DEWS2006 6B-o1

派生関係の解析が可能な XML のバージョンモデルとアクセス制御への応用

林　良太郎†　　岩井原　瑞穂†
チャットウィチェンチャイ　ソムチャイ‡

†京都大学大学院情報学研究科
〒606-8501　京都市左京区吉田本町
‡県立長崎シーボルト大学国際情報学部
〒851-2195 長崎県西彼杵群長与町まなび野 1-1-1
E-mail: †iwaihara@db.soc.i.kyoto-u.ac.jp　‡somchaic@sun.ac.jp

あらまし　XML によるニュース配信や wiki などのデータでは、文書の更新が行われる一方で過去のデータもアクセス可能であることが多い。このようなバージョン化された XML 文書に対して読み書きの権限判定を行うアクセス制御として、派生関係にあるバージョン系列に同じアクセス制御ルールを適用することが考えられる。このようなアクセス制御ルールを記述できる、XPath にバージョンの軸を加えて拡張した XVerPath について述べ、差分をベースにした XML 文書バージョンの圧縮形式と、その上での XVerPath の評価方式について述べる。

キーワード　**XML、問合せ処理、セキュリティ**

１．はじめに

近年、様々な分野において XML 文書の利用が盛んに行われており、作成される XML 文書の量は急速に増え続けている。特に、ニュース配信や wiki などのデータでは、文書が随時更新されていく一方で、過去のデータへのアクセスも可能であることが多い。過去のデータに個人情報や企業の機密文書などの極秘情報を含んでいる場合など、バージョンを持つ XML 文書のアクセス制御は、重要な課題の一つである。バージョン管理に関しては、たとえば米国のウェブアーカイブ www.archive.org ではウェブページの変遷を時間軸に沿ってたどる機能を提供しており, また利用者参加型の百科事典 Wikipedia では, 事典項目ごとの利用者書き込みによる変更差分がすべて閲覧できるようになっている。これらは XML アーカイブの基本的なバージョン管理を行っているといえる一方で、細粒度のアクセス制御は提供していない。

これまでに様々な XML アクセス制御モデルが考案されてきたが、文献[2]は、 XML 文書のバージョン管理とアクセス管理の統合概念を初めて述べている。例えば、XML アーカイブにおいて著作権処理や課金処理のため公開を制限すべき部分が含まれている場合や、企業情報のアーカイブにおいて厳密な情報管理を行う必要がある場合に、バージョンの依存関係をもつ部分をまとめてアクセス制限を行うことである。 また文献[1]では XML 文書の更新操作によって生じる依存関係に基づいたアクセス制御モデルの概念と、そのための XPath に時間軸の概念を導入した言語 XVerPath を提案している。 本稿では文献[1]を実現するために重要な、XML 文書の更新差分をデータベース管理する手法を述べ、その上で XVerPath を評価する手法を提案する。

XML 文書が更新されるごとに生じるバージョンをデータベースに蓄積し、過去のバージョンへのアクセスを可能にし、任意の時刻における XML 文書のスナップショットの再現を可能することがバージョン管理の基本であるが、更新のたびに新しい XML 文書木を生成して保存していくと、文書の格納コストが膨大となる。本稿では、XML 文書木に様々な工夫を施すことによって、XML 文書が更新されても新たな文書木を生成することなく、一つの木に変更の差分を保存し、任意の時刻のスナップショットを再現する手法や、文書のノード単位での派生関係を求める方法を提案する。

本稿は以下、２節で関連性に基づくアクセス制御について述べ、3 節では提案手法の概要について述べる。それぞれの更新操作に対して、４節では差分木の処理を、５節では文書木に対する処理を述べ、６節で処理後の木に対するバージョン枝の判定方法について述べる。 ７節では実験について述べ、８節はまとめである。

２．関連性に基づくアクセス制御

本稿では、文献[1]で提案されている XVerPath のアイデアと“関連性に基づくアクセス制御”（Relevancy-Based Access Control）について述べる。

XVerPath は、XPath の基本的なサブセット(子および子孫軸, 親および祖先軸、述語、集合演算)に対し、ノード間の派生関係に基づくバージョン軸（バージョン親 vpar()、バージョン祖先 vanc(), バージョン子 vchild(), バージョン子孫 vdec()）を新たに導入し、さらにノードのタイムスタンプに基づいた述語を導入した言語である。XVerPath を用いることによって、あるノードの祖先ノードの集合や子孫ノードの集合が選択可能になり、これを用いて指定されるアクセス制御ルールは、派生関係にあるノード系列を単位としたアクセス制御や、ノードのタイムスタンプに基づくアクセス制御を実現する。XVerPath に導入されるバージョン軸の概念によれば、ノードの複製などの新旧のノード間にデータの変化がない場合に設定される no-change 枝と、ノードのテキストの更新の際に設定される updated 枝、ノードの置換操作の際に設定される replaced 枝の３種類の枝で新旧のノードが連結され、それぞれ更新操作前の古いノードをバージョン親、更新操作後の新しいノードをバージョン子として、これらの枝に基づいてバージョンの親子関係や先祖・子孫関係が定義される。さらにノードの作成時刻についての関数を設け、それを述語として記述することによって、ノードの作成時刻についての指定を実現する。

さらに文献[1]では、XVerPath を用いたアクセス制御機能として、関連性によるアクセス制御を導入している。これは XVerPath によって関連性クラスと呼ぶ文書集合を非明示的に定義しておき、あるノードにあらたなアクセス制御が導入されると、そのノードを含む関連性クラスのノードすべてに同一のアクセス制御を適用するというものである。定義可能な関連性にはおおまかに３種類あり、ノードの派生関係に基づくもの、文書のスキーマに基づくもの、ノード作成時刻など時間的な関連性に基づくもの、さらにこれらを混在させたものが考えられる。

文献[1]では、文書更新操作を枝ラベルとし、文書をノードとするデルタバージョングラフと、これに文書インスタンスを適用することにより生成される要素バージョングラフを用いて、ノード間に存在する派生関係を表現した。要素バージョングラフでは、更新によって新たに生成された文書木の各ノードと、その元になった古い文書木の各ノードとの間にバージョン枝を張る。さらにバージョン枝のラベルとして要素テキストの更新(updated 枝), 要素テキストの置き換え(replaced 枝), 無変更(no-change 枝)のラベル付けを行ない、バージョン枝のパスのラベルを考慮しながら探索することにより、あるノードの更新履歴を遡りもっとも古いノードを求めたり、あるいは無変更で同じ内容を持つノードの集合を求めたりといったことが可能である。さらにバージョンに関する選択条件を通常の XPath 質問と組み合わせることもできる。

文献[1]で提案されているようなアクセスポリシーの例として次のようなものを考える。

<Reader, /[vdate()>now()-7days]/Article.vanc(n,u,r), read, grant>

この例において“/[vdate()>now()-7days]/Article.vanc(n,u,r)”は、現在より７日前以降に作成された記事（Article）の先祖バージョンであるノード集合に評価される。つまりこのアクセスポリシーは、読者（Reader）に対して、現在より７日前以降に作成された記事の参照（read）を許可する（grant）ものである。このように、XVerPath によるノード評価を利用したアクセス制御を実現するのである。

要素バージョングラフをそのまま実現した場合、ひとつひとつの更新ごとに文書の実体が生成され、更新操作が繰り返されると要素バージョングラフが大きくなりすぎ、記憶効率が悪いと考えられる。そこで本稿では、XML 文書をデータベースに保存し、差分に基づいた圧縮方法によって、更新操作による記憶容量の膨張を防ぐ手法を提案する。これにより関連性に基づくアクセス制御が効率良く実現できると考えられる。

３．提案手法の概要

本稿では、XML 文書集合に対する過去から現在にいたるすべての差分を保存し、任意の時刻における XML 文書のスナップショットを再現しうる、単一の文書木を取り扱う。これをアーカイブ木と呼び、そのルートノードである Archive Root 以下に、更新操作の対象となる複数の XML 文書木と、更新履歴に関する情報を保持させるための構造である一つの差分木を持つ。図１にその概観を示したが、この図において差分木とは、Delta Root ノード以下の部分木である。差分木には、更新操作の度に Delta ノードが Delta Root の子ノードとして蓄積されていき、更新操作と一対一で対応する各 Delta ノードに各更新操作の情報を保持させる。

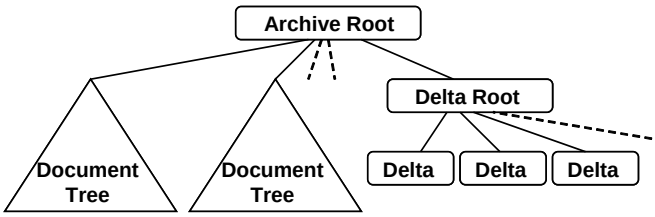


図１　アーカイブ木の概観

更新処理の対象となる文書木はすべてこのアーカイブ木の下で処理が施される。この文書木に含まれていない XML 文書に更新処理を行う場合、事前にこのアーカイブ木の部分木としてインポートされることになる。また、アーカイブ木への更新操作によって新たな文書木が生成されることはなく、削除されたノードも含め、過去のデータとなったノードすべてをアーカイブ木以下に保持する。文書木の各ノードに対していくつかの属性を付与し、同時に差分木に更新の履歴を残すことによって、更新操作によるノード間の派生関係の情報を保持し、任意の時刻のスナップショットを再現することを可能にする方法を

提案する。

文書木に対する更新操作は、文献[1]で採用されているものと同じ、次の６種類を想定する。

① delete(x)…ノード x を削除する。

② insert(x,y)…ノード x の子として、ノード y を挿入する。

③ update(x,c)…ノード x の内容を、c で書き換える。

④ replace(x,y)…ノード x を、新たにノード y で置換する。

⑤ copy(x,y)…ノード x を複製し、ノード y の子として挿入する。

⑥ move(x,y)…ノード x を移動させ、ノード y の子として挿入する。

なお、操作対象のノードがある部分木のルートである場合、③の update 操作を除き、部分木全体が操作対象となる。本稿で提案する手法では、これらの更新操作の指示に完全に従ってノード操作を行うのではなく、ノードに対する属性の付与やその変更、差分木内に Delta ノードを生成して情報を保持させるなどの工夫によって、それらの更新操作がなされたことを表現する。

前述した通り、このアーカイブ木のノードに対して、XVerPath を用いてアクセスすることになるが、ノードに付与された様々な属性と、差分木内の Delta ノードの持つ情報を利用することで、XVerPath 文字列で指定されるノードを選択する。

４．更新操作と差分木

更新操作がなされた時と、新しく文書がインポートされた時には、必ず一つの Delta ノードが生成され、差分木に加えられる。差分木への処理は、文書木への処理と同時に行われなければならない。ここでは、Delta ノードに付与される各属性の持つ意味や、差分木への具体的な処理について説明する。

Delta ノードは更新操作に関する情報を保持するため、いくつかの属性を付与される。与えられる属性によって一部異なっているが、すべての Delta ノードに共通しているものに number 属性と time 属性、さらに operation 属性がある。number 属性は各 Delta ノードに一意に与えられる自然数値であり、重複しないため自然数値と Delta ノードとが一対一で対応する。time 属性には更新操作がなされた時刻が保持されるため、Delta ノードは与えられた number 属性と time 属性を対応させる役割を果たす。文書木内で各ノードの作成・削除時刻を表現する際、この number 属性を利用することになる。operation 属性はどの種類の更新操作が行われたのかを記すために用いられる。

文書のインポートを含め、各更新操作における差分木での処理は以下の通りである。まず、どの更新操作でも共通しているのは次の処理である。

① 新しい Delta ノードを、Delta Root ノードの子として挿入する。（末子として挿入する。）

② 一つ上の Delta ノードの number 属性値を参照して、それよりも１多い自然数を number 属性値として与える。（もしくは Delta Root ノードに常に次の Delta にふさわしい number を保持しておいて、それを与えてもよい。）最初の Delta ノードには”0”を与える。

③ 更新操作が行われた時刻を time 属性として与える。

更新操作の種類によって異なる処理をするのは、次の処理である。なお、属性値として与えられる TID については後述するが、文書木内のすべてのノードに割り当てられる識別子で、各ノードの文書木内における位置を表すものである。

１．delete(x)

operation 属性として”delete”を与え、ノード x の TID を element 属性として与える。

２．insert(x,y)

operation 属性として”insert”を与え、ノード x の TID を parent 属性、ノード y の TID を child 属性として与える。

３．update(x,c)

operation 属性として”update”を与え、ノード x の TID を element 属性として与える。

４．replace(x,y)

operation 属性として”replace”を与え、ノード x の TID（これはノード y の TID と同じ）を element 属性として与える。

５．copy(x,y)

operation 属性として”copy”を与え、ノード x の TID を from 属性、ノード x のコピー先での TID を to 属性として与える。

６．move(x,y)

operation 属性として”move”を与え、ノード x の TID を from 属性、ノード x のコピー先での TID を to 属性として与える。

７．import

これは Archive Root の下に新しく文書木をインポートしてきた際の特別な処理である。operation 属性として”import”を与え、新しくインポートしてきた文書木のルートの TID を document 属性として与える。

以上のように Delta ノードが差分木に蓄積されていき、これらの持つ情報を利用して、XVerPath によるノード選択が行われる。

５．文書木内のノードに付与される属性

アーカイブ木内にインポートされた文書木はすべて、その各ノードに次の３種類の属性が付与される。

① TID（Tree ID）属性

各ノードの、文書木内での位置を示した識別子である。アーカイブ木内には複数の文書木がインポートされるが、各文書木のルートノードに対して TID として自然数値を割り当てる。例えば、最

初にインポートされた文書木のルートには”1”を、次にインポートされた文書木のルートには”2”を、といった具合である。ルート以下のノードについてはDeweyオーダ[3] によるTID割り当てを行う。新しくノードを挿入する際に頑健となるように、ORDPATH [4]の手法を採用してもよい。ただし、更新操作によって同じ TID を持つノードが現れる（TID の重複）こともある。

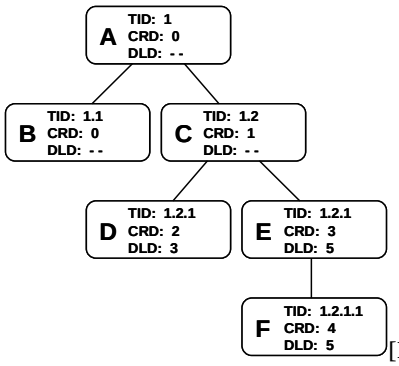
② CRD（CReated Delta）属性

各ノードがどの更新操作で作成されたのかを、各更新操作と一対一で対応するDeltaノードのnumber属性で表現したものである。CRD属性はノードが作成されたときに付与されるため、すべてのノードがこれを持ち、また一度付与されると変更されることはない。Deltaノードのnumber属性はtime属性と対応しているため、CRD 属性によって各ノードがいつ作成されたのかを表現することができる。

③ DLD（DeLeted Delta）属性

ノードを削除するような操作が行われた際に、実際にアーカイブ木からノードを削除してしまうのではなく、この属性を付与することによって削除されたことを表現する。ノードがどの更新操作で削除されたのかを、各更新操作と一対一で対応する Delta ノードの number 属性で表現したものである。Delta ノードの number 属性は time 属性と対応しているため、DLD 属性によって各ノードがいつ削除されたのかを表現することができる。また削除されていないノードは、この属性が付与されていないため、この属性を持たないノードは、現時点で削除されていないことを表す。

以上の属性のうち、DLD 属性を除く２種類の属性は、文書木内のノードすべてに付与される。また、TID 属性は更新操作によって重複することもあるが、一回の更新操作で TID の重複するノードが作成されることはないため、TID 属性と CRD 属性の組み合わせが重複するノードは存在しない。この性質により、与えられたタイムスタンプ t におけるある文書木のスナップショットを生成するためには、文書木のノードのうち(1) CRD 属性の指す Delta ノードの time 属性が t 以下でありかつ (2) DLD 属性を持たないか DLD 属性の指す Delta ノードの time 属性が t 以降であるもの選択するればよい。これらの属性を付与した文書木の例を図２に示す。



[M. I1]

図２　文書木の例

文書木がアーカイブ木にインポートされると、更新操作に対する処理を行うために、文書木に TID や CRD を付与して、更新処理が可能な状態にしなければならない。すなわち、Dewey オーダもしくは ORDPATH によって TID を割り当て、インポート時の Delta ノードの number 属性を CRD として各ノードに付与する。こうして更新処理の準備が整った文書木に対して、次のような更新処理を行う。

１．delete(x)

ノード x、もしくはノード x 以下の部分木を削除する更新操作である。しかし実際には文書木から削除してしまわずに、DLD 属性を付与して削除したことを表現する。図３にこの操作の具体例を示した。ノード C はノード D とともに部分木を構成しているため、そのルートであるノード C に DLD 属性が付与されるのと同時に、ノード D にも DLD 属性が付与される。

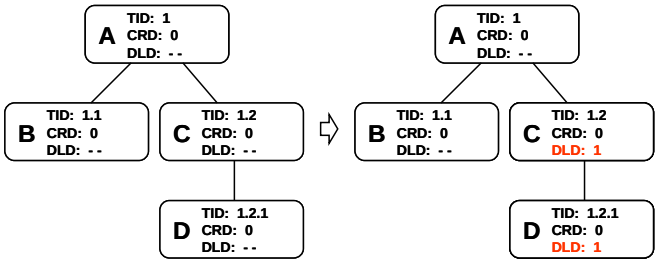


図３　delete(C)の実行

２．insert(x,y)

ノード y をルートとする新たな部分木を、ノード x の子として挿入する更新操作である。新しく生成されるノードには TID と CRD を付与する。delete 操作においてノードが実際に削除されるわけではないため、すでに delete されたノードと TID が重複する場合がある点に注意しなければならない[M. I2]。図４にこの操作の具体例を示した。すでにノード B に DLD 属性が付与されているところにノード C を挿入している。このため TID 属性に重複が起こるが、ノード B の DLD 属性とノード C の CRD 属性は異なっており、これによって update 操作や replace 操作との区別が可能となる。

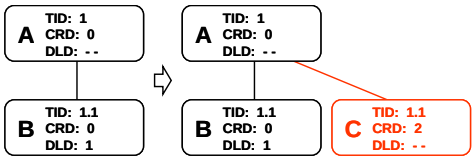


図４　　insert(A,C)の実行

３．update(x,c)

ノード x の内容を c に置き換える更新操作である。しかし実際に文書木内の指定ノードの内容を書き換えるのではなく、新たなノードを生成して内容を書き込み、古いノードは残したまま、新しいノードを文書木に挿入することとする。古いノードには DLD が付与され、新しいノードには CRD が付与されるが、新しいノードの TID は古いノードの TID と同じものを割り当てる（TID を重複させる）。また古いノードが子ノードを持っていた場合は、それらを新しいノードの子ノードに移し替える。図５にこの操作の具体例を示した。ノード B はノード C を子として持っているので、ノード B が update されるとノード C は新しく生成されたノード B'の子となる。また、ノード B とノード B'の TID は重複しており、さらにノード B の DLD とノード B'の CRD が一致している。

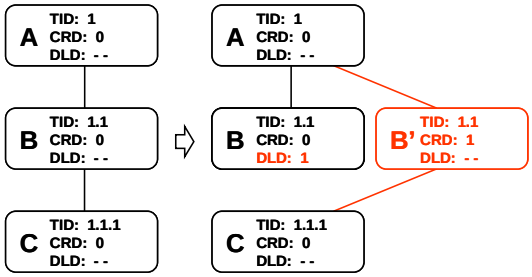


図５　　update(B,c)の実行

４．replace(x,y)

ノード x 以下の部分木を削除して、そこに新しくノード y 以下の部分木を挿入する、つまり x 以下の部分木を y 以下の部分木で置換する更新操作である。しかし実際に部分木を置換するのではなく、x 以下の部分木に DLD を付与し、y 以下の部分木に CRD を付与して、y の TID を x の TID と同じものを割り当てるように、y 以下の部分木の各ノードの TID を決定する（TID を重複させる）。図６にこの操作の具体例を示した。ノード B はノード C と部分木を構成しているため、ノード B とともにノード C にも DLD 属性が付与される。ノード D はノード E を子に持ったものとして置換されているので、ノード D の TID はノード B の TID と重複するように付与され、ノード E の TID はノード D の TID に基づいて付与される。またノード B の DLD とノード D の CRD が一致している。

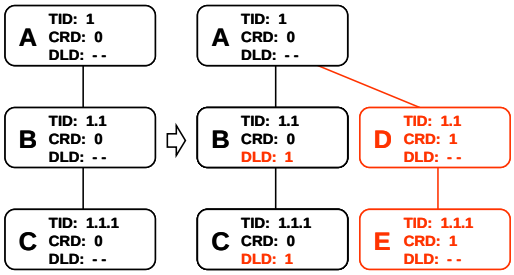


図６　　replace(B,D)の実行

５．copy(x,y)

ノード x 以下の部分木をコピーした部分木を、ノード y の子として挿入する更新操作である。コピーされた部分木には改めて CRD を付与する。図７にこの操作の具体例を示した。ノード C を複製したノード C'を挿入するのと同様であるが、差分木にノード C とノード C'間に no-change 枝があることが記録される。

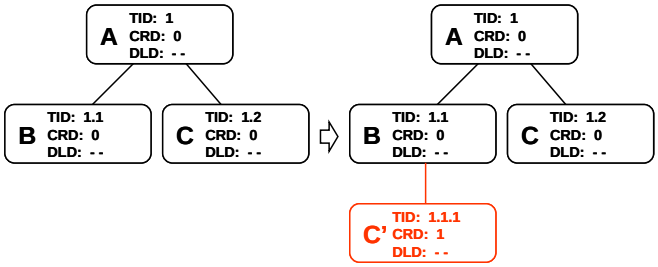


図７　　copy(C,B)の実行

６．move(x,y)

ノード x 以下の部分木を、ノード y の子と移動させる更新操作である。実際には、移動させるというよりも、copy(x,y)の後 delete(x)を実行するのと同じような処理を行う。図８にこの操作の具体例を示した。図７とほぼ同じであるが、ノード C に DLD 属性が付与されている点で異なる。

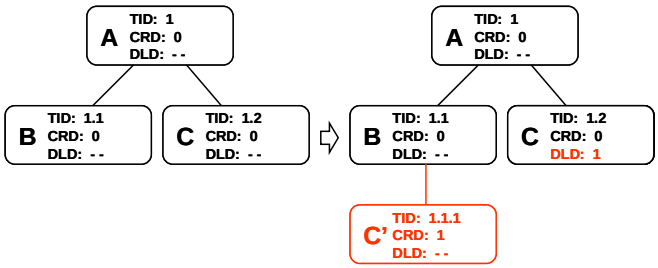


図８　　move(C,B)の実行

６．バージョン枝の判定方法

no-change 枝、updated 枝、replaced 枝の３種のバージョン枝について、それらによるノード間の親子関係の有無を判定する方法を述べる。文書木内のあるノード a が与えられたときの、次の各関数が返すノードを述べる。条件を満たすノードがない場合は、それぞれの関数値に該当するノードが存在しないこととなる。ここで a.vpar(l), a.vchild(l)はそれぞれ、ノード a の（文書木での親でなく）バージョンの親およびバージョンとしての子を求める関数であり、l は枝ラベ

ルの集合であり n (no-change), u (updated), r (replaced)のいずれかからなる。

1．a.vpar(n)

a の TID を T1、a の CRD を C とする。Delta Root ノード以下の Delta ノードのうち、number 属性値が C である Delta ノードを求め、これを d とする。ノード d の from 属性値を S1、to 属性値を S2 とすると、必ず T1 が S2 を接頭辞として持っているので、この S2 の部分を S1 で置き換えたものを T2 とする。文書木から、T2 を TID に持ち、CRD が C より小さく、DLD が C より大きいノードを選択し、関数の返り値とする。

2．a.vchild(n)

a の TID を T1、a の CRD を C、a の DLD を D とする。Delta Root ノード以下の Delta ノードのうち、number 属性値が C より大きく D 以下であり、さらに operation 属性値が”copy”もしくは”move”である複数の Delta ノードを求め、これを d1、d2、d3、…、dn（n は該当する Delta ノードの個数）とする。k を自然数（1≦k≦n）として、ノード dk の number 属性値を Nk、from 属性値を S1k、to 属性値を S2k とすると、T1k が S1k を接頭辞として持っていれば、この S1k の部分を S2k で置き換えたものを T2k とする。文書木から、T2k を TID に持ち、CRD が Nk のノードを選択し、これを ek とする。1≦k≦n なるすべての k について ek を探索してみて、これらを関数の返り値とする。

3．a.vpar(u)

a の TID を T1、a の CRD を C とする。Delta Root ノード以下の Delta ノードのうち、number 属性値が C、operation 属性値が”update”、element 属性値が T1 である Delta ノードが存在するか調べる。これが存在していて、a の真上のノード（すぐ上の兄ノード）b について、b の TID が T1、b の DLD が C であるならば、b を関数の返り値とする。

4．a.vchild(u)

a の TID を T1、a の DLD を D とする。Delta Root ノード以下の Delta ノードのうち、number 属性値が D、operation 属性値が”update”、element 属性値が T1 である Delta ノードが存在するか調べる。これが存在していて、a の真下のノード（すぐ下の弟ノード）b について、b の TID が T1、b の CRD が D であるならば、b を関数の返り値とする。

5．a.vpar(r)

a の TID を T1、a の CRD を C とする。Delta Root ノード以下の Delta ノードのうち、number 属性値が C、operation 属性値が”replace”、element 属性値が T1 である Delta ノードが存在するか調べる。これが存在していて、a の真上のノード（すぐ上の兄ノード）b について、b の TID が T1、b の DLD が C であるならば、b を関数の返り値とする。

6．a.vchild(r)

a の TID を T1、a の DLD を D とする。Delta Root ノード以下の Delta ノードのうち、number 属性値が D、operation 属性値が”replace”、element 属性値が T1 である Delta ノードが存在するか調べる。これが存在していて、a の真下のノード（すぐ下の弟ノード）b について、b の TID が T1、b の CRD が D であるならば、b を関数の返り値とする。

図９に、アーカイブ木の例を示した。この図のアーカイブ木はすでにいくつかの更新処理がなされ、それに従って各属性の付与と、差分木の Delta ノードの処理が施されている。なお、この例ではアーカイブ木にインポートされている文書木は一つであり、従って一つの XML 文書データに対して更新操作を行った例である。

上記のような判定の条件に従って、実際にこのアーカイブ木からバージョン枝を識別してみる。まずノード E に着目すると、B = E.vpar(n)、すなわちノード E はノード B を no-change 枝による親ノードとして持つことがわかる。そして同時に、E ⊂ B.vchild(n) も成り立っている。ノード F に着目すると、E = F.vpar(u)、すなわちノード F はノード E を updated 枝による親ノードとして持つことがわかる。逆に、F = E.vchild(u) も成り立つ。さらにノード D について、C = D.vpar(r)、すなわちノード D はノード C を replaced 枝による親ノードとして持つことがわかる。逆に、D = C.vchild(r) も成り立つ。これらと同様に、B ⊂ F.vanc(n,u)なども成り立つ。

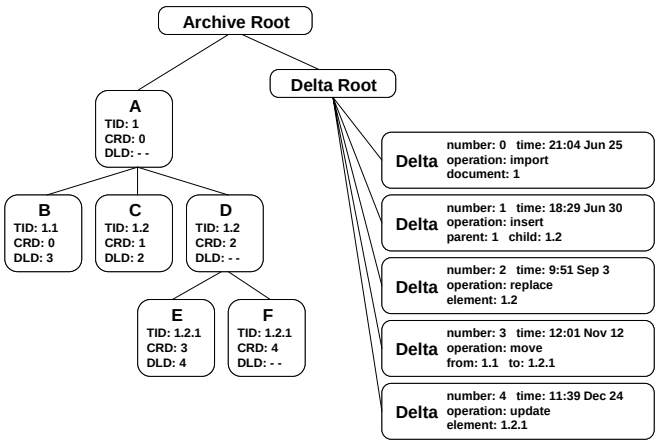


図９　処理後のアーカイブ木

なお、図９の文書木において最新のデータとして存在しているノードは、DLD 属性が付与されていない A, D, F の３つである。また、時刻として”22:13 Sep 3”（９月３日の午後１０時１３分）を指定した場合、その時刻におけるスナップショットに存在するノードは、A, B, D の３つとなる。このように、XVerPath によるタイムスタンプに基づくノード指定に対して選択を行う。

７．実験

本稿で提案する手法に従って、アーカイブ木を Java プログラムで実装し、文献[1]で述べられている要素バージョンモデルをそのまま実装した場合との全体の記憶容量の比較や、XVerPath による解析速度の測定を行った。なお、要素バージョンモデルの実装は同じように Java プログラムで行い、更新処理のたびに新しい文書木を生成し、バージョン枝としてバージョン親とバージョン子に互いに情報を属性として持たせる仕組みで実装した。

１．環境

今回の実験では、本稿で提案しているアーカイブ木と、文献[1]で提案されている要素バージョングラフとを、どちらも Java プログラムとして実装したものを用いた。Java プログラムの開発環境は、表 **1** の通りである。

Java のバージョン	J2SE v1.4.2_09 SDK
開発環境	Eclipse SDK v3.1.1
使用ライブラリ	java.io.* java.util.* java.text.* javax.xml.parsers.* javax.xml.transform.* org.w3c.dom.* org.xml.sax.*

表 1　Java プログラムの開発環境

なお、Java のプラットフォーム J2SE と基本的なライブラリは、Sun Microsystems のサイト（http://www.sun.com/）よりダウンロードし、Java プログラム開発ソフト Eclipse は、Eclipse のサイト（http://www.eclipse.org/）よりダウンロードし、表 **1** の使用ライブラリのうち下の４つのライブラリは Apache XML Project のサイト（http://xml.apache.org/）より Apache Xalan をダウンロードした。

また、サンプルとして使用する XML 文書であるが、XMark のサイト（http://monetodb.cwi.nl/xml/）にある xmlgen という XML 文書作成プログラムを用いて生成したものである。このソフトは、XMark という XML Benchmark Project で作成され、指定したサイズの、一般的に用いられるものに近い内容を持つ XML 文書ファイルを生成することができる、Ｃプログラムである。

２．記憶容量

約１ＭＢの容量の XML 文書サンプルを用意し、これを本稿で提案したアーカイブ木として構築して更新していった場合と、文献[1]の要素バージョングラフをそのまま実装して更新していった場合とで、全体の記憶容量を比較したものが、表 **2** である。なお、表 **2** で比較している各 XML 文書ファイルは、次のような工程で処理を施したものである。

① 通常用いられるような約 1.2ＭＢの一般的な XML 文書ファイルが doc である。

② doc をそれぞれの保存形式に加工して、doc0 を生成した。

③ doc0 に対して、delete 操作を施して doc1 を生成した。

④ doc1 に対して、insert 操作を施して doc2 を生成した。

⑤ doc2 に対して、update 操作を施して doc3 を生成した。

⑥ doc3 に対して、replace 操作を施して doc4 を生成した。

⑦ doc4 に対して、copy 操作を施して doc5 を生成した。

⑧ doc5 に対して、move 操作を施して doc6 を生成した。

ファイル	アーカイブ木		要素バージョングラフ	
	ノード数（個）	サイズ（Byte）	ノード数（個）	サイズ（Byte）
doc	17,131	1,167,360	17,131	1,167,360
doc0	17,134	1,617,920	17,132	1,482,751
doc1	17,135	1,618,015	34,262	2,957,312
doc2	17,137	1,618,433	51,393	4,440,064
doc3	17,139	1,618,689	68,524	5,914,624
doc4	17,141	1,619,116	85,655	7,389,184
doc5	17,150	1,620,623	102,794	8,867,840
doc6	17,156	1,622,016	119,933	10,342,400

表 2　記憶容量の比較

表 **2** から分かる通り、要素バージョングラフの場合は更新操作の度にその記憶容量が更新回数に比例して大きくなっていくのに対し、アーカイブ木の更新操作に対する容量の増加は微小である。これは、要素バージョングラフの場合は更新操作によって変更が加えられない部分についても新しい文書木・ノードが生成されていくのに対し、アーカイブ木は差分についての情報のみを蓄積していくため変更の加えられていない部分についての冗長な情報が省かれ、圧縮されているためといえる。したがって、要素バージョングラフの場合に比べ、アーカイブ木は圧倒的に記憶効率がよいといえる。

３．解析速度

本稿で提案しているアーカイブ木と、文献[1]で提案されている要素バージョングラフをそのまま実装したものとで、XVerPath による解析速度を比較するため、記憶容量の比較のときに作成した文書 doc6 に、さらに次の加工を行ってサンプル文書を作成し、次のような XVerPath の関数を実行した。

① doc6 のあるノード A を２回 update して、B とする。

② ノード B を１回 update して、C とする。

③ ノード C を２回 replace して、D とする。

④ ノード D を１回 copy して、複製先のノードを E とする。

⑤ doc6 内の、更新操作対象に一度もならなかったあるノードを F とする。

これらに、次のような XVerPath 関数を実行して、結果を得るまでの時間を測定したのが、である。なお、結果として示した数値は 10 回の実験を行った測定値の平均値である。

XVerPath 関数	アーカイブ木	要素バージョングラフ
E.vpar(n)	0.4	0.1
D.vchild(n,r)	1.0	0.5
D.vanc(r)	0.6	0.3
B.vcon(n,u,r)	4.0	15.3
F.vcon(n)	0.2	2.7

表 3　計算時間の比較（単位は msec）

表 **3** を見ると、要素バージョングラフの方が処理速度の速い関数と、アーカイブ木の方が処理速度の速い関数とに分かれる。表 **3** における上３行の関数の結果を見ると、単純なバージョンの親子検索には要素バージョングラフの方が向いているといえる。なぜならば、要素バージョングラフは各ノードに、そのバージョン親または子へのバージョン枝の情報が直接連結されているため、バージョン親または子を短時間で求めることができるからであると考えられる。

一方、表 **3** の下２行の関数では、アーカイブ木の方が圧倒的に処理速度が速い。これは、関数に no-change 枝による先祖・子孫バージョンを求めるものが含まれているかどうかで、処理速度に差が出ることを示していると考えられる。アーカイブ木はノードに何の更新処理もなされない場合はそのノードが一つのままであるのに対し、要素バージョングラフの場合は何の更新処理もなされないノードも次の新しいバージョンの木へと複製されるため、no-change 枝による先祖・子孫バージョンによるノード系列が相当数存在する。表 **3** の下２行の関数をアーカイブ木の方が圧倒的に速く処理できるのは、更新処理のなされないノードについては複製を生成しないこうした性質によるものと考えられる。

単純な親子バージョンを求める場合でも、アーカイブ木の処理速度は要素バージョングラフの処理速度に比べて大きく劣るということもないので、実用の面から見ても、全体的にアーカイブ木の方が XVerPath の解析処理速度が速いといえる。

さらに、アーカイブ木において解析時にたどる枝の数と計算時間の関係についても実験で測定した。この結果を示したものが図１０である。この図１０より、バージョン枝の解析における計算時間は、枝の数に比例して大きくなることが分かる。

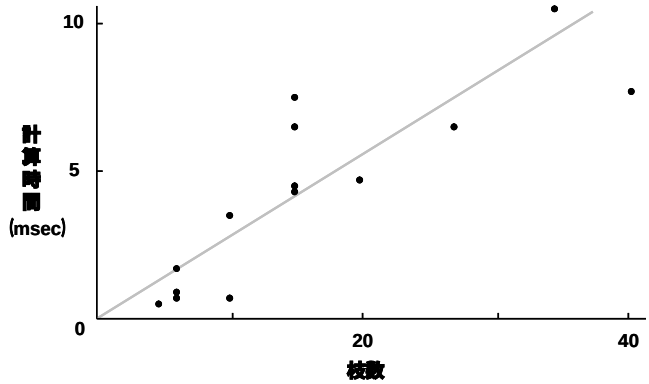


図１０　　たどるバージョン枝数と計算時間の関係

８．まとめ

本稿では更新され生成されていく XML 文書データを、更新操作の差分によって圧縮した、記憶効率の良い手法を提案した。本手法は、ノードレベルでバージョンの依存関係を求めることができるという特徴があり、これにより XML 文書アーカイブに対する派生関係や生成時間によるアクセス制御を行なうことが可能になる。また本手法で蓄積されたデータに対し、付与された種々の属性値や差分木を参照することによって、XVerPath でのバージョン枝の判別や、タイムスタンプによる検索への対応を述べた。最後に、実験によって、提案手法が記憶効率・解析速度の両面で優れていることを確かめた。

参考文献

[1] M. Iwaihara, S. Chatvichienchai, C. Anutariya, and V. Wuwongse. Relevancy Based Access Control of Versioned XML Documents., Proc. 10th ACM Symposium on Access Control Models and Technologies (SACMAT), Stockholm, pp. 85-94, June 2005.

[2] S. Chatvichienchai, C. Anutariya, M. Iwaihara, V. Wuwongse, and Y. Kambayashi, Towards Integration of XML Document Access and Version Control, Proc. 15th International Conference on Database and Expert Systems Applications - DEXA 2004, LNCS3180, pp. 791-800, Sep. 2004..

[3] I. Tatarinov, S. Viglas, K.S. Beyer, J.Shanmugasundaram, E.J. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. SIGMOD* 2002, pp. 204-215, 2002

[4] P.E.O' Neil, E.J.O' Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. Ordpaths : Insert-friendly xml node labels. In *Proc. SIGMOD Conf. 2004*, pp. 903-908, 2004.