signatures to find matchings and another top down phase to compute further matchings heuristically.

Software configuration management systems such as CVS [4] use diff-based techniques which stores the last version together with backward deltas based on line-diffs. Our archiving system is more like source code control system (SCCS) [21] which uses timestamps to mark when lines of text exists at various versions. However, the difference between SCCS and our archiving technique is that SCCS uses line-based diff and does not use keys. Therefore, if the same line is repeatedly inserted and deleted over the versions, a series of identical lines marked as inserted or deleted will be stored in the SCCS repository. Using keys in our archiving technique, our archive will only contain the line once with timestamps to mark when this line exists. Nested Merge bears resemblance to Merge Template idea in [23] which merges two XML documents according to a specified template. The template specifies which elements can be merged together, inserted or replaced.

Our work is largely inspired by the work of Driscoll *et. al.* [9] where they studied methods for making data structures *partially persistent* – updates and accesses are allowed on the latest version but only accesses are allowed on past versions. However, the set of edits is known and they do not exploit key information.

7. CONCLUSION

We have described an archiving technique for XML data with key structure. Our approach is the first archiving technique for XML data that is not diff-based. We have shown that one can efficiently add a version into an existing archive. Our experiments also show that our archiving technique is comparable to diff-based approaches in terms of space efficiency for scientific data. For example, SWISS-PROT archive is never more than 1.08 times the size of incremental-diff repository and OMIM archive is never more than 1.01 times the size

of incremental-diff repository. Our archiving technique also provides one with meaningful change descriptions — we are able to describe changes around "objects" or elements by identifying elements through keys. Diff techniques are based on minimizing edit costs which may give meaningless deltas as seen by the Person(Name, DateOfBirth, Address, Zip) example earlier. Moreover, since the structure of our archive is conceptually meaningful, it is easy to support various temporal queries on the archive, by building index structures on top of the archive. In addition, since our archive is yet another XML document, this allows one to directly leverage existing XML tools. Existing XML query languages such as XQuery [28] can be used to query our archive. It is also mentioned in [18] that the ideal solution is to represent the history of a versioned document as yet another XML document for this allows one to exploit web-browsers, style sheets, query processors and other tools that already support XML. Our experiments also showed that our archive works well in tandem with compression; our compressed archive is always more space efficient than compressed diff-based repositories in all data sets that we have used.

One issue that is not discussed in this paper is how one can extend the archiver to handle order among XML elements. One approach is to use tree alignment, which is a generalization of string alignment, instead of the nested merge. For now, we have implemented a prototype based on this idea, but experiments and the theoretical analysis on its efficiency is future work. Please refer to [2] for this issue. How one can extend our archiver to handle other definitions of keys found in [15] is also an interesting future work.

8. REFERENCES

[1] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL. *Nucleic Acids Research*, 28:45–48, 2000.

[2] P. Buneman, S. Khanna, K. Tajima, and W. Tan. Archiving Scientific Data. Technical report, University of Pennsylvania, 2002.

[3] The WWW Virtual Library of Cell Biology. http://vlib.org/Science/Cell_Biology/databases.shtml.

[4] Concurrent Versions System. Unix man pages – cvs.

[5] E. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[6] G. Cobena and S. Abiteboul and A. Marian. Detecting Changes in XML Documents. In *Int'l Conf. on Data Engineering*, 2001.

[7] XML TreeDiff. http://www.alphaworks.ibm.com/formula/xmltreediff.

[8] J. Clark and S. DeRose. XML Path Language (XPath). W3C Working Draft, November 1999. http://www.w3.org/TR/xpath.

[9] J. R. Driscoll and N. Sarnak and D. D. Sleator and R. E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[10] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245–1262, 1989.

[11] K. Zhang and D. Shasha. Fast algorithms for unit cost editing distance between trees. *Journal of Algorithms*, 11(6):581–621, 1990.

[12] H. Liefke and D. Suciu. XMill: an Efficient Compressor for XML Data. In *Proc. of ACM SIGMOD Int'l Conf. on Mgmt. of Data*, 2000.

[13] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-Centric Management of Versions in an XML Warehouse. In *Int'l Conf. of Very Large Data Bases*, 2001.

[14] Online Mendelian Inheritance in Man, OMIM (TM), 2000. http://www.ncbi.nlm.nih.gov/omim/.

[15] 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.

[16] The NIST Reference on Constants, Units, and Uncertainty. http://physics.nist.gov/cuu/Constants/links.html.

[17] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.

[18] S. Chien and V.J. Tsotras and C. Zaniolo. Efficient Management of Multiversion Documents by Object Referencing. In *Int'l Conf. of Very Large Data Bases*, 2001.

[19] S. S. Chawathe and A. Rajaraman and H. Garcia-Molina and J. Widom. Change Detection in Hierarchically Structured Information. In *Proc. of ACM SIGMOD Int'l Conf. on Mgmt. of Data*, 1996.

[20] S. S. Chawathe and H. Garcia-Molina. Meaningful Change Detection in Structured Data. In *Proc. of ACM SIGMOD Int'l Conf. on Mgmt. of Data*, 1997.

[21] Source Code Control System. Unix man pages – sccs.

[22] A.R. Schmidt, F. Waas, M.L. Kersten, D. Florescu, I. Manolescu, M.J. Carey, and R. Busse. The XML Benchmark Project. Technical report, INS-R0103, CWI, 2001. http://monetdb.cwi.nl/xml/index.html.

[23] K. Tufte and D. Maier. Aggregation and Accumulation of XML Data. *IEEE Data Engineering Bulletin*, 24(2):34–39, 2001.

[24] W. Miller and E. Myers. A file comparison program. *Software-Practice and Experience*, 15(11):1025–1040, 1985.

[25] W3C. Extensible Markup Language (XML) 1.0, Feb 1998. http://www.w3.org/TR/REC-xml.

[26] W3C. Namespaces in XML, January 1999. http://www.w3.org/TR/REC-xml-names.

[27] W3C. XML Schema Part 0: Primer, May 2000. http://www.w3.org/TR/xmlschema-0/.

[28] W3C. XQuery 1.0: An XML Query Language, June 2001. http://www.w3.org/TR/xquery/.