Vol.2012-DPS-150 No.23
Vol.2012-CSEC-56 No.23
2012/2/29

PostgreSQL 向け高速 XML データ型の提案

榎本俊文[†1]　鈴木源吾[†1]
小林伸幸[†1]　山室雅司[†1]

XML データへの操作は一般的には多種多様であるが, データベース上で必須かつ高頻度で行われる処理は, 経験的に, 属性値/テキストノードの個別の値の操作である. 特にインデックス作成時には特定の値の取得が全 XML データに対し行われる.

そこで, 個別の値操作を高速に行うことを主眼とした, 独自のバイナリ形式と操作手順を考案し, PostgreSQL 上に実装を行った. 性能測定結果も含め報告する.

Implementation of Yet Another XML-type for PostgreSQL

Toshifumi Enomoto,[†1] Gengo Suzuki,[†1]
Nobuyuki Kobayashi[†1] and Masashi Yamamuro[†1]

There are various manipulations for XML data, but most frequent and essential manipulations on XML database are getting and setting values of text nodes or attributes. In case of creating an index, specified values are extracted from all XML data.

Therefore, we propose the yet another XML-type implementation that enables to get and set values of text nodes or attributes quickly for PostrgreSQL.

1. はじめに

XML(eXtensible Markup Language) は, Web 系サービスにおけるデータ交換フォーマットとして盛んに利用されている. さらに近年, 金融分野, 流通分野等の法人系システムのデータ交換フォーマットとして, XML 標準規格の採用が進展するなど, その適用範囲は益々拡大している.

その一方で, データ格納フォーマットとしての XML, 即ち XML データベース (XMLDB) の採用事例は, リレーショナル・データベース (RDB) に比べると非常に少ない. その要因の一つは, XML データの利便性はあるものの, 一般的に RDB に比べ XMLDB の性能が低く, 総合的には優位性をもつに至っていないことが考えられる.

そこで本論文では, XMLDB の性能向上を目的に, 新しいデータ型を提案する. そして, PostgreSQL 上で実装・評価を行った結果を報告する.

2. 研究の背景

2.1 XML データベース

RDB は, 定型的なビジネス領域において大きな成功を収めた. しかし, その一方で, 非定型的で, データ構造が複雑で, 変更の多い応用分野も, 情報処理の進歩・インターネットの普及等に従って増えてきた. そのような領域に適した様々なデータベースが提案されたが, 現在その中で最も有望な選択肢は XMLDB と言えるであろう. XMLDB の特徴は,

- 多様なデータ構造 (スキーマレス含む) の XML を蓄積し, 高速に検索できる
- スキーマ・データ構造の変更に柔軟に対応できる
- データ交換で流通する XML データをそのまま蓄積・検索できる

であり, RDB と比較して「柔軟性」を持っている.

実装例としては, NeoCoreXMS[1)], TX1[2)], pgBoscage[3)] 等が挙げられる. また, Oracle[4)], DB2[5)] といった従来の RDB においても, XML 機能が追加されており, PostgreSQL もデータ型の 1 つに XML 型が用意されている.[6)]

2.2 XML データベースの性能

XMLDB は柔軟性をもつことに加え, サポートする XML データ処理も幅広い. 操作言語として, XPath, XQuery, XSLT 等が W3C で標準化されており, RDB における SQL 以上に多くの機能をもつ仕様となっている.

一般的に言えば, XMLDB は RDB と比較し, 柔軟性・処理バリエーションともに機能的な優位性をもつため, そのトレードオフとして処理性能が劣ることは避けられない. そのため, 前述の XMLDB 実装では, 性能低下を抑えるために様々な技術・手法が考案, 実装されている.

†1 NTT サイバースペース研究所
NTT Cyber Space Laboratories

Vol.2012-DPS-150 No.23
Vol.2012-CSEC-56 No.23
2012/2/29

3. 高速 XML データ型の提案

実際のシステムにおいて, DB アクセスが少なく負荷が低い場合, DB 上で多様で複雑な処理を実行させることはアプリケーション (AP) 開発が効率化できる. しかし DB アクセスが多く負荷が高い場合は, DB 上では限定された単純な処理のみを実行させることで負荷を抑え, 多様で複雑な処理は AP 側で分散実行するよう設計する. これは従来の RDB システムにおいても同様である.

したがって, XMLDB の実装においては, 多様で複雑な XML データ処理は機能的にサポートしていればよく, それほど高性能でなくてよい. 一方, 限定された単純な処理については, 高性能に実現していることが重要となる.

そして, 高性能にすべき単純な処理とは,

- 特定の値 (XML データのテキストノード/属性値) の参照, 更新
- XML データ全体の取得, 挿入, 更新

であることが経験的にわかっている. 前者はインデクス作成時には全データに対して行われる必須の処理であり, 後者は複雑な処理を行う AP とデータ授受を行う際に必須の処理である.

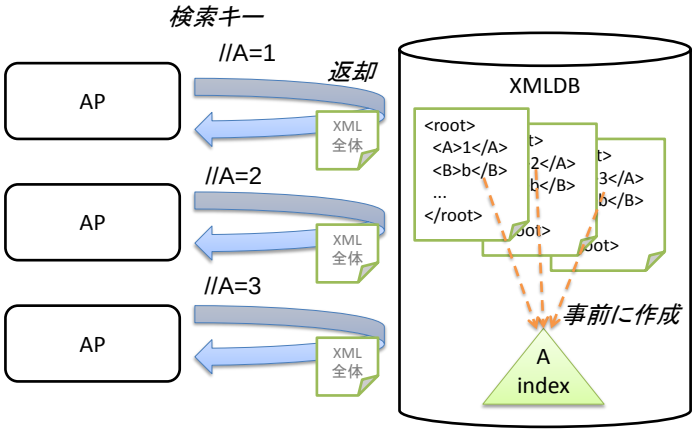

図 1 想定するシステム構成

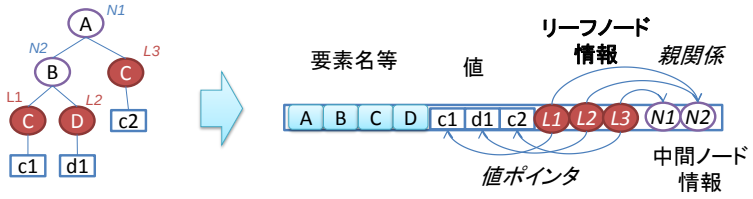

図 2 提案する XML データ型概要

典型的な例として, 図 1 のようなシステム構成および DB アクセスが挙げられる.

XMLDB に対し AP が複数用意される. 各 AP は XMLDB に対し, 検索キーとしてリーフノードとその値を指定し, マッチする XML データをまるごと取得し, AP 内でそれぞれの処理を行う. この場合, XMLDB 側に要求される処理は,

- 事前に, 検索キーに指定されるリーフノードの値に対するインデクスを作成する
- AP からの検索要求を受け, インデクスを使って検索キーにマッチする XML データを特定し, まるごと返却する

ことのみとなる.

3.1 提案する XML データ型

本論文では, データ列から直接値にアクセスでき, データサイズを小さくすることで, XML データ全体の格納・取り出し・AP との授受を高速化する, 新しい XML データ形式を提案する.

提案するデータ型の概要を図 2 に示す. 値を保持するリーフノードを特別扱いし, 中間ノードとは分けて特定の領域に格納する. これにより, 値へのアクセス時はリーフノードを中心に走査することで, 高速な処理を可能とさせる.

また, データサイズを小さくするために

- 名前空間, 要素, 属性の名称を ID(整数) 化する
- ID は小さい値の場合少ないバイト数で表現できるような符号化を行う
- ノード間の関係は子から親のみ保持し, 親から子の関係は保持しない

といった工夫を行う.

例えば, 以下の XML データは図 3 に示すデータ列で保持する.

Vol.2012-DPS-150 No.23
Vol.2012-CSEC-56 No.23
2012/2/29

```
<A xmlns='ns0'>
  <BY>
    <C X='x1'>c01</C>
    <D/>
  </BY>
  <C xmlns='ns1'>c002</C>
</A>
```

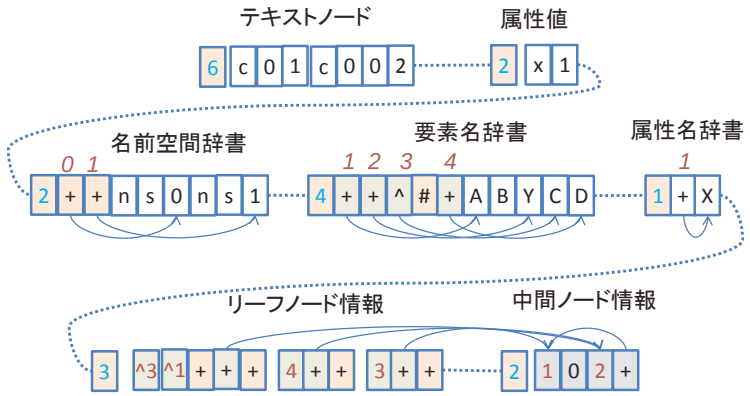

図 3 データ列の例

テキストノードの値, 属性値, 名前空間, 要素名, 属性名, リーフノードの情報, 中間ノードの情報, と各領域に分けて保持する. リーフノードの情報はノードの出現順に配置し, 要素名の ID, テキストノードへのポインタ (オフセット), 属性名の ID と属性値へのポインタ (オフセット) のペアを任意数, 親ノードへのポインタ (オフセット) が記されている.

さらなる工夫として, 名前空間, 要素名, 属性名の領域を外部保持する手法も考えられる. 通常, 同一カラムには同一スキーマの XML データが格納されることが想定できるため, 共通的な情報として外部で管理することで, データサイズが更に縮小できる可能性がある.

このデータ列に対し, 値へアクセスする場合の手順を考える. 例えば,

```
//C/text()
```

といったリーフノードの要素名のみを指定した参照クエリ (XPath) を処理する場合, 以下の手順で行える.

表 1 測定環境

測定マシン	Dell PowerEdge 1900 (2008 年製) CPU: QuadCore Intel Xeon X5365 3.00GHz × 2 RAM: 16GB HDD: 1TB SATA II 7200rpm × 6 (RAID5)
OS	RedHat Linux 5
PostgreSQL	9.1.2

(1) クエリの XPath をパースし, リーフノードの名前空間と要素名を取り出す.
(2) データ列から, すべての名前空間と要素名を参照して, 要素名 “C” の ID を特定する.
(3) リーフノード情報を読み出し, 要素名の ID がマッチするものを特定する.
(4) 特定したリーフノード情報からテキストノード値へのポインタ (オフセット) を辿って, 値を取得する.

つまり, 基本的にはリーフノードの全探索を行うのであるが, メモリに載り切らないほどの大きな XML データではなく, 数多くある小さな XML データを扱うことを想定しているため, 高速に実行される.

また, リーフノードのみの指定だけでなく,

```
/A/BY/C/text() や //A//BY//C/text()
```

のような, 上位ノードも指定しているような XPath についても, 手順 (3) で特定したリーフノードから親ノードの関連を辿っていくことで検証でき, 直接アクセスできる.

それ以外のクエリについては, 高速処理の対象外とし, データ全体を一旦 XML 文字列化もしくは DOM 化し, 既存の XML 処理ライブラリを用いて操作することで機能的なサポートのみとする.

4. PostgreSQL での実装・評価

前章で提案した XML データ型およびテキストノード値への参照および更新関数を PostgreSQL 上に実装し, 性能評価を行った.

4.1 測定ツール・条件

測定に用いたマシン, PostgreSQL のバージョンを表 1 に示す. 測定ツールは, PostgreSQL の contrib に同梱されている pgbench[7] を XML 用に改造して用いた.

格納するデータについては, 例えば “pgbench_account” というテーブルがあり,

```
(aid int not null, bid int, abalance int, filler char(84))
```

Vol.2012-DPS-150 No.23
Vol.2012-CSEC-56 No.23
2012/2/29

というカラムが定義されているが, XML 用には “data” という 1 カラムとし,

```
<accounts>
  <aid>0</aid>
  <bid>0</bid>
  <abalance>0</abalance>
</accounts>
```

といった, カラム名を要素名とした XML データとした. ここで XML データの柔軟性を活かすという観点から, 項目 “filler” は省略することとしている. また, PostgreSQL のテーブル定義オプション fillfactor はすべて 90 とした. *1

pgbench の DB アクセスは 3 種類のクエリ

- 特定の値を検索キーに, 特定の値を参照
- 特定の値を検索キーに, 特定の値を更新
- XML データをまるごと挿入

が混じったトランザクションであり, 本論文で想定しているクエリとなっている.

測定対象は以下の 4 種類とし, スケールファクタ (sf) を変えて測定を行った.

- 「normal」: 通常の pgbench(RDB での利用)
- 「xml」: PostgreSQL のデフォルトの XML 型
- 「LSXML」: 本論文で提案する XML 型
- 「LX(no dic)」: 本論文で提案する XML 型から名前空間, 要素名, 属性名を別テーブルに格納・管理したもの

また, トランザクションの測定においては, 通常の更新クエリを含むアクセスパターンと参照クエリのみのアクセスパターンの 2 種類行い, 測定時間は 15 分とした. ただし, トータルで算出される TPS 値は測定ごとにバラツキが大きかったため, 全レスポンス値から最良値および最悪値を 0.5%ずつ除いて算出した TPS 値を今回の測定結果として採用している.

4.2 測定結果

DB 作成

まず, 全データ挿入とインデクス作成時間を合わせた, DB 作成時間の結果を図 4 に示す. デフォルトの XML 型と比較し, 提案する XML 型 (LSXML) は 40%以上短縮できている.

*1 PostgreSQL では「ページ」と呼ばれる領域単位でデータを管理していおり, データ更新時には新データを同じページ内に配置する方が性能的に有利となる特性をもっている. そのためには, ページ内に空き領域を作っておく必要があり, fillfactor は使用領域の割合を指定するオプションである.

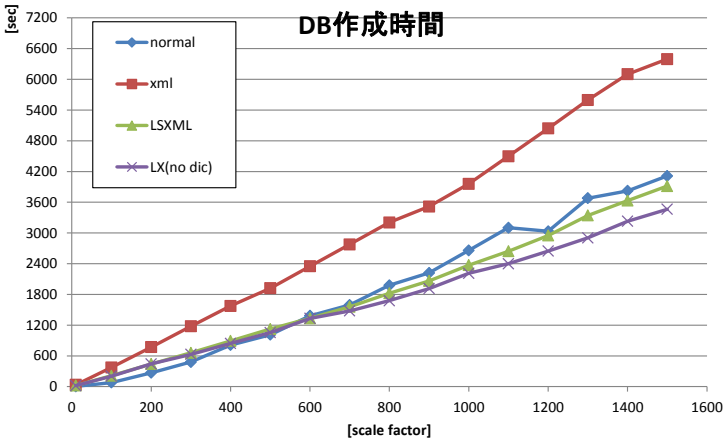

図 4 DB 作成時間

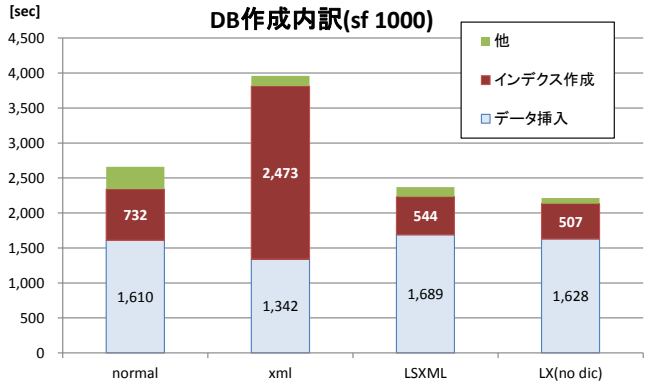

図 5 DB 作成時間の内訳 (sf 1000)

DB 作成は, データ規模が大きくなれば何時間, 何十時間という単位になるため, この時間が大きく短縮できることは, 非常に有益と言える.

短縮された理由は, 狙い通りインデクス作成時間の大幅な改善によることが, 処理時間の内訳 (図 5) からわかる. デフォルトの XML 型では, XML 文字列をそのまま格納しているため, 全カラムの XML データに対し, 再度 XML パースして DOM 作成し, さらに XPath

Vol.2012-DPS-150 No.23
Vol.2012-CSEC-56 No.23
2012/2/29

適用を行うという 2 ステップが必要になり, スケールファクタ 1000 において 2473 秒かかっている. それに対して, 提案する XML 型は XML パースが不要で 1 ステップかつ高速に値参照できるため, 544 秒で実行でき 78%の短縮が実現できている.

逆にデータ挿入時間が 1342 秒から 1689 秒と 26%大きくなっているのは, XML パースに同じ時間かかり, さらに独自のデータ形式にエンコードする時間が加わるためである. これを改善するためには, XML パースとエンコード時間の短縮手法を検討する必要がある.

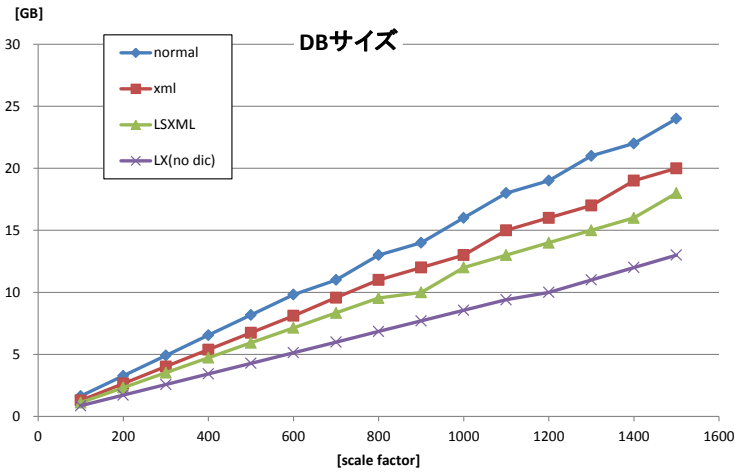

図 6 DB サイズ

次に, DB サイズについて図 6 に示す. デフォルトの XML 型と比較し, 提案する XML 型 (LSXML) は 13%圧縮できている. さらに要素名等を別格納した LX(no dic) については 35%圧縮できている.

また, 通常の pgbench(normal) の方が DB サイズが大きいのは, 項目 “filler” のためであることを付記しておく.

トランザクション

トランザクション性能について, 横軸にスケールファクタ (sf) を変化させていったものを図 7 に示す. 上が更新クエリを含むトランザクションで, 下が参照クエリのみのトランザクションを実行した結果である.

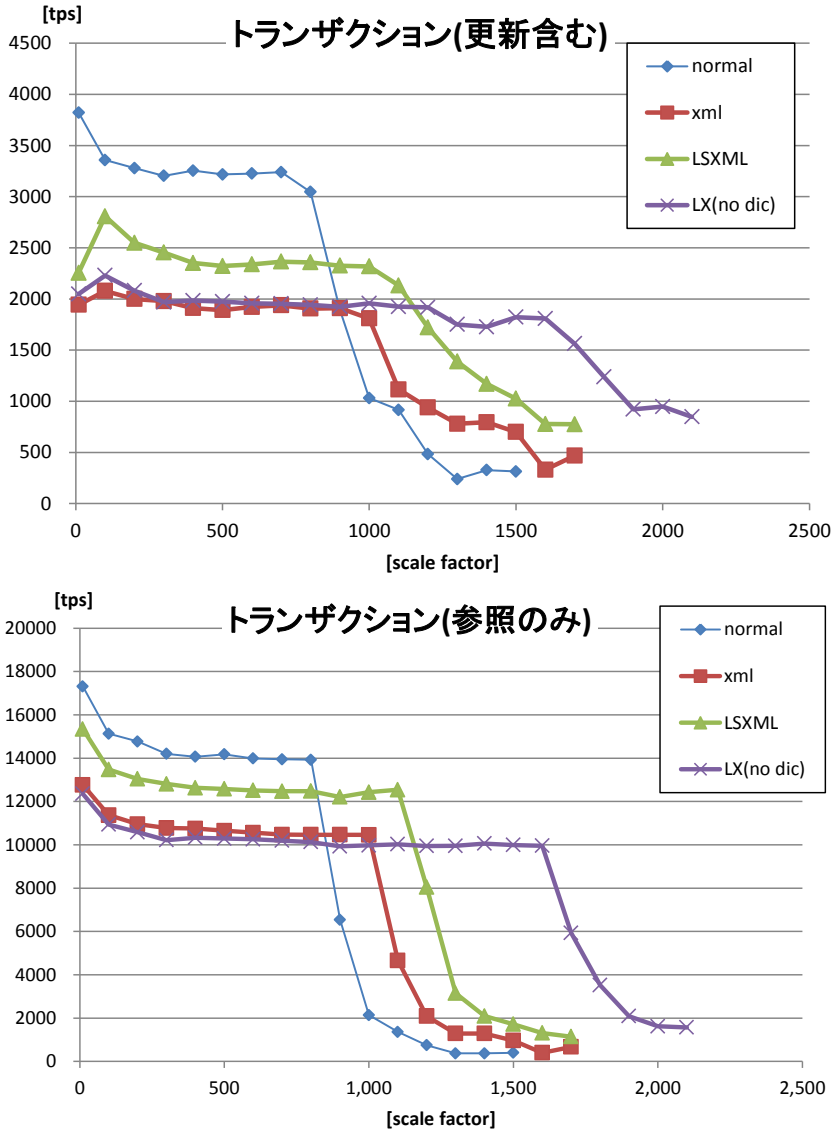

図 7 スケールファクタごとのトランザクション (同時接続 6)

Vol.2012-DPS-150 No.23
Vol.2012-CSEC-56 No.23
2012/2/29

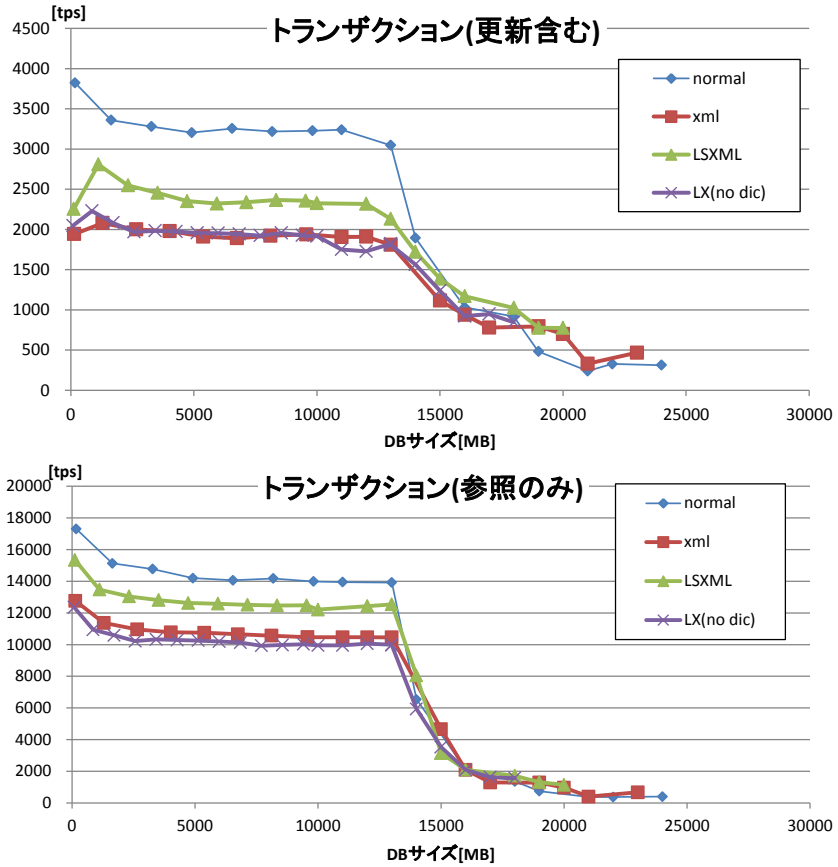

図 8 DB サイズごとのトランザクション (同時接続 6)

まず, トランザクション性能が限界に達するまでの間 (sf が 1000 以下) においては, デフォルトの XML 型では更新を含むアクセスで 2000TPS 前後, 参照のみのアクセスでは 11000TPS 前後となっている. それに対し, 提案する XML 型 (LSXML) では更新を含むアクセスで 2400TPS 前後と 20〜23%の向上, 参照のみのアクセスでは 12500TPS 前後と 17〜20%の向上が確認できた. それに対し, 要素名等を別格納した LX(no dic) については, 更新を含むアクセスでは数%の向上, 参照のみのアクセスでは逆に数%低下した. これは, 要素名等を管理する外部テーブルへのアクセスが予想以上に足枷となってしまい, 性能向上しなかったと考えられる. 要素名等の管理テーブルはサイズも小さく今回の場合は更新もないことから, ほとんど性能に影響しないと予想していたが, 実際はそうではなかった.

また, スケールファクタ (sf) の限界については, デフォルトの XML 型が 1000 を超えたところから大きく性能を落としているのに対し, 提案する XML 型 (LSXML) は 1100 を超えるまで耐えている. 要素名等を別格納した LX(no dic) については, さらに 1600 まで耐えている. ただしこれは, ほとんど DB サイズが支配項となっていることがわかり, 図 8 に横軸を DB サイズに変えたものを示すと, どれも DB サイズが 13GB を超えたところで大きく性能が低下している. つまり, 今回の測定環境では 13GB が DB サイズの限界であり, データサイズを小さくできたものほど限界値を伸ばすことができたと解釈できる.

以上より, PostgreSQL 上において, 提案する XML 型の性能的優位性が確認できた. ただし, 要素名等を別格納する方式については, DB サイズが非常に大きい場合は有効とも言えるが, 実装コストの複雑さと限界に達するまでの間の上限性能を考慮すると, 総合的にはあまり推奨できない.

5. おわりに

本論文では, XMLDB の性能向上を目的に, データベースで必須かつ高頻度で実行される機能に絞って高速化を行う, 新しいデータ型の提案を行った. そして PostgreSQL 上で実装を行い, 性能測定を行ったところ, デフォルトの XML 型と比較し, DB 初期化時間では 40%以上短縮, トランザクション性能では 20%前後の性能向上が確認を確認することができ, 手法の有効性を確認できた.

参 考 文 献

1) http://www.cybertech.co.jp/xml/xmldb/neocorexms/
2) http://www.toshiba-sol.co.jp/pro/xml/
3) 兵藤正樹, 江田毅晴, 榎本俊文, 山室雅司: pgBoscage : PostgreSQL を用いた XMLDB の実装, 電子情報通信学会総合大会 (2008)
4) http://www.oracle.com/technetwork/database/features/xmldb/index.html
5) http://www-01.ibm.com/software/data/db2/xml/
6) http://www.postgresql.org/docs/9.1/interactive/datatype-xml.html
7) http://www.postgresql.org/docs/9.1/static/pgbench.html