要求駆動型 XML 計算環境 Nanafusi の実装と評価

新村健治[†,☆] 山中真和[†] 鎌田十三郎[††]

多くのデータが XML 形式でネットワーク上に公開され，それにともない，ストリーム入力される XML データを処理する機会が増加している．ネットワークから受信したデータを効率的に処理し，素早く返答を返すためには，パイプライン的な処理や各種遅延隠蔽が重要である．本論文は，遠隔データの処理効率化のための XML 計算環境 Nanafusi を提案する．システムは，データバインディングツールと木構造演算ライブラリから構成され，プログラマは XML を木構造データとして簡単にアクセスすることができる．一方で，システム内部では，データ処理プログラムからのアクセス要求に応じて，各種計算の遅延評価や部分木の遅延構築を行うことで，パイプライン的なデータ処理や各種遅延の隠蔽を目指した．作成したシステムの評価は，複数のベンチマークを通して行われ，従来手法に比べて，反応速度向上やメモリ使用量削減に加え，全実行時間も最大 9%の短縮を示すことができた．

Nanafusi: A Programming Environment for Demand-driven XML Processing with an Efficient Implementation

Kenji Niimura,[†,☆] Masakazu Yamanaka[†] and Tomio Kamada[††]

The opportunity to process streaming XML data across a network is increasing as many kinds of information are exhibited as XML data. To process multiple network data efficiently and return quick responses, it is important to adopt techniques such as pipelining processing and latency hiding. This paper proposes a programming environment for network XML data processing. Our system consists of a databinding tool and a XML tree operation library. Using our system, programmers can handle XML data as typed-tree structure objects. And our system provides efficient data processing based on demand-driven evaluation, which enables process pipelining and latency hiding. Through the performance evaluation on several benchmarks, our system reveals quick response, and also succeeds to reduce the total execution time in many cases where join operations are used (9% reduction at maximum).

1. はじめに

近年，ネットワーク上に規格化された XML データが公開され，第 3 者がこれらの公開データを各種要望に応じて 2 次加工する機会が増大している．たとえば，検索エンジンの Google や書籍データを管理している Amazon などでは，Web サービスという形でそれぞれが保持するデータを XML 形式で公開している．これにともない，様々なユーザの要望に対して，必要なデータを随時取り寄せて処理することも可能となった．

このようにネットワーク上の XML ストリームデータを受信して処理するサービスをプログラマが実装する場合には，応答時間に影響を及ぼす様々な要因をかかえることになる．たとえば，表やリストを含んだ大きな XML 文書を処理する場合，全データ到着待ちによるデータ受信待ち遅延や構文解析処理完了待ちによる遅延が問題となる．また，計算途中にネットワーク問合せを行う必要がある場合は，その待ち時間が問題となる．このような問題に対処するためには，受信したデータから随時処理を進めるなどのパイプライン的なデータ処理やネットワーク問合せの非同期化などが必要となる．しかし，現状で一般プログラマがこのような処理を実現するには，イベント駆動型プログラムの実装やマルチスレッドプログラムのスケジューリング管理などを自身で行う必要があり，容易な作業とはいえない．

そこで本研究では，プログラム記述性を保ちつつ，遠隔 XML データ処理のための実装技術を備えたプログラミング環境を提案する．本環境は，ネットワー

† 神戸大学大学院自然科学研究科情報知能工学専攻
Graduate School of Science and Technology, Kobe University
†† 神戸大学工学部情報知能工学科
Department of Computer and Systems Engineering, Faculty of Engineering, Kobe University
☆ 現在，株式会社リコー
Presently with Ricoh Co., Ltd.

ク上の XML データを木構造データにマッピングするデータバインディングツールと，木構造データに関する合成演算ライブラリから構成される．プログラマは，これらを利用することにより，XML 文書を型づけされた木構造データとして操作することができる．一方で，システム内部では，データ処理プログラムからのアクセス要求に応じて，各種計算の遅延評価や部分木の遅延構築を行うことで，パイプライン的なデータ処理を目指した．また，ライブラリでは非同期呼び出し機構も提供しており，ネットワーク問合せ遅延の隠蔽もできる．加えて，メモリ使用量削減や再帰的データの処理に向けたシステムデザインや実装も行っている．

論文構成は，以下のとおりである．まず 2 章でネットワーク越しの XML 処理事例を取り上げ，効率的な処理の実現に向けての課題を整理する．3 章と 4 章で提案プログラミング環境のデザインと実装法について述べ，5 章で本システム性能評価を行う．6 章で関連研究について述べ，7 章でまとめる．

2. 効率的な処理の実現に向けての課題

ネットワーク上の複数の XML を処理する例を通して，効率的な処理の実現に向けての課題を検討する．

今回想定する例は，賃貸住居検索サービス作成事例であり，近くに好みのコンビニエンスストアがあり，ADSL サービスが利用可能な物件を検索する．各情報は，以下の別々の WEB ページから取得する．

(a) ネット上の住宅情報誌は，指定地域の住宅物件情報を提供しており，家賃や所在地情報も含まれる．
(b)「i タウンページ」より，指定コンビニエンスチェーン店舗の所在地と電話番号情報を取得できる．
(c) ADSL 事業者の WEB ページでは，指定住所や電話番号からサービス提供の有無を取得できる．

(a)，(b)，(c) は現状で必ずしも XML 化されていないが，XML 化されていると仮定し，実現方法を考える．

計算手順は，おおむね以下のとおりである（**図 1** 上図）．まず指定地域の物件一覧 (a) と指定地域の店舗一覧情報 (b) を取得し，所在地に関して Join 演算を行う．最後に情報 (c) の検索は，Join 演算結果の各要素に対して逐一ネットワーク問合せを行う．

以上の手順を DOM や一般のデータバインディングツールを利用し逐次的に処理した場合，プログラムの記述面では XML を木構造として扱え簡単である．ただし，処理の効率面では，(a)，(b) でデータ転送待ち，(c) では外部問合せ結果待ちなどのデータ待ち時間が生じ，加えて，Join 計算がすべて完了しないと最初の結果が出てこないため，サービス全体のレスポンスも遅くなる（図 1：逐次処理）．

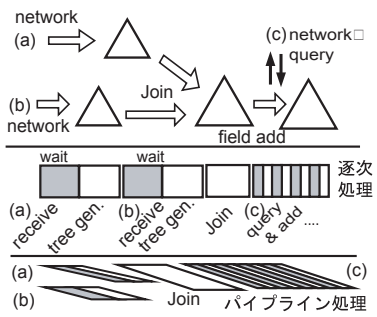


図 1 サンプルケースにおける処理の流れ
Fig. 1 Process flow and diagrams for a sample case.

以上の問題を解決するには，(a)，(b)，Join，(c) の計算をパイプライン的に処理し，遅延隠蔽を行う方法が考えられる（図 1：パイプライン処理）．このような処理には，SAX などのイベント駆動的なプログラミングが適切であるが，課題も多い．1 つの課題は，Join などの合流操作である．Join 結果をすぐに次の段階に伝達するには，2 系統のデータ入力それぞれに対しイベント駆動スタイルの処理をマルチスレッドで実現し，両スレッドの面倒な同期作業を行う必要がある．もう 1 つの課題は (c) の問合せ作業である．待ち時間の隠蔽のためには，プログラマ自体が別スレッドによる非同期問合せや並列問合せ数を制限するなど非同期問合せ間の協調処理を実装する必要がある．このように効率的な処理の実現には，プログラマへの負担が大きく，実装が容易であるとはいえない．

3. 提案プログラミング環境

3.1 概　　要

Nanafusi では，XML データは XTree と呼ぶ木構造にマップされ，データ処理は XTree の走査/合成を通して行う．また XTree に対する更新操作を許さないことで，計算実行順序によって結果の変わらないモデルを導入し，システム側でスケジューリング変更を行いレスポンス向上や遅延隠蔽を目指す．

本システムは，ネットワーク上の XML データを XTree にマッピングするデータバインディングツールと，XTree に関する合成演算ライブラリから構成される．基本的なプログラミングスタイルとしては，まず外部データを XTree にマッピングし，その後，XTree の各部位にアクセスしながら合成演算ライブラリを利用して，木構造変換/合成の形で計算を行う．このように，プログラマは本システムを利用することで，XML

を木構造データとして簡単にアクセスすることができる．また，従来レスポンス向上や遅延隠蔽のためにプログラマ自身が行う必要があったプログラムのマルチスレッド化やスレッド間の同期などはシステム側で行われるため，その詳細な実装作業からプログラマは解放される．

次に，内部挙動について簡単に紹介する．本ツール内部では，データ処理プログラムからのアクセス要求に応じて，各種計算の遅延評価や部分木の遅延構築を行っており，データ処理はパイプライン的に実行される．また，2 章の (c) のように，計算途中に随時ネットワーク問合せを行う箇所は，先行実行や並行問合せを行うこと（以後，**Future** 演算と呼ぶ）ができ，ネットワーク問合せ遅延の隠蔽も可能である．

以後，Java Generics 対応の Java 上に実装したシステムの詳細について，2 章の例を通して説明する．

3.2 木 構 造

Nanafusi では，ある形をした XTree は，複数の方法によって生成されることがある．たとえば，ある形をした XTree の木構造は，XML データの構文解析の結果として生成される場合もあり，また，演算ライブラリの結果として出力される場合もある．本システムでは，XTree を型クラス（子要素へのアクセス方法を保持）と実装クラス（各種変換処理の実体）の 2 種類に分離して，同じ形を持っている木構造は，生成方法にかかわらず，同一のインタフェースで処理可能にしている．

たとえば，2 章の (a) のデータをマップした型クラスと実装クラスは，**図 2** のようになる．まず型クラスは，通常の要素木を表すクラスと，要素木の集合を表すリストクラス（XList ⟨T⟩）から構成される．(a) のデータの個々の物件情報はクラスの `House` 型として表現され，木構造が保持する各子供木/葉へのアクセサが定義される．また，`House` のリストは `XList⟨House⟩` で表現され，`XIterator⟨House⟩` を介して順次アクセス可能である（逆順アクセス不可）．また，(a) の木全体はクラス `HouseInfo` 型として格納される．一方で，実装クラスは各種変換処理の実体であり，変換後に生成される木構造の型クラスを継承した形で実装される．たとえば，図 2 の `HouseInfoImpl` は，XML データの構文解析をして `HouseInfo` 型の木構造へ変換するクラスであるため，変換後に生成される木構造の型クラス（`HouseInfo`）を継承する形で実装される．

次に，XTree の特徴には，ユーザ側からは XTree への更新を許さないデザインになっており，そのため，木構造の葉で許されるデータ構造は Immutable な基本データ型のみであり，部分木の値を変更する setter メソッドを提供していない．また XTree ではデータの利用状況に応じて 2 種類のアクセサを使い分けることで，メモリ使用量の削減を行うことができる．XTree では，部分木へのアクセサとして通常の `get` 系アクセサのほかに，メモリ使用量削減を意図したアクセサである `remove` 系アクセサを提供しており，ある XTree の部分木を一度だけ走査するような場合に利用することで，各部分木を親木から切り放し，ゴミ集め（GC）の対象にできる．ただし，切放し後のアクセスでは実行時例外が発生する．

```
/** 型クラス */
interface XTree { // marker interface }
abstract class HouseInfo implements XTree {
    abstract XListHouse⟩ getHouses();
    abstract XListHouse⟩ removeHouses(); }
abstract class House implements XTree {
    abstract Integer getPrice();
    abstract Integer removePrice();
    abstract String  getAddr();
    abstract String  removeAddr(); }
abstract class XListT⟩ {
    abstract XIteratorT⟩ iterator();
    abstract XIteratorT⟩ removeIterator(); }
interface XIteratorT⟩ {
    T next();    boolean hasNext(); }

/** 実装クラス */
class HouseInfoImpl extends HouseInfo {
    HouseInfoImpl(URL url){ ... }
    XListHouse⟩ getHouses(){ // ... } //...
}
class HouseImpl extends House { // ... }
```

図 2 XTree クラス（例）
Fig. 2 `XTree` class (sample code).

3.3 データバインディング

XML データを XTree にマッピングするプログラムは，以下で説明するマッピング指定ファイルに対して，コマンドを実行することで自動生成される．

まず XTree へのマッピング指定であるが，RELAX NG [5] で書かれたスキーマに `c:class`，`c:field`，`c:package` の 3 種類のアトリビュート（`c:`は名前空間指定の prefix）の追加記述を通して行う．本ツールは，他のデータバインディングツールと比べて，XML 文書とデータ構造とのマッピングに柔軟性があり，不要な部位のインスタンス化は当初から避けることもできるという特徴がある（詳細は文献 6) を参照）．

図 3 は，(a) の入力データを図 2 の XTree クラスに変換するためのマッピング指定例である．生成された実装クラス群の内部では，XTree の部分木へのアクセス要求に応じて，必要な部分まで構文解析を進める．生成された実装クラス群の内部では，XTree の部分木へのアクセス要求に応じて，対応部分木の出現の有/無が確定する部分（たとえば，アクセスされた部分木

```
<start c:class="HouseInfo">
  <element name="houseList" >
    <oneOrMore>
      <element name="house" c:class="House" c:field="houses">
        <element name="price">
          <data type="int" c:field="price"/></element>
        <element name="address">
          <text c:field="addr"/></element>
        .....
      </element>
    </oneOrMore>
  </element>
</start>
```

図 3 マッピング指定（例）
Fig. 3 Mapping rule (sample).

```
<class name="JHouse">
  <field name="addr" type="String" />
  <field name="price" type="Integer" />
  <field name="shops" type="XList<Shop>" />
</class>

abstract class JHouse implements XTree {
    abstract Integer getPrice();
    abstract String  getAddr();
    abstract XList<Shop> getShops(); // ...
}
```

図 4 型クラス生成定義ファイル（例）
Fig. 4 Generating structure class (sample case).

に対応する開始タグの出現ポイント）まで構文解析を進め，部分木の作成を行う実装が施されている．ただし，現在の実装では，扱える文法は 1 つ先読みで構文解析可能なものに限られている．

3.4 木構造の合成

プログラマは以下の 3 種類の基本操作を通じて，XTree クラスの作成や合成演算の実現を行う．

- 新規型クラスの作成
- 既存木の足し合わせ（Delegation）
- 集合演算テンプレートの利用

本節では，2 章の Join 演算を例にして木構造の合成について説明する．この例は，データバインディングで生成済みの House 型と Shop 型の集合から，各 House 要素に対し，住所が近い複数の Shop 要素を結合し，JHouse 型の要素集合を返すものである．

新規型クラスの作成： 作成補助プログラムが用意されており，**図 4**（上）の XML 形式の定義ファイルを与えると，図 4（下）に型クラス JHouse が生成される．

既存木の足し合わせ（Delegation）： 既存の XTree を足し合わせて新たな木構造を作成するとき，新たな木構造へのアクセス要求を合成元の木構造に委譲（delegation）するといった，要求駆動に基づくスタイルをとる．今回の例では，既存の木構造の足し合わせをするクラスは，**図 5** の JHouseImpl のようになり，部分木へのアクセス要求は合成元の木構造である `house` や `shops` に委譲される．このようなスタイルをとることで，合成木の各要素がアクセスされるまで，合成元の木構造の作成に必要な計算は行われない．

```
public class JHouseImpl extends JHouse {
  private House house; private XListShop⟩  shops;
  JHouseImpl(House house,XListShop⟩ shops){
    this.house = house; this.shops = shops; }
  public Integer getPrice(){return house.getPrice();}
  public String  getAddr(){return house.getAddr();}
  public XListShop⟩ getShops(){return shops;} // ...
}
```

図 5 既存木の足し合わせ（例）
Fig. 5 XTree composition (sample code).

```
/** Join のテンプレートクラス */
abstract class MJoin<IN0,IN1,OUT> extends XList<OUT>{
  MJoin(XIterator<IN0> in0,XIterator<IN1> in1){ ..}
  MJoin(IProducer<IN0,IN1> inPro){ ... }
  abstract OUT  constElem(IN0 in0,XList<IN1> in1);
  abstract boolean condition(IN0 in0,IN1 in1);
  public XIterator<OUT> iterator(); // ....
}
/** テンプレート利用例 */
class MJoinImpl extends MJoin<House,Shop,JHouse>{
  MJoinImpl(XIterator<House> in0,XIterator<Shop> in1){//(1)
    super(in0,in1);  }
  MJoinImpl(IProducer<House,Shop> inPro){super(inPro);}//(2)
  JHouse constElem(House in0,XList<Shop> in1){
    return new JHouseImpl(in0,in1);  }
  boolean condition(House in0,Shop in1){
    return in0.getAddr().startsWith(in1.getAddr()); }
}
/** main 文（利用例） */
XList<House>  h = new HouseInfoImpl(H_URL).getHouses();
XList<Shop>   s = new ShopInfoImpl(S_URL).getShops();
XList<JHouse> j = new MJoinImpl(h.iterator(),s.iterator());
XIterator<JHouse> it=j.iterator(); // ....
```

図 6 集合演算テンプレートとその利用例 1
Fig. 6 XList operation (sample case 1).

プログラマが `JHouseImpl` クラスのようなクラスを作成したい場合は，システム側が提供する作成補助プログラムを利用すれば簡単に作成できる．またプログラマ自身が，木構造の足し合わせ/新規作成することも可能であるが，部分木へのアクセサはいつ呼ばれても同じ結果を返すように実装しなくてはならない．

集合演算テンプレートの利用： 本システムでは，要素木のリストに対する Selection，Join などのデータベース的な集合演算と全集合要素に特定の変換を行う ListMap 演算などをテンプレートライブラリの形で提供している．たとえば，1 対多結合を行う MJoin 演算は，**図 6** の MJoin のような形で提供されている．この MJoin 演算は，入力集合を合成して OUT 型のリストに変換するプログラムであるため，XList⟨OUT⟩ を継承するクラスとして宣言される．また演算内部では，処理効率を考慮した様々な実装が施されている．

プログラマが 2 章の例を作成する場合は，図 6 の `MJoinImpl` のように，テンプレートとなるクラス（この例では，MJoin）を継承し，コンストラクタ実装と結合条件判定メソッド（condition）と個々の結果の作成メソッド（constElem）を実装するだけでよく，演

```
class Visitor extends IProducer<House,Shop> {
  void run(){
    traverseTree( root );
    super.putEnd0(); super.putEnd1();// 入力取り込み終了
  }
  /** XTree をトラバースして，入力要素を見つける */
  void traverseTree(RootNode root){
     ...; super.putInput0( aHouse );// House を発見
     ...; super.putInput1( aShop ); // Shop  を発見
  }
}
/** main 文 (利用例) */
RootNode root = new RootNodeImpl(INPUT_URL);
IProducer<House,Shop> inpro = new Visitor( root );
XList<JHouse> j = new MJoinImpl(inpro);
for(XIterator<JHouse> it=j.iterator(); it.hasNext()){
   JHouse jhouse = it.next(); // ....  }
```

図 7 演算クラスの利用例 2
Fig. 7 XList operation (sample case 2)

算内部の詳細な実装を知る必要はない．

次に，作成した演算クラス `MJoinImpl` の入力集合の指定方法について．まず，2 章の Join 演算のように入力 XML データの構造が表データのような単純で，簡単に入力集合を取得できる場合は，図 6 の main 文のように演算クラスのコンストラクタの引数として渡すことで指定できる．

一方で，入力要素が再帰的な構造など複雑な木構造中にあり，木構造の階層に関係なく散在しているような場合には，簡単に入力集合へのアクセサが取得できない．このような場合には，入力集合を取り出す方法を記述したクラス（`IProducer<IN0,IN1>`型）をプログラマが作成することで簡単に，入力集合の指定ができる．たとえば，1 つの木構造をトラバースして，House 型と Shop 型の木構造を取り出すクラス（`IProducer<House,Shop>`型）は，**図 7** の Visitor のようになる．木構造をトラバースし，House/Shop 型の木構造入力要素（aHouse/aShop）を見つけるごとに，`putInput0(aHouse)/putInput1(aShop)` を呼び出し，これ以後，House/Shop の要素が出現しなくなった時点で，`putEnd0()/putEnd1()` の呼び出しを行っている．IProducer を利用した `MJoinImpl` への入力集合の指定方法は，図 7 の main 文のようになる．

最後に，Future 演算について．演算テンプレート内部では，本来，ユーザプログラムから結果へのアクセス要求である `XIterator<OUT>#hasNext()/next()` メソッドが呼ばれた時点で，次の結果要素の作成を行うものであり，基本的に必要になるまで計算は行われない．ただし，パラメータを変更することで状況に応じて計算順序を調整することが可能である．たとえば，2 章の (c) の箇所では，ユーザは演算クラスの結果作成メソッド（constElem）の中で，ネットワーク処理内容を書いておき，最大並列問合せ数（num 個）を指定するだけで，先行非同期問合せが実現できる．

4. 実 装 法

4.1 スケジューリング方針

全般的なスケジューリング方針： 本ツール内部では，データ処理プログラムからのアクセス要求に応じて，XTree の部分木の遅延構築を行っており，なるべく木構造の構築を遅らせる方針をとっている．ただし，現在の遅延評価機構の一部は，マルチスレッドを利用して実装しているため（4.2 節），完全に要求駆動で実行すると，頻繁なスレッド切替えが発生してしまう．このため，マルチスレッドで実装された部分は，バッファリング（ユーザ指定可）による効率化を行っている．なお，バッファリングに関する評価は，5.2 節で行う．

1 対多結合を行う演算での方針： 3.4 節の MJoin などの 1 対多結合を行う演算では，多結合する要素が何個見つかった時点でデータ処理側に結果を返すのかについては，様々な選択肢が考えられる．たとえば，多結合する要素をすべて求めてから，個々の結果を返す場合には，1 つの結果を取り出すだけでも，演算内部では多結合する側の入力集合をすべて読み込む必要があり，反応速度が問題となる．また，多結合する要素が 1 個見つかった時点で結果を返す場合には，多結合した要素の 2 つ目の要素アクセスでは，計算はブロックされる可能性がある．このように，結果を返すタイミングには様々な選択肢があり，反応速度や計算のブロックに影響を与える．また，どのタイミングで結果を返すべきかは，データ処理側の処理内容により異なる．このため 1 対多結合を行う演算では，多結合する要素が何個（bounds 個）見つかった時点で，結果を返すのかを指定可能にしている．

4.2 個々の実装技術

データバインディング： 構文解析部を別スレッドに分離して Producer/Consumer スタイルの実装を行うことで，要求駆動で構文解析処理を実行させている．構文解析部は，イベント駆動型のパーザジェネレータ RelaxNGCC[7] を利用して生成させた SAX で書かれたプログラムであり，XTree をデータ格納のための同期構造体としてデータ処理部と協調動作させている．

集合演算ライブラリ： まず Selection，ListMap などの 1 入力演算では，各要素へのアクセス要求である `XIterator<OUT>#hasNext()/next()` メソッドが呼ばれた時点で，次要素がまだ確定していなければ，必要な計算だけを行うように実装が施されている．このため，1 入力演算が多段に使われた場合でも完全に

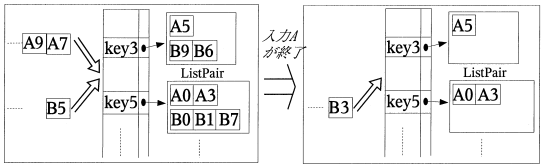


図 8 HashJoin の内部実装イメージ
Fig. 8 Internal behavior of `HashJoin`.

要求駆動で計算が進む.

一方で，Join などの 2 入力演算では，データベースなどでも行われているレイテンシ重視の実装により，反応速度向上やネットワーク遅延の隠蔽を目指している．ライブラリの実装では，2 入力の先頭部分どうしで Join 可能なものがあれば即出力する方針をとり，ライブラリ内部での入力集合のとり込みには，それぞれ別々のスレッドを割り当て，並行して処理を実行させている．加えて，入力側スレッドとデータ処理側スレッドを Producer/Consumer スタイルの同期で制御し，要求駆動に結果の生成を行わせている．

Join 演算の実装例（HashJoin）： Join 演算の実装では，上記の反応速度向上のための実装に加えて，メモリの使用量を考慮した実装も行っている．たとえば，Join 演算の実装の一例である HashJoin では，入力要素は，ユーザ指定の key ごとに ListPair というデータ構造に格納され，結合条件判定/結合演算は ListPair 内の要素どうしで行われている（**図 8**（左））．この際，HashJoin の内部では，図 8（右）に示すように，一方の入力集合の読み込みが完了した段階で，演算内部で保持している他方の入力集合への参照を破棄するように実装しており，演算内部で消費するメモリ使用量についても考慮している．

Future 演算： 結果作成メソッドだけを別スレッドで非同期に実行させている．演算処理スレッドは，個々の結果作成完了を待つことなく次々に結果作成メソッドの呼び出しを行うため，結果作成メソッドの呼び出しが並列化される．加えて，各非同期呼び出し処理の終了をコールバック機構により把握し，並列実行数を一定数以下に抑えるように演算処理スレッドの制御も行っている．

5. 性能評価

評価環境は，Pentium4 Xeon 2.4 GHz，1 Gbytes memory，Red Hat Linux 8.0 uniprocessor 仕様を用い，また，Java VM は SUN jdk1.5.0 alpha，構文解析部は RelaxNGCC ver1.1 を利用している．

5.1，5.2，5.3 節の各評価プログラム［Web 上[8] にて公開］を次の 3 つの方法で実装し，最初の解が出るまでの時間（first），全実行時間（total）を比較する．

- Naive：データを完全に木構造に展開し，すべての計算を行って次の段階の処理を実行する（本システムを改変して実装）．
- ED：全処理をイベント駆動型スタイルで実装（構文解析部は RelaxNGCC を利用）．
- DD（remove/no-remove）：本システム．要求駆動に計算を進める方法で，remove 系アクセサ使用/未使用を各々実装.

加えて，実行時間に対する影響を検討するために最大メモリ使用量（mem）を測定した．

次に，測定方法について．それぞれの入力 XML データは，別々のサーバ上にあるデータ転送プログラムと Socket 通信を行い取得しており，データ転送速度は約 22〜26 Mbps である．first/total については，クラスロードの時間やデータ転送速度の揺らぎの影響を抑えるため，評価プログラムを連続して 13 回実行し，最初の 1 回と遅い 2 回の値を除いた 10 回の平均値をとって測定した．また mem は，頻繁に明示的な GC 起動を行うプログラムにより測定した．

最後に，評価プログラムの実装について述べる．評価プログラムでのすべての Join 演算は，4.2 節で紹介した HashJoin のアルゴリズムを使って実装した．ただし Naive での Join 演算は，最初にすべての演算処理を実行するように変更を加えた．また，DD のデータバインディング部と Join 演算部では，バッファリングの調整を行い，スレッド切替えコストの削減も行った．

表 1 性能評価 1
Table 1 Performance results 1.

参考	first(ms)	total(ms)	mem(MB)
データ転送	7	463	-
RelaxNGCC	8	466	-
実験 1	first(ms)	total(ms)	mem(MB)
ED	47	487	1.0
DD(remove)	109	499	2.1
DD(no-remove)	117	524	3.2
Naive	539	539	3.0
実験 2	first(ms)	5th(ms)	25th(ms)
Naive	1,239	12,565	21,468
Future0	781	11,672	19,674
Future1	629	11,249	19,331
Future5	1,074	4,059	5,919
Future10	1,781	3,879	4,987

5.1 各機能の基礎的評価

2 章で紹介したアプリケーション例を通して，各機能に関する評価を行った（**表 1**）．評価データは，(a), (b) については 2 章で紹介した html 形式データを

XML 化したものである．利用したデータは東京 23 区の住宅情報 (a)［6,267 件，約 1.5 MB］，店舗情報 (b)［605 件，約 100 KB］であり，2 入力のデータサイズには偏りがある．また，データ (c) への問合せは ADSL 事業者の提供する WEB 検索サービスに対して Java から POST アクセスを行い，出力 html 文書を処理した．1 件あたり 600〜900 ms 程度で完了する．

5.1.1 Join 演算の評価

(a)，(b) の Join 演算では住宅情報 1 つに対して複数の店舗情報を 1 対多結合しており，今回の例では，住宅情報に対応する店舗の有無のみを調べることができればよいため，ED および DD における MJoin のパラメータ（bounds）を 1 とした．これにより，住宅情報に対応する店舗が 1 つでも見つかった時点で最初の解を生成することができ，その解を取得した時点を first としている．また，参考資料として XML 文書の転送のみの場合と空のハンドラで RelaxNGCC を実行した場合にかかる時間を表 1 に併記している．

まず，プログラム記述面の評価を行う．ED 方式では，開始タグや終了タグの読み込みイベントごとのプログラム記述や入力データの木構造の階層管理などの低レベル処理をプログラマが行う必要がある．加えて，反応速度の速いプログラムの実装には，2 系統の XML データのマルチスレッドを利用した同時読み込みや Join 演算の部分では両スレッドの同期処理をプログラマ自身が行うことが必要となる．このため，2 つの表データの Join 演算を行うという本評価プログラムでさえ，レスポンスの良い実装を行うには，SAX を利用した場合には約 650 行のプログラム記述を必要とした．一方で，本提案方式では，3 章で紹介したように，木構造の階層管理やプログラムのマルチスレッド化，スレッド間の同期などはシステム側で行われるため，プログラマは簡単なプログラム記述で操作することができ，50 行程度ですべてのプログラムを記述できる．

次に，本提案方式の実行性能について評価する．レスポンスに関しては DD は Naive よりも大幅に改善されており，参考資料の RelaxNGCC のみの全実行時間と比べると，構文解析をすべて完了する前にレスポンスを返していることが分かる．今回のように店舗の有無だけを調べたい場合には，(b) の全入力データの読み込みを待つ必要がなく素早いレスポンスを返せる．また，全実行時間に関しては，DD(remove) では Naive に対して 8%程度の時間短縮が確認でき，メモリ使用量は 4.2 節で紹介した HashJoin の実装法により削減が図られている．本実装では同期やスレッド切替わりのコストが発生するが，入力データ受信待ち遅延隠蔽の効果や，メモリ使用量削減にともなう速度向上（GC 時間短縮やキャッシュヒット率向上）が期待できる．また，DD(remove) と ED を比較しても，スレッド切替えを抑えるためのバッファリングによってレスポンス時間は約 2 倍となったが，全実行時間は 2.5%程度の増加にとどまった．

5.1.2 全計算と Future 機構の評価

実験 2 は，全計算の評価であり，データ (c) への問合せ部分は ListMap 演算を利用して，実験 1 の Join の個々の結果に対して，ネットワーク問合せ結果を付加した．Naive では，実験 1 の Naive に加えネットワーク問合せを順次同期的に行った．Future0 では，実験 1 の DD(remove) に加えネットワーク問合せを同期的に行い，Future1，5，10 で非同期呼び出しを行い，最大並列問合せ数をそれぞれ 1，5，10 に設定した．また，本実験では冒頭の 25 件までについて順次問合せをし，先頭/5 件目/25 件目までの ADSL 情報がアクセス可能になった時間を測定している．ただし，問合せ時間に揺れがあるため，測定結果にも相当の誤差が予想される．

まず，Future0，Future1，Naive のレスポンスを比較すると，Future0，Future1 では，Naive よりも早い段階で外部問合せ可能なため良い結果を示している．また，並列問合せ数を増やすことで全実行時間は短縮傾向にあるが，レスポンスについては低下しており，相手サーバ側の負荷の問題やマルチスレッド起動によるオーバヘッドの問題と考えている．

5.2 大きな XML データの処理での評価

実験 3 は，大きな XML 文書どうしの Join 演算処理を通して，提案環境の評価を行う．入力は，XMLData Repository [9] 上にある TPC-H Benchmark [10] の単純な表データ：orders.xml［約 1.5 万件，約 5 MB］と lineitem.xml［約 6 万件，約 30 MB］であり，処理内容は，2 つの XML データの各要素にある ORDER_KEY の値が同一な要素を 1 対 1 に結合するものである．なお，それぞれの入力は ORDER_KEY でソート済みであったが，入力データの組合せが悪い場合を想定するため，あらかじめ lineitem.xml を逆順に並べ替えている．また，この実験では，バッファリングの効果を示すため，バッファリングをまったく行わない場合：DD(no-buf) の測定値も併記している．

測定結果を**表 2** に示す．実験結果より，DD(remove) では Naive に比べてレスポンスの大幅な向上や全実行時間の約 9%の短縮が図られた．また，メモリ使用量の面でも，Naive，DD(no-remove) に比べて約 65%の削

表 2 性能評価 2
Table 2 Performance results 2.

実験 3	first(ms)	total(ms)	mem(MB)
ED	1,607	9,574	13.7
DD(remove)	1,614	10,024	18.2
DD(no-remove)	1,897	10,259	54.8
DD(no-buf)	4,668	17,998	18.3
Naive	11,324	11,327	51.2

減が図られており，4.2 節で紹介した演算内部で不要な参照を持たない HashJoin テンプレートと remove 系アクセサを併用した場合の効果が確認できる．また，DD(remove) は ED と比べた場合でも，レスポンスは同等の性能が出ており，全実行時間に関しても約 5%の増加に抑えられている．次に，バッファリングの効果について．処理のバッファリングを行っている DD(remove) では，DD(no-buf) と比べてレスポンス/全実行時間が大幅に向上している．これは，バッファリングによりスレッド切替えコストを抑えることができたためであると考えられる．実際に構文解析部のスレッド切替え回数を測定したところ，DD(no-buf) では，約 60 万回のスレッド切替えが発生していたが，DD(remove) はスレッド切替え回数が 5 千回程度に抑えられていた．この結果から，マルチスレッドを利用した本実装方法でも極端なスレッド切替えを防止するためのバッファを導入することで，スレッド切替えによるオーバヘッドを吸収することができた．

5.3 再帰的な階層 XML データの処理での評価

実験 4 は，Michigan Benchmark [11] の QJ4 を使用する．入力 XML のスキーマは**図 9**（上）に示したもので，その特徴は，タグの先読み 1 個で分岐可能で，データは再帰的な階層を持つことがあげられる．入力 XML データは eNest 要素が 66,655 回，eOccasional 要素が 1,041 回出現し，XML の深さの最大が 16，サイズが約 45 MB のファイルである．処理内容は，eNest 要素を 1/4 確率でフィルタした集合（16,664 要素）と eOccasional 要素の集合（1,041 要素）に対して，特定の属性値が合致した要素どうしを 1 対 1 に結合する演算で，結果は 258 個生成される．

この実験では，データ階層が再帰的であり，表データのように簡単に Join 演算に対する入力集合を取得できない．このため，プログラマ自身が入力集合を取り出すクラスを作成する必要がある．本実験では，本システムが提供する入力集合生成クラス（IProducer）を利用した場合のプログラム記述面や性能面での効果について考察するため，入力集合の取り出し方を後に示す 3 通りの方法で行う．また，参考データとして，

```
<start c:class="Root"><ref name="A" c:field="root"/>...
<define name="A"  c:class="A">
  <element name="eNest">
    <attribute name="eUnique1"><... c:field="id"/>...
    <oneOrMore><choice  c:class="C" c:field="clist">
        <ref name="A"  c:field="a"/>
        <ref name="B"  c:field="b"/>
        <text  c:field="text"/> ····· (1)
    </choice></oneOrMore>
  </element>
</define>
<define name="B"  c:class="B">
  <element name="eOccasional">...</element>
</define>

/** マッピングされる XTree の型クラス */
abstract class Root implements XTree {
  abstract A getRoot();          // 他省略 }
abstract class A implements XTree {
  abstract Integer  getId();
  abstract XList<C> getClist(); // 他省略 }
abstract class B implements XTree {
  abstract Integer  getId();
  abstract String   getData(); // 他省略 }
abstract class C implements XTree {
  abstract A      getA();
  abstract B      getB();
  abstract String getText();   // 他省略 }
```

図 9 実験 5：スキーマおよびプログラム
Fig. 9 Case 5: schema and program code.

代表的な既存のデータバインディングツールである Relaxer [12]（version: 1.1b）を利用したときの測定値も併記している．

まず，評価プログラムの実装について．DD/Naive 方式では，入力 XML は図 9（上）のマッピング指定を行い，図 9（下）に示す木構造にマッピングした．DD での Join 演算への入力要素の取り込み方法は，下の 3 通りの方法で行った．

- DD(ipro)：木構造を前順走査して A，B の入力要素の取り出しを行う IProducer⟨A,B⟩ 型のクラスを作成し，それを演算クラスへ渡す．
- DD(naive-trav)：最初に木構造から A，B の入力集合の取り出しを行い，演算クラスへ各集合へのアクセサを渡す．
- DD(opt-trav)：要求駆動に基づいて，木構造を走査するプログラムを自作した場合［詳細は後述］．

また，Naive では DD(naive-trav) と同様，Relaxer では DD(ipro) と同様の方法で Join 演算への入力要素の取り込み部を実装した．

次に，プログラム記述面の評価について．DD(ipro) と DD(naive-trav) では，木構造をトラバースし各要素を探し出すための記述のみでよく，プログラム記述は比較的簡単である．たとえば，DD(ipro) の入力集合作成クラスの概要は**図 10** に示したものであり，20 行程度のプログラムで実装が可能である．一方，DD(opt-trav) では DD(naive-trav) と同様に，A，B の入力集合クラスを演算クラスに渡すが，その入力集合クラス

```
class TreeVisitor extends InputProducer<A,B> {
  // コンストラクタなど省略
  void run(){
     super.putInput0( rootNode );
     visit( rootNode.getClist() );
     super.putEnd0();
     super.putEnd1();
  }
  void visit(XListC〉 nodes){
    for(XIterator<C> it=nodes.iterator(); it.hasNext();){
       C c = it.next();
       if( c.getA()!= null ){
          super.putInput0( c.getA() );
          visit( c.getClist() );
       } else if( c.getB()!= null )
          super.putInput1( c.getB() );
    }
  }
}
```

図 10 Join 演算への入力生成クラス作成例
Fig. 10 `IProducer` code (case 5).

表 3 性能評価 3
Table 3 Performance results 3.

実験 4	first(ms)	total(ms)	mem(MB)
ED	30	13,831	13.2
DD(ipro)	38	13,847	14.6
DD(naive-trav)	14,023	14,049	14.6
DD(opt-trav)	256	14,013	15.0
Naive	14,302	14,302	118.2
Relaxer（参考）	18,586	18,672	399.5

内部では要求駆動に次要素を取り出すアクセサの実装を行っている．アクセサ内部の実装では，A，B それぞれの次要素へのアクセス要求に応じて，次要素が見つかるまで木構造をトラバースを進め，見つかった時点で値を返している．このため DD(opt-trav) のプログラムにおいては，各時点でどの部分木からトラバースを開始すべきかについてやどこまでトラバース処理が終了したのかについてプログラマが明示的に管理する必要があり，面倒なプログラム記述が必要となる．

次に，プログラム性能面の評価について．測定結果を**表 3** に示す．まず，Naive と DD(naive-trav) では，Join 演算を行う前に全入力集合の作成を行っており，Join 処理の開始前に全データの構文解析が必要となるためレスポンスは遅い．一方で，DD(ipro) と DD(opt-trav) では，入力集合の作成は要求駆動で行われており，同等の素早いレスポンスを返していることが分かる．このように，IProducer を利用した DD(ipro) では簡単なプログラム記述で DD(opt-trav) と同等の性能を出せることが分かった．最後に，Relaxer では，DD(ipro) と同様，入力集合の作成は要求駆動で行われているが，最初に木構造すべてを構築しており，反応速度向上は見られない．

6. 関連研究

本章では，各種の遅延隠蔽を目指した処理系や研究について紹介する．

DOM Parsers： MSXML[1] では，入力 XML 文書の非同期読み込みを提供しており，読み込みの待ち時間に他の処理を行うことができる．また Xerces[2] では，構文解析処理の後にデータを木構造に展開せずに，ユーザアクセスに応じて必要な部分木を遅延作成するモードを提供している．また構文解析処理の一部を遅延評価することにより遅延隠蔽を図った研究も行われている[3]．この研究では，構文解析処理を木構造の階層を抽出する処理（preprocessing）とタグ名，属性値などを取り出す処理（progressive parsing）を分離し，progressive parsing を対応部分木アクセス時に行う．ただし，上記のどの研究も，少なくとも入力 XML 文書をすべて読み込むまで木構造にアクセスできず，パイプライン的なデータ処理は実現できない．

Web Services Platform： 多くの実装では，個々の Web Service の非同期呼び出しをサポートしており，ネットワーク問合せ処理の遅延隠蔽が可能である．ただし，SOAP メッセージ受信部の多くは DOM を使って実装されており，SOAP 上のデータとして大きな XML データが送られてきた場合，構文解析処理待ち遅延などが問題となる．データ処理部を本データバインディングツールで実装を行うと遅延隠蔽も可能で，処理の効率化が図られると考える．今後，このような実装も検討していきたい．

XQuery Processor： Tukwila[4] はネットワーク越しの XML データ処理を対象とした XQuery 処理系で，処理系内部で計算順序の変更を行い，処理をパイプライン的に実行させることで遅延隠蔽を目指している．実装では入力 XML データを tuple で管理し，リレーショナル DB で行われているパイプライン的に動作する Join 演算などを，ストリーム入力される XML データに対して適応している．ただし，5.3 節で紹介したような再帰的なデータ構造をパイプライン的に処理することはできない．

7. まとめと今後の課題

本論文では，ネットワーク上の XML データの効率的な処理を目指した要求駆動型 XML 計算環境 *Nanafusi* のデザイン/実装法について述べた．また，大きな/再帰的な XML データを扱った複数のプログラムを通してシステムの性能評価を行った．評価プログラムおいて，従来手法に比べて，反応速度の大幅な向上

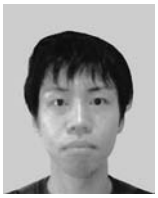
に加えて，全実行時間も最大約 9 %の向上が確認できた．

今後は，ネットワークを越えて多段に演算がつながった処理などより複雑なデータ処理に対しての有効性についても検討を行いたい．

参 考 文 献

1) Microsoft XML Core Services (MSXML) 4.0.
2) Xerces Java Parser. Apache XML Project. http://xml.apache.org/xerces-j/
3) Noga, M.L., Schott, S. and Lowe, W.: Lazy XML Processing, *ACM DocEng'02*, ACM Press (Nov. 2002).
4) Ives, Z.G., Halevy, A.Y. and Weld. D.S.: Integrating Network-Bound XML Data, *IEEE Data Engineering Bulletin* (2001).
5) RELAX NG. http://www.oasis-open.org/committees/relax-ng/
6) 新村健治，鎌田十三郎，田村直之：ストリーム処理に適した効率的 XML データバインディングツールの提案，日本ソフトウェア科学会第 19 回大会論文集 (2002).
7) RelaxNGCC. http://sourceforge.net/projects/relaxngcc/
8) Benchmark of Nanafusi. http://www.cs26.scitec.kobe-u.ac.jp/~pl/nanafusi/benchmark.html
9) XMLData Repository. http://www.cs. washington.edu/research/xmldatasets/
10) TPC Transaction Processing Performance Council. http://www.tpc.org/tpch/
11) The Michigan Benchmark. http://www.eecs.umich.edu/db/mbench/
12) Relaxer. http://www.asahi-net.or.jp/~dp8t-asm/java/tools/Relaxer/

(平成 16 年 1 月 31 日受付)
(平成 16 年 4 月 27 日採録)



新村 健治

1978 年生．2002 年神戸大学工学部情報知能工学科卒業．2004 年同大学大学院自然科学研究科情報知能工学専攻修了．2004 年 4 月より株式会社リコー．Web サービス技術や分散処理等に興味を持つ．

山中 真和

1979 年生．2003 年神戸大学工学部情報知能工学科卒業．現在，同大学大学院自然科学研究科情報知能工学専攻在学中．XML 技術や Web アプリケーション等に興味を持つ．

鎌田十三郎（正会員）

1970 年生．1993 年東京大学理学部情報科学科卒業．1995 年同大学大学院理学系研究科情報科学専攻修士課程修了．1998 年同博士課程単位修得退学．1996 年〜1998 年日本学術振興会特別研究員（東京大学）．1998 年より神戸大学工学部助手．博士（工学）．並列・分散処理，言語処理系等に興味を持つ．日本ソフトウェア科学会，ACM 各会員．