社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2007－DBS－142 (12)
2007－ＦＩ－ 87 (12)
2007／5／31

リレーショナルデータベースにマッピングされた XML ドキュメント XMark に対する XPath 問い合わせ実験の試み

金子 邦彦† 谷 論‡

†九州大学大学院システム情報科学研究院知能システム学部門
‡九州大学大学院システム情報科学研究院知能システム学専攻

概要

データベースによる XML 文書の管理には，ネイティブ XML データベースを用いた方法とリレーショナルデータベースへのマッピングによる方法の2つがある．前者は XML 文書をそのままの形でデータベース化し，後者はリレーショナルデータベースに XML 文書をマッピングする．本稿は，ネイティブ XML データベースとリレーショナルデータベースの両方で，55MB, 111MB, 222MB の 3 種類の XMark データベースを作り，簡単な XPath を実行した試みの報告である．リレーショナルデータベースシステムは PostgreSQL（バージョン 8.2.1）を用いた．リレーショナルデータベースのマッピングでは前順後順法を用いるとともに，各ノードに 5 つの数値属性を持たせる．ネイティブ XML データベースシステム Berkeley DB XML （バージョン 2.3.8）による実験も行った．

Experimental Tests of XPath Expressions for a XML document XMark mapped onto a Relational Database

Kunihiko Kaneko and Satoshi Tani

Graduate School of Information Science and Electrical Engineering, Kyushu University

Abstract

There are two major methods for managing XML documents using databases. They are native XML database systems and relational database systems. The former manages XML documents as they are. In the later systems, the XML documents are mapped to a relational database. After the XML documents are mapped into relational data, XPath expressions are translated into SQL expressions. We performed some experimental tests to evaluate some XPath expressions on the above two. To map XML documents onto a relational database, we use two numbers. They are *preorder* and *postorder*. In addition to this, we used five numerical attributes for the nodes in XML documents. In this paper, we report the test results of the execution time and disk I/O cost using a relational database system PostgreSQL v 8.2.1 and the native XML database system named Berkeley DB XML v2.3.8.

1 はじめに

我々が行っている研究の最終的な目標は，Well-Formed な形式の XML ドキュメントに対する XPath 式及び XPath 式を含む XQuery 式の処理において，各種のインデックスを使い，高速処理を行うことである．こうした研究の重要性は，言うまでもないことであるが，筆者らが，興味を持ったきっかけの 1 つは，手持ちの XML ドキュメント(XML で記述された一種のアンケートや，各種の実験結果のデータなどで，途中でアンケート項目や実験項目の変更・追加があり得るため，前もって DTD を定めることが難しいかも知れないと考えている)をデータベース化し，各種の集計や分析を行えるシステム作りに興味を持ったためである．システム作りでは，既存のリレーショナルデータベースシステムのソフトウエアを使いたいと考えた．その理由は，PostgreSQL[7] など，リレーショナルデータベースシステムには，機能的に申し分無く，かつ，ソースコードが公開されているものが多く，今後の研究に便利だと判断したためである．PostgreSQL には，ユーザが比較的容易にインデックスを追加する機能があり，この点でも，我々の研究に便利であると考えた．

一方で，XML ドキュメントのファイルを直接扱う各種のプロセッサ・ソフトウエア(つまり，XML ドキュメントをデータベース化しない)や，XML ドキュメントを容易に扱えるネイティブ XML データベースシステム(その代わり，SQL 等，リレーショナルデータベースとしての機能は限定的であるか，あるいは無い)も，種々存在する．つまり，リレーショナルデータベースシステムを使うと決めてしまうのは早計かも知れない．そこで，予備的に，下記に示す実験を行ったので，本稿で報告する．

Well-Formed な形式の XML ドキュメントを扱うために，(1) リレーショナルスキーマの定義と，この上への XML ドキュメントのマッピング，(2) インデックスの生成，(3) XPath 式/XQuery 式 の SQL 文への変換が必要である．我々が行った実験の概要は下記の通りである．

実験では，リレーショナルデータベースシステム PostgreSQL(バージョン 8.2.1)を使う．このデータベースシステムに，XMark データベース[1,3]を作る．XMark データベースは，インターネットオークションをモデルとするデータであり，比較的大きなサイズの XML ドキュメントを扱う．XML ドキュメントのサイズは，XMark データベース生成プログラムで自由に設定できるが，今回は 55,111,222MB(メガバイト)の 3 種類のデータベースを作った．XMark データベース格納用のリレーショナルスキーマは，後述するような制限はあるが，XMark 以外の一般の Well-Formed な形式の XML ドキュメントを格納できるように定義した．インデックスとしては，PostgreSQL が備える B-tree インデックスと，hash インデックスを用いた．以上で，XMark データベースが PostgreSQL 上に出来上がる．実験では，XMark データベースで動く 3 種類の XPath と 2 種類の XQuery を(どれも簡単なもの)，手動で等価な SQL 文に変換し，実行する．但し，高速化のため，SQL 文には，PostgreSQL のみが備える SQL の機能も使うことにした．

一方で，参考実験として，同様の処理を，ネイティブ XML データベースである Berkeley DB XML(バージョン 2.3.8) と，ファイルベースの XPath プロセッサである XMLStarlet[10] (バージョン 1.0.1)とでも，一部実行し，対照した．但し，今回の報告では，PostgreSQL, Berkeley DB XML, XMLSatrlet の性能比較というには全く不十分であることは断っておく．その理由は，まず，実行した XPath 式，XQuery 式は，XPath, XQuery が持つ機能を網羅するものではない．また，データベース生成時間や，各種の更新処理の比較も行っていない．

本稿で示す実験の意義は次のようにまとめられる．今回の実験でのリレーショナルスキーマの定義及び SQL 文は，高速な処理結果が得られるように，実験結果を見ながら何度か変更を行って出来たものである．結果として，多くの場合，Berkeley DB XML, XMLStarlet に匹敵する性能になり，場合によっては，これらを凌駕する性能が得られた．そのため，最終的に実験で使ったリレーショナルスキーマの定義及び SQL 文を報告することに，一定の意義があるかも知れないと考えた．

本稿の構成は次の通りである．2 章では，XML ドキュメントのノードへの番号付けを中心に説明する．3 章は，我々が実験で用いたリレーショナルスキーマの定義の説明である．4 章は，実験で用いた SQL の説明である．5 章は，実験の結果の報告と考察である．

2 ノードの番号付けと数値属性

XMLドキュメントのツリー構造をリレーションにマッピングするために，XML ドキュメントのノード(node)に，preorder, postorder の 2 つの番号を付け，preorder と postorder の 2 軸からなる 2 次元の空間にマッピングする方法が広く行われている[2]．この方法を，本稿では，前順後順法と呼ぶ．本章では，まず，前順後順法を簡単に説明し，その後，我々のリレーショナルスキーマ定義において，前順後順法をどのように使っているかを説明する．

2．1 前順後順法

XML ドキュメントをトリー(tree)と見立てると，要素(element)，属性(attribute)，テキスト(text)，コメント(comment)，プロッセッシングインストラクション(processing instruction)がノードをなし，これらのノード間の親子関係がアーク(arc)をなす．前順後順法では，ノードに，preorder, postorder という 2 つの番号を与える．前順後順法を使い，XPath 式における 13 種類の軸に関する処理が効率よく行えるという報告[4]がすでに行われている．これは，2 つのノードが，ある位置関係(preceding, descendant, ancestor, following) にあるとき，この 2 つのノードの preorder, postorder に下記が成り立つという性質を使っている．

(a) preceding
$pre(v_1)<pre(v_2) \wedge post(v_1)<post(v_2)$
$\Leftrightarrow$ v_1 は v_2 の前(preceding)にあるノード

(b) descendant
$pre(v_1)>pre(v_2) \wedge post(v_1)<post(v_2)$
$\Leftrightarrow$ v_1 は v_2 の子孫(descendant)であるノード

(c) ancestor
$pre(v_1)<pre(v_2) \wedge post(v_1)>post(v_2)$
$\Leftrightarrow$ v_1 は v_2 の祖先(ancestor)であるノード

(d) following
$pre(v_1)>pre(v_2) \wedge post(v_1)>post(v_2)$
$\Leftrightarrow$ v_1 は v_2 の後(following)にあるノード

(ここでは，ノード v の preorder を $pre(v)$, postorder を $post(v)$ のように記す)．

2．2 我々が用いる前順後順法

我々は，前順後順法を使う．前順後順法は，ノード数を n としたときに，preorder, postorder ともに 1 から n のような n 個の連続した番号として実装されていることが多い．

```
<people>
  <person id="1">
    <name>X</name>
    <age>21</age>
  </person>
  <person id="2">
    <name>Y</name>
    <age>32</age>
  </person>
</people>
```

(a) XML ドキュメントの例

pre'値	post'値	タグ名，テキスト値，属性名，属性値
1	26	people
2	13	person
3	4	id, 1
5	8	name
6	7	X
9	12	age
10	11	21
14	25	person
15	16	id, 2
17	20	name
18	19	Y
21	24	age
22	23	32

(b) (a) の XML ドキュメントの preorder, postorder

図1．我々が使う前順降順法における preorder, postorder

それに対し，我々が用いる前順降順法では，1から $2n$ までのとびとびの値をとる．以下，本稿では，前者と後者を区別するために，適宜，前者(1 から n までの連続した番号)を $pre(v)$, $post(v)$，後者(1から $2n$ までのとびとびの番号)を $pre'(v)$, $post'(v)$ と書く．我々が使う $pre'(v)$, $post'(v)$ は，次のようなものである．

preorder
$pre'(v)$
$= pre(v) +$ v の前(preceding)にあるノード数

postorder
$post'(v)$
$= post(v) +$ 全ノード数
$-v$ の後ろ(following)にあるノード数

$pre'(v)$, $post'(v)$ の例を図1に示す．$pre'(v)$, $post'(v)$ は，元の $pre(v)$, $post(v)$ が持っている順序を崩さない(つまり，$pre(v_1)<pre(v_2)$ ならば $pre'(v_1)<pre'(v_2)$ であり $post(v_1)<post(v_2)$ ならば $post'(v_1)<post'(v_2)$ である)．一方で，上記のように定義された $pre'(v)$, $post'(v)$ は，次

の性質を持つ.

(a) preceding

$post'(v_1)<pre'(v_2)$

$\Leftrightarrow$ v_1 は v_2 の前(preceding)にあるノード

(b) descendant

$pre'(v_1)>pre'(v_2) \wedge pre'(v_1)<post'(v_2)$

$\Leftrightarrow$ v_1 は v_2 の子孫(descendant)であるノード

(c) ancestor

$pre'(v_1)<pre'(v_2) \wedge post'(v_1)>pre'(v_2)$

$\Leftrightarrow$ v_1 は v_2 の祖先(ancestor)であるノード

(d) following

$pre'(v_1)>post'(v_2)$

$\Leftrightarrow$ v_1 は v_2 の後(following)にあるノード

まず, preceding, following については, $pre(v)$, $post(v)$ では条件が2つあったのが, $pre'(v)$, $post'(v)$ では 1 つに減っている. つまり, 2次元の検索であったのが, 1次元になる. descendant についても, 2 次元であったのが, 1 次元になっている. つまり, ノード v_2 の descendant を求めることは, $pre'(v_2)$, $post'(v_2)$ という 2 つの値を使い, ($pre'(v_2)$, $post'(v_2)$)の範囲にある pre'値を持つようなノードの検索になる. これは, pre'値という単一属性についての範囲検索である.

以上のように, preceding, ancestor, following に関する軸検索は, 1 次元の範囲検索に帰着される.

2.3 データモデル

まず, 本稿で扱う XML ドキュメントの種類を説明しておく. 本稿で扱う XML ドキュメントは, 図2のように, テキストの途中でタグが現れるものは扱わない(検討は今後の課題である). また, 現在は, コメント(comment), プロッセッシングインストラクション(processing instruction)の 2 種のノードは無視している(これらを扱えるための検討も今後の課題である). 以上の制限はあるが, XMark 以外の一般の Well-Formed な形式の XML ドキュメントを扱えるように, リレーショナルスキーマを設計している.

XML ドキュメントの各ノードは, pre'値と post'値を持つ. DOM では, 同一要素が複数属性を持つ場合, 属性が登場する順序を区別しないことになっている. 一方, 我々の属性に対する番号付けは, 属性ノードに対する pre'値と post'値の付与の処理などを単純化するために, 次のように行う. (1) 属性については, 各属性ごとに1個の属性ノードがあると考える. (2) 同一要素の複数属性に対する pre'値と post'値は, XML ドキュメント中の属性の登場順で違う値を付ける.

```
<X>
  te<Y>x</Y>t
</X>
```

図2. 上記のように, テキストの途中にタグが現れるものは, 本稿で扱わない.

表1. ノードに付与される番号と数値属性

		内容
番号	*pre'*	ノードの preorder
	post'	ノードの postorder
数値属性	*parent*	ノードの親が持つ *pre'*値
	firstch	ノードの最初の子が持つ *pre'*値
	lastch	ノードの最後の子が持つ *pre'*値
	psib	ノードの一つ前の兄弟(sibling)が持つ *pre'*値
	nsib	ノードの次の兄弟(sibling)が持つ *pre'*値

その他, pre'値と post'値以外にも, いくつかの数値属性をノードに与えておくと, 処理のときに便利であると考えて(その確認は, 今後の課題として残っている), DTM (Document Table Model)を参考にして, 5 つの数値属性を定めた. DTM は, Xalan プロジェクト[9]における XPath や XSLT プロセッサの実装等に用いられているデータモデルである. DTM では, 各ノードに対して, 親ノードと, 最初の子供ノードと, 一つ前の兄弟ノードと, 次の兄弟ノードに関する情報を持っている. これに習うとともに, 最後の子供ノードの pre'値も持たせる. 以上のことをまとめたのが, 表 1 である.

3. データベース定義

本研究では, XML ドキュメントは, リレーショナルデータベースシステム PostgreSQL で定義された複数のリレーションにマッピングされる. 本章では, このリレーションの定義について説明する.

データベースは, enode, anode, tnode の 3 つのリレーションと, インデックスからなる.

enode, anode, tnode は，それぞれ，要素ノード，属性ノード，テキストノードに対応する．つまり，ノードの種類ごとに分けて格納する．リレーションは，ノードの番号(pre'値とpost'値)の他，2 章で説明した 5 つの数値属性，タグ名，テキスト値，属性名，属性値を格納するための属性を持つ．リレーションの定義を行うSQL 文は図3の通りである．要点は次の通りである．

・全てのリレーションは，INT 型の属性 pre, post を持つ．これらには，2 章で説明したpre'値とpost'値を格納する．

・全てのリレーションは，INT 型の属性 parent, firstch, lastch, psib, nsib を持つ．これらには，2 章で説明した5 つの数値属性を格納する．

・リレーション enode の属性 name には要素のタグ名を格納する．

・リレーション anode の属性 name には属性名を格納し，属性 value には属性値を格納する．

・リレーション tnode の属性 value にはテキスト値を格納する．

5 つの数値属性及び value は，空値(NULL)をとり得る．例えば，ルートノードの parent，リーフノードの firstch, lastch などは空値を持つ．

テーブルへのマッピングは，XML ドキュメントを構文解析し，2 章で説明した番号付けを行い，ノードの種類ごとに，enode, anode, tnode の 3 つのリレーションに格納することで行う．以上のように，XML のトリー構造のアークやパス(path)を格納するためのリレーションは設けていない．

属性 pre には，範囲検索が多用されるので，B-tree インデックスを作る．一方，要素のタグ名を格納するリレーション enode の属性 name は，完全一致による検索が多用されるので hash インデックスを作る(図4を参照)．

4. XPath から SQL への変換

実験は，5 章で説明するように，XMark データベースで動く簡単な XPath 式と XQuery 式について，人間の判断によって，等価な SQL 文を書き，実行することで行った．本章では，SQL 文への変換について説明する．SQL 文は，PostgreSQL に固有の機能を使っている箇所がある．

表2．データベースを構成するリレーション

リレーション名	内容
enode	要素ノード
anode	属性ノード
tnode	テキストノード

```
create table enode(pre INT, post INT,
parent INT, firstch INT, lastch INT,
psib INT, nsib INT, name VARCHAR);

create table anode(pre INT, post INT,
parent INT, firstch INT, lastch INT,
psib INT, nsib INT, name VARCHAR,
value VARCHAR);

create table tnode(pre INT, post INT,
parent INT, firstch INT, lastch INT,
psib INT, nsib INT, value VARCHAR);
```

図3．データベース定義を行うSQL 文

```
create index idx_enode_pre on enode
(pre);

create index idx_enode_name on enode
using hash (name);

create index idx_anode_pre on anode
(pre);

create index idx_tnode_pre on tnode
(pre);
```

図4．インデックス生成を行う create index 文
(Postgre SQL 用)

4.1 要素名による絞り込み

XPath 式や XQuery 式では，要素のタグ名による絞り込みが多用される．例えば「/descendant::age」という XPath 式では，要素名 age によるノードの絞り込みが行われる．これを SQL 文に変換すると，リレーション enode についての，属性 name による選択(selection)になる．「/descendant::age」を例にとると，「SELECT * FROM enode WHERE name = 'age';」になる．

4.2 軸検索と動的なインデックス生成

XPath 式や XQuery 式の軸検索は，SQL では，リレーションが持つ属性 pre による結合(join)になる．以下，descendant 軸による軸検索である「/descendant::open_auction/descendant::description」を例にとり説明する．

最初は，要素による絞り込みである．「/descendant::open_auction/de

scendant::description」では，要素による絞り込みが 2 か所ある．そこで，4.1 章で説明した方法で，要素名が open_auction である要素集合と，要素名が description である要素集合を得る．それぞれの集合を，ここでは，table1，table2 と書く．これらはタプル集合（つまりリレーション）である．

要素による絞り込みの次が軸検索である．「/descendant::description」とあるため，descendant 軸による軸検索である．2 章で説明した通り，v_1 が v_2 の子孫 (descendant) であるとき，$pre'(v_1)>pre'(v_2) \wedge pre'(v_1)<post'(v_2)$ である．この条件で，(1) で得られた 2 つのリレーション table1，table2 を結合する．SQL 文は「SELECT table2.* FROM table1, table2 WHERE table2.pre BETWEEN table1.pre + 1 AND table1.post - 1;」のようになる．

軸検索における結合処理を高速化するために，結合処理の前に，リレーション table2 にインデックスを作る．そのために，PostgreSQL の機能である create temporary table 文を使用し，SQL 文「CREATE TEMPORARY TABLE table2 AS SELECT * FROM enode WHERE name = 'description';」を発行する．これで，一時的なリレーション table2 が，データベース中に出来る．その後，リレーション table2 の属性 pre で，B-tree インデックスを作る．そのために，PostgreSQL の create index 文を使い，「CREATE INDEX table2_pre ON table2 (pre);」を実行する．このインデックス生成は，SQL 実行時に動的に生成されるインデックスである．

結果として得られる SQL 文は，付録の図 7(a)に示している．

4.3 結果値の生成

4.2 章までの処理で，条件を満足するノード集合が得られるが，XPath 式や XQuery 式と等価の結果を得るには，結果値を生成するための処理が必要である．XPath 式の出力は，XPath 式にマッチングする部分をすべて，XML ドキュメントの形で出力するため，条件を満足するノードだけでなく，子孫のノードについても出力する．こうした結果値の生成は，SQL で簡単に書ける（付録の図 7(d)）．XQuery の場合は，XPath のように一般化できず，個別の XQuery ごとに結果値生成用の SQL を書く．

4 実験結果

実験では，リレーショナルデータベースシステム PostgreSQL と，ネイティブ XML データベースシステム Berkeley DB XML と，ファイルベースの XPath/XSLT プロセッサ XMLStarlet を使った．実験環境は表 3 にまとめている．

実験は，文献[5]などと同様に，XMark を用いた．XMark データベースの生成には，XMark が配布[8]する XMark データベース生成プログラム win32.exe を使用した．/f オプションで factor 値を設定することで，生成される XML ドキュメントのファイルサイズを変えることができる．サイズは，factor 値の倍数であり，factor 値が 1 のとき，111MB バイトになる．今回の実験では，55MB, 111MB, 222MB の 3 種類のデータベースを作った．

本実験で用いた 3 つの XPath 式は，文献[3]中の問い合わせ式 Q1 から Q3 を用いた．本実験での 2 つの XQuery 式は，XMark で定められている Query3 と Query5 を参考にした．付録の図 7 と図 8 に，実験で用いた XPath 式と XQuery 式を，PostgreSQL で実行した SQL 文とともに示している．実験では，これら，3 つの XPath 式と 2 つの XQuery 式を実行し，実行時間(elapsed time)及びディスク I/O の測定を行った．実験方法は下記の通りである．

・全ての実験において，測定前にコンピュータそのものを再起動した．つまり，測定結果はすべて cold（最初は，データベースバッファ等が空）である．

・BerkeleyDB XML では，データベース作成の際にノード単位でデータを分割するか

表 3. 実験環境

型番	DELL DIMENSION8400
CPU	Intel(R) Pentium(R) 4 3.2GHz, 1MB L2 キャッシュ
OS	Linux 2.6.9-42
メモリ	2GB
ソフトウエア	PostgreSQL v8.2.1 Berkeley DB XML v2.3.8 XMLStarlet v1.0.1

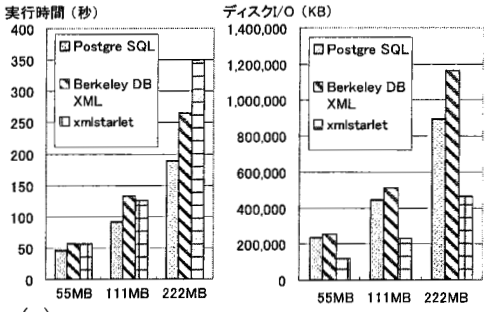
選択する機能がある．本実験ではデフォルトの設定である，ノード単位で分割しインデックスを持たないという設定で使用した．

・PostgreSQL では，データベース生成後，vaccumdb –a –z コマンドを実行した．

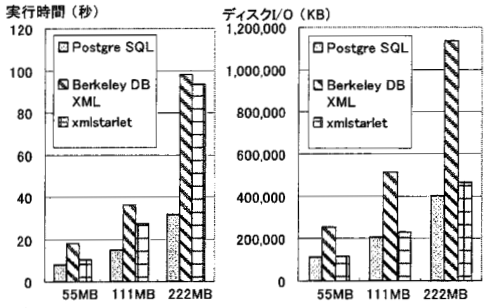
・ 問い合わせを実行した際に結果が出力され終わるまでの実行時間及び処理の際に行われたディスクI/Oの測定を行った．ディスクI/Oの測定には，iostatコマンドを用いた．

・XMLStarlet では，XQuery の実行は行わなかった．

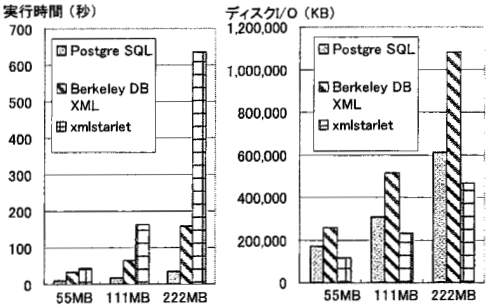
XPath式の実行結果を図5の(a)から(c)に，XQuery 式の実行結果を図6の(a)と(b)に示す．実験結果では，多くの場合（つまりXQuery#1 を除き）で，PostsgreSQL が最も良い性能を得られた．但し，Berkeley DB XML では特別な設定を行っていない（つまり，デフォルトの設定のまま）ため，性能の改善の見込みがある．つまり，PostsgreSQL の優位性を示すものでは無い．しかしながら，PostsgreSQL で，他と遜色ない性能が得られたことは満足だった．詳細な分析は，今度の課題であるが，2 つのことを指摘しておく．(1)Berkeley DB XML, PostgreSQL ともに，ディスク I/O 量がかなり多い（Berkeley DB XML の Query#2 のみが例外）．データベースのクラスタリングなどの新しい工夫を施し，再実験する必要性を示唆している．(2) PostsgreSQL の XQuery#1 は，あまりにも遅い．SQL の見直し，データベース設計の見直しが必要だと判断される．



(a) /descendant::open_auction/descendant::description



(b) /descendant::age/ancestor::person

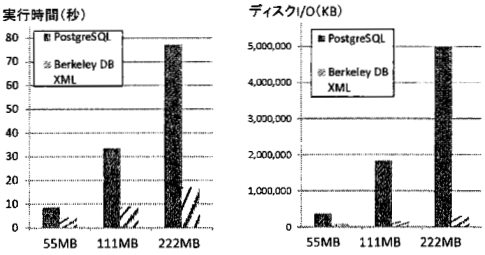


(c)/descendant::open_auction/child::privacy/preceding-sibling::bidder

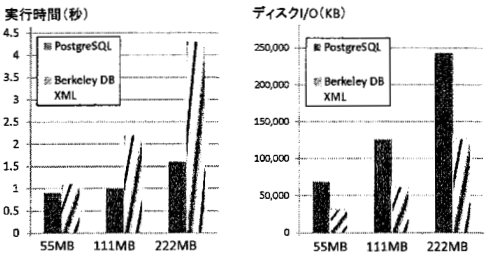
図５．３つの XPath 式の実行結果

5 おわりに

PostgreSQL を使い，XMark データベースを使った処理を行ったが，ネイティブXMLデータベースをデフォルトの設定で使う場合と比較して，遜色ない性能が得られる場合がある．この結論は，一般性を持つものでは無いが，限定的な局面では，リレーショナルデータベースでも十分な性能が得られるという可能性を示す．詳細な検討と裏付けは今度の課題である．



(a) XQuery #1（図8(a)を参照）の実行結果



(b) XQuery #2（図8(c)を参照）の実行結果

図６．２つの XQuery 式の実行結果

参考論文

[1] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, Ralph Busse, XMark: A Benchmark for XML Data Management, Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002.

[2] Jens Teubner, Torsten Grust, Maurice van Keulen, "Bridging the GAP Between Relational and Native XML Storage with Staircase Join." Grundlagen von Datenbanken, pp.85-89, 2003.

[3] Torsten Grust, Maurice van Keulen, Jens Teubner, Accelerating XPath evaluation in any RDBMS, ACM Trans. Database Syst. 29, pp.91-131, 2004.

[4] Torsten Grust, Accelerating XPath Location Steps, Proc. 21st ACM SIGMOD, pp. 109-120, 2002.

[5] 藤本 圭, 清水 敏之, 吉川 正俊 "文書スキーマに基づくノード識別子を用いた関係データベースへの XML 文書の効率的な変換" DEWS2004 7-A-04, 2004.

[6] http://monetdb.cwi.nl/xml/index.html

[7] http://www.postgresql.org/

[8] http://www.xml-benchmark.org/

[9] http://xalan.apache.org/index.html

[10] http://xmlstar.sourceforge.net/

付録

```
CREATE TEMPORARY TABLE table1 AS
SELECT * FROM enode WHERE name =
'open_auction';
CREATE TEMPORARY TABLE table2 AS
SELECT * FROM enode WHERE name =
'description';
CREATE INDEX table2_pre ON table2
(pre);
CREATE TEMPORARY TABLE result AS
SELECT table2.* FROM table1, table2
WHERE table2.pre BETWEEN table1.pre
+ 1 AND table1.post - 1;
```

(a) /descendant::open_auction/descendant::description の SQL 文(結果値の生成を除く)

```
CREATE TEMPORARY TABLE table1 AS
SELECT * FROM enode WHERE name =
'age';
CREATE TEMPORARY TABLE table2 AS
SELECT * FROM enode WHERE name =
'person';
CREATE INDEX table1_pre ON table1
(pre);
CREATE TEMPORARY TABLE result AS
SELECT table2.* FROM table1, table2
WHERE table1.pre BETWEEN table2.pre
+ 1 AND table2.post - 1;
```

(b) /descendant::age/ancestor::person の SQL 文(結果値の生成を除く)

```
CREATE TEMPORARY TABLE table1 AS
SELECT * FROM enode WHERE name =
'open_auction';
CREATE TEMPORARY TABLE table2 AS
SELECT * FROM enode WHERE name =
'privacy';
CREATE TEMPORARY TABLE result1 AS
SELECT table2.* FROM table1, table2
WHERE table1.pre = table2.parent;
CREATE TEMPORARY TABLE table3 AS
SELECT * FROM enode WHERE name =
'bidder';
CREATE TEMPORARY TABLE result AS
SELECT table3.* FROM result1, table3
WHERE table3.parent = result1.parent
AND table3.post < result1.pre;
```

(c) /descendant::open_auction/child::privacy/preceding-sibling::bidder の SQL 文(結果値の生成を除く)

```
SELECT enode.* FROM result, enode
WHERE enode.pre BETWEEN result.pre
AND result.post;
SELECT anode.* FROM result, anode
WHERE anode.pre BETWEEN result.pre
AND result.post;
SELECT tnode.* FROM result, tnode
WHERE tnode.pre BETWEEN result.pre
AND result.post;
```

(d) (a), (b), (c) に続いて実行される結果値の生成用 SQL 文

図 7．3 つの XPath 式と SQL

```
let $auction := doc("auction.xml")
return
for $b in
$auction/site/open_auctions/open_a
uction
where
zero-or-one($b/bidder[1]/increase/
text()) * 2 <=
$b/bidder[last()]/increase/text()
return
  <increase
  first="{$b/bidder[1]/increase/te
  xt()}"
  last="{$b/bidder[last()]/increas
  e/text()}"/>
```

(a) XQuery #1(XMarkQ3 を参考にした)

```
CREATE TEMPORARY TABLE table1 AS
SELECT * FROM enode WHERE name =
'open_auction';
CREATE TEMPORARY TABLE table2 AS
SELECT * FROM enode WHERE name =
'bidder';
CREATE TEMPORARY TABLE result1 AS
SELECT table2.* FROM table1, table2
WHERE table2.parent = table1.pre;
CREATE INDEX table3_parent ON table2
(parent);
CREATE TEMPORARY TABLE result2 AS
SELECT max(pre) AS max, min(pre) AS
min, parent AS key FROM result1 GROUP
BY parent;
CREATE TEMPORARY TABLE table3 AS
SELECT * FROM enode WHERE name =
'increase';
CREATE TEMPORARY TABLE result3 AS
SELECT max_t3.pre AS max, min_t3.pre
AS min, key FROM result2, table3 AS
max_t3, table3 AS min_t3 where
max_t3.parent = max AND
min_t3.parent = min;
CREATE TEMPORARY TABLE result AS
SELECT t1.value AS first, t2.value AS
last FROM result3, tnode AS t1, tnode
AS t2 WHERE t1.parent = result3.max
AND t2.parent = result3.min AND
to_number(t1.value,9999.99) *2 <=
to_number(t2.value,9999.99) ORDER
BY key;
SELECT * FROM result;
```

(b) (a) に示した XQuery#1 に対応する SQL 文

```
let $auction := doc("auction.xml")
return
count(
 for $i in
  $auction/site/closed_auctions/cl
  osed_auction
  where $i/price/text() >= 40
  return $i/price
  )
```

(c) XQuery #2(XMarkQ5 を参考にした)

```
CREATE TEMPORARY TABLE table1 AS
SELECT * FROM enode WHERE name =
'closed_auction';
CREATE TEMPORARY TABLE table2 AS
SELECT * FROM enode WHERE name =
'price';
CREATE INDEX table2_pre ON table2
(pre);
CREATE TEMPORARY TABLE result1 AS
SELECT table2.* FROM table1, table2
WHERE table2.pre BETWEEN table1.pre
+ 1 AND table1.post - 1;
CREATE TEMPORARY TABLE result AS
SELECT result1.* FROM result1, tnode
WHERE tnode.pre BETWEEN result1.pre
+1 AND result1.post -1 AND
to_number(tnode.value,9999.99) >
40;
SELECT count(*) FROM result;
```

(d) (c) に示した XQuery#1 に対応する SQL 文

図 8．3 つの XQuery 式と SQL