Change-Centric Management of Versions in an XML Warehouse

Serge Abiteboul, Grégory Cobéna, Amélie Marian, Laurent Mignet
Verso INRIA

Presented by:
Todd J. Green
University of Pennsylvania
November 16, 2004
Introduction

This paper concerns the management of versions of XML documents in a database. The idea is to somehow keep track of changes from revision to revision so that we can reconstruct what the document looked like at any given point in time. Main topics from the paper include:

- Edit Scripts
- Deltas
- Completed Deltas
- Implementation Strategies
- Management of Node Identifiers

I will cover these one by one.
Preliminaries

We model an XML documents as an ordered tree. We are interested in following the evolution of a document $V$ as a sequence of versions $V_1, \ldots, V_n$ of the document. Additionally, we assume that each node in the tree has a unique node identifier (more on this later). Throughout this talk, I will use $v$ to mean a vertex $v$ itself, or the identifier for $v$, interchangeably.
**Edit Scripts**

An *edit script* is any finite sequence of the following operations on a tree $S$:

1. $\text{delete}(m)$ — deletes the XML tree rooted in node $m$, where $m$ is not the root of $S$
2. $\text{insert}(n, k, T)$ — inserts the XML tree $T$ as the $k$-th child of $n$
3. $\text{move}(n, k, m)$ — moves the XML tree rooted in node $m$ to be the $k$-th child of $n$
4. $\text{update}(m, v)$ — changes the value of node $m$ to $v$

Additionally, each operation in an edit-script must be *consistent* with the tree $S$ (no out-of-bounds indices, etc).
Example: An Edit Script

1. update($v_3$, ‘d’)
2. move($v_0$, 2, $v_3$)
3. delete($v_1$)
4. update($v_3$, ‘e’)

$S =$

```
v0
  ┌──────┐
  │     │
  ├───v1───┐
  │     │
  └──────┘
     "a"
```

```
v0
  ┌──────┐
  │     │
  ├───v2───┐
  │     │
  └──────┘
     "b"
```

```
v0
  ┌──────┐
  │     │
  ├───v3───┐
  │     │
  └──────┘
     "e"
```

```
v0
  ┌──────┐
  │     │
  ├───v3───┐
  │     │
  └──────┘
     "b"
```

$T =$

```
v0
  ┌──────┐
  │     │
  ├───v3───┐
  │     │
  └──────┘
     "e"
```

```
v0
  ┌──────┐
  │     │
  ├───v2───┐
  │     │
  └──────┘
     "b"
```
Example: An Edit Script

Equivalently, we could have said:

1. delete($v_1$)
2. move($v_0$, 1, $v_3$)
3. update($v_3$, ‘e’)

$S = v_0 \quad v_1 \quad v_2 \quad v_3 \quad v_2 \quad v_0$

$T = v_0 \quad v_3 \quad v_2 \quad v_2 \quad v_0$

‘a’ ‘b’ ‘c’

‘e’ ‘b’
Edit Scripts

This example illustrates that with edit scripts, there may be many ways (indeed, infinitely many ways) to describe the changes between two consecutive versions. We would prefer to have a representation of a change that is more “normal” or “canonical”. This is the motivation for *deltas*.
## Deltas

Given two trees $S, T$ and a vertex $v$, denote by $p(v), i(v), m(v)$, and $S_v$ (resp. $q(v), j(v), n(v)$, and $T_v$) the parent, rank, value, and subtree of $v$ in $S$ (resp. in $T$). A *delta* from $S$ to $T$ is a set $\Delta$ of operations satisfying, for every vertex $v$, the following properties:

1. $\text{delete}(v) \in \Delta$ iff $v \in S \land v \not\in T \land p(v) \in T$;
2. $\text{insert}(q(v), j(v), T_v) \in \Delta$ iff $v \not\in S \land v \in T \land q(v) \in T$;
3. $\text{update}(v, n(v)) \in \Delta$ iff $m(v) \neq n(v)$;
4. $\text{move}(q, j, v) \in \Delta$ implies $q = q(v) \land j = j(v)$;
5. $v \in S \land v \in T$ implies the set of children of $v$ that were neither deleted, nor inserted, nor moved are the same in $S$ and $T$ and in the same order.
Deltas

A delta $\Delta$ from $S$ to $T$ is still not totally “canonical”, because there can be another delta $\Gamma$ from $S$ to $T$ with $\Delta \neq \Gamma$. However, $\Delta$ and $\Gamma$ must agree on all insert, delete, and update operations, and may disagree only on move operations. So that seems like progress.
Applying Deltas

A delta is a set of operations, so there is no explicit order for their evaluation given. Nevertheless, we have the following desirable property:

**Theorem 1 (Uniqueness).** Let $S$ be a tree and $\Delta$ a set of operations. Then there exists at most one $T$ such that $\Delta$ is a delta from $S$ to $T$.

This says that given $S$ and $\Delta$, $T$ is uniquely determined. That’s good. However, actually computing $T$ from $S$ and $\Delta$ is a little trickier than you might expect.
Applying deltas

The obvious thing is to apply the operations in the delta in some order (one where all the operations are consistent). This assumes, of course, that the following proposition holds:

**Proposition 1 (False Proposition).** Let $S$ and $T$ be trees and let $\Delta = \{op_1, \ldots, op_n\}$ be a delta from $S$ to $T$. Then there is some ordering of the operations $\langle op_{i_1}, \ldots, op_{i_n} \rangle$ such that $op_{i_n} (op_{i_{n-1}} \ldots (op_{i_1} (S)) \ldots) = T$.

But unfortunately, this proposition is false!
Applying deltas

Proof. We claim that the following delta is a counterexample to Proposition 1:

\[ \Delta = \{ \text{move}(v_2, 2, v_3), \text{move}(v_1, 2, v_5) \} \]

Clearly \( \Delta \) is a delta from \( S \) to \( T \). But evaluating \( \text{move}(v_2, 2, v_3) \) then \( \text{move}(v_1, 2, v_5) \) yields a tree where \( v_3 \) has rank 1, which is incorrect; while evaluating \( \text{move}(v_1, 2, v_5) \) then \( \text{move}(v_2, 2, v_3) \) results in a tree where \( v_5 \) has rank 1, which is also incorrect. \( \square \)
Applying deltas

This counterexample shows that, when applying a delta, we may have to adjust the indices in the *insert* and *move* operations to get the correct result.

This is just a technical detail, though.
**Completed Deltas**

Regular deltas have some shortcomings. In order to compute the inversion $\Delta^{-1}$ of a delta $\Delta$ from $S$ to $T$ (i.e., such that $\Delta^{-1}(T) = S$), one must in general look not just at $\Delta$ but also at the tree $S$. For example, consider the delta $\Delta = \{\text{update}(a, v_1)\}$ from tree $S$ to tree $T$. In order to “reverse” the delta, we have to look at $S$ to see what the old value of $v_1$ was. The same thing is true for composition of two deltas. To remedy this, we need to keep track of some more information. This motivates the introduction of completed deltas.
Completed Deltas

We introduce a new set of operators:

1. \(\text{delete}(v, k, T)\) — deletes the XML tree \(T\) whose root is the \(k\)-th child of node \(v\);
2. \(\text{update}(n, v, m)\) — where \(m\) is the old value;
3. \(\text{insert}(v, k, T)\) — inserts the XML tree \(T\) as the \(k\)-th child of \(v\);
4. \(\text{move}(q, j, v, p, i)\) — moves \(v\), the \(i\)-th child of \(p\), so that it becomes the \(j\)-th child of \(q\).

A set \(\Delta\) of these operations is a completed delta if there exist \(S\) and \(T\) such that \(\Delta\) is the set of operations that transforms \(S\) to \(T\).

Question: is this definition “tight”?
Completed Deltas

Sometimes, we might care only applying a completed delta $\Delta$ in one direction, instead of in both directions, for example because we have access to the full tree in one direction. For this purpose we introduce the concept of forward and backward deltas, denoted $\Delta_f$ and $\Delta_b$, respectively. Intuitively, the forward delta $\Delta_f$ is $\Delta$ pruned of any information not needed to compute $\Delta(S)$; the backward delta $\Delta_b$ is $\Delta$ pruned of any information not needed to compute $\Delta^{-1}(T)$. Note that the forward and backward deltas are simple deltas, rather than completed deltas.
Implementation Strategies

Completed deltas form a solid theoretical base for our implementation, because they allow us to compute the way the document looked at any given time, given the set of completed deltas and a single copy of the document at some point in time.

But, how exactly should we implement our versioning system? Which deltas do we keep track of? Which copies of the document?
Implementation Strategies

There are three basic strategies identified:

1. **Store first and last versions \( V_1 \) and \( V_n \) and forward deltas**
   
   +: deltas are simple, hence small
   
   -: retrieving a recent version \( V_i \) of document is expensive, since we have to start from \( V_1 \); two copies of full document stored

2. **Store last version \( V_n \) and backwards deltas**
   
   +: deltas are simple, hence small; retrieving recent version \( V_i \) of document is efficient
   
   -: have to compute the inverted deltas, which costs something

3. **Store a history** Not sure what this means, but apparently it doesn’t work too well for us.
Implementation Strategies

So the best approach of these seems to be

2. Store last version \( V_n \) and backwards deltas

and indeed, that is the approach they take in Xyleme.

Another detail is that they encode the deltas themselves as XML documents. This way, they can use the same storage system (Natix) for both documents and deltas.
Management of Node Identifiers

The model we have presented for describing deltas depends crucially on each node in the tree being associated with a unique identifier. This identifier could be kept in the document itself as an attribute of the node. However, for performance reasons (space-savings), they choose to decouple the identifiers from the nodes, and store them in a separate ID-map, which is a compact representation of the labeling of the tree with identifiers. An ID-map looks like

\[(1-3, 7-13, 5, 14-28) | 29\]

To assign IDs to nodes, you perform a postorder traversal of the tree, and hand out IDs from the list specified by the map. The number after the bar is the next number available for a new node. Let’s look at an example to see how it works.
The ID-map for this tree is: (1-3, 7-13, 5, 14-28) | 29.
The End