情報処理学会論文誌　データベース　Vol. 3　No. 1　64–81 (Mar. 2010)

ノードごとのアクセス統計値に基づくXMLセマンティックキャッシュ管理

朴　　大一[†1]　遠山元道[†2]

本論文では，XML（Extensible Markup Language）データにおける新たなセマンティックキャッシュ管理手法を提案する．セマンティックキャッシュとはクエリとそれに対する結果の組をキャッシュの対象とする手法である．提案する管理手法はユーザクエリに対するキャッシュのヒット率を向上させ，データの再利用度を高め，クエリ応答遅延時間を短縮させる．従来では，キャッシュ管理手法はスペースを確保するためにキャッシュされたクエリの結果をすべて削除していた．しかし，新しいクエリがキャッシュされた結果と包含関係であることが頻繁にある場合には，これではキャッシュの空間を効率的に利用するのに不十分である．本論文で提案する管理手法は，XML データを構成する要素に対するセマンティック情報—ユーザアクセス統計値—をノードごとに格納するものである．これはキャッシュに格納される XML データの部分木に対するメタデータになり，セマンティックキャッシュを構成する基本単位であるセグメントの統合によって複数のセグメントを 1 つにまとめることで，断片化されたキャッシュの再構成に用いられる．これらにより従来の XML セマンティックキャッシングより有用性があることが期待できる．実験により提案する管理手法がヒット率や応答遅延時間など効率性に優れていることを示す．

XML Semantic Cache Management Based on Node-wise Access Statistics

Daeil Park[†1] and Motomichi Toyama[†2]

In this paper, we propose a novel semantic cache management method. Semantic caches are both semantic descriptions and results of previously cached queries. Applying this method on a cache of XML data server, the cache hit ratio becomes higher than that of any other conventional method. Therefore, we get an improved query response time and high quality of reuse to cache space. To manage the space of the cache, traditional replacement strategies remove a complete cached query and the result data when space needs to be freed. This coarse granularity method however does not have good performance because new queries often have containment relationship with cached queries. The proposed Semantic Cache accumulates semantics - user access statistics - for elements that compose the XML data in each node. They present meta data about semantic regions - partial tree of the XML data - in the cache. These are used to reassemble the fragmented cache by intergrating and reorganizing the segments used as units in the esatablishment of the cache. We expect the proposed method to achieve better performence than any other conventional method. We also present experimental results showing the effectiveness of our proposal.

1. はじめに

XML[4]（Extensible Markup Language）はビジネスや科学の分野における半構造データの表現やデータ交換のためのディファクトスタンダードの言語である．XML の重要性と有用性の高まりにともない，XML データに対する様々な格納手法，索引手法[8),9)]や問合せ手法が研究されてきた．本論文では，XML データにおける新しいキャッシュ管理手法を提案する．

問合せ処理を行う際，頻繁に使われているクエリの結果をキャッシュに格納しそれを再び用いるのは問合せ最適化技術の 1 つである．従来，関係データベース環境でキャッシュを管理する場合，キャッシュ内部のセグメントとしてページ単位[13)]やタプル単位[5)]で管理を行うことが多かった．さらに，クエリ結果間の包含関係に基づいたセマンティックキャッシュの研究もあげられる[14),16)]．また通常キャッシュテーブルを管理する際には，LRU（Least Recently Used），LFU（Least Frequently Used）やその類似アルゴリズムを用いて管理を行うことが一般的である．

一方 XML の分野では，キャッシュ側においてクエリ（XPath[2]，XQuery[17)]）間の包含関係の判定に基づいて処理を行った研究をいくつかあげることができる[1),3),11),18)]．XML クエリ間の包含関係の判定[6)]については数多く研究されてきたが，その概念をキャッシュ側でクエリ処理に導入した研究はごく一部である．それらは包含関係に基づいてクエリに対するキャッシュでの対応についての研究が主流である．

さらに，Mandhani ら[3)]はクエリ処理を行うとき，応答可能性（answerability）を提案

†1 慶應義塾大学大学院理工学研究科
Graduate School of Science and Technology, Keio University
†2 慶應義塾大学理工学部
Faculty of Science and Technology, Keio University

し，その手法を実体化ビューで用いる研究を行った．これは最初にビューキャッシュ（実体化ビュー）をウォームアップするものである．一般的にはウォームアップワークロードにあるクエリをそのまま抽出してウォームアップする手法をとるが，Mandhani らはそれとは異なる手法を提案する．それは，ウォームアップワークロードを代表する最適化されたクエリパターンを抽出し，それを用いてビューキャッシュをウォームアップするものである．それによって初期化されたビューキャッシュはクエリ間に包含関係による重複は生じないが，クエリの結果間に部分的に重複が生じる場合がある．さらにウォームアップワークロードからクエリパターンを抽出し一括処理でビューキャッシュを生成するのでウォームアップワークロード（ユーザからのクエリパターン）が変わるたび，ビューキャッシュに対する初期化を行わなければならない．これはクエリパターンが頻繁に変わる環境では適合しないと考えられる．

そして，ACE-XQ[11] では XQuery とその FLOWR 表現に使われる XPath とその結果についてキャッシュ管理を行う．その際，XQuery 間において共通で使われる XPath や XPath 間の包含関係の判定によって以前キャッシュされた結果を再利用することになる．この場合もキャッシュのクエリ結果間に重複の問題が生じる．さらに，キャッシュの交換を行う際は以下の問題があげられる．

たとえば，あるキャッシュレコードに/a/b/c が格納されているとする．/a/b/c/f というクエリに対して/a/b/c から結果が得られる．しかし，/a/b/c/f にアクセスされるのにもかかわらず，アクセスカウント（ヒットカウント）は/a/b/c のレコードにカウントされる．結果で得られる/a/b/c/f の部分木が持つサイズが/a/b/c サイズの 1 割とすると 9 割のスペースが無駄になると考えられる．そして，/a/b/c/f も格納するとなると今度は/a/b/c と/a/b/c/f の結果間の重複が問題になる．もう 1 つ予想される問題には断片化があげられる．たとえば，キャッシュレコードに/a/b/c，/a/b/d，/a/b/e が格納されると仮定する．3 つのクエリ結果の合計が/a/b の結果サイズとほぼ同じだとすると，/a/b 1 つを格納する方が効率的である．要するに，3 つのレコードを 1 つにまとめればキャッシュの全体的なレコード数が減り，スペースの節約が期待できるのである．さらに，潜在的に生じるクエリとして/a/b[d]/c のような枝分かれクエリにも対応できる．しかし，ここで重要なのはどちらを格納するのかを見極めることである．そのためには，ユーザからのクエリ（XPath）がアクセスされるノード情報だけではなく，そのノードに至るまでの経路ノードの情報を集める必要があると思われる．

そこで，本研究ではその情報を集める手法と集めた情報から新しいアクセスカウントであるセマンティックアクセスカウント（以下 SAC）を提案する．SAC とは XML 文書の上にある各ノードのその兄弟ノードに対する占有率を表す．そして SAC を用いたセマンティックキャッシュマネジメントシステムもあわせて提案する．さらに，提案するシステムによるキャッシュ中のデータ重複問題の改善，キャッシュの有効性向上を図る．

本論文の構成は次のとおりである．2 章では，前提事項として，既存のキャッシング手法とユーザクエリとキャッシュ間の応答可能性について述べる．3 章では，XML データに適合する新たなキャッシュ管理手法について記述する．4 章では，本研究のキャッシュシステムを構成するそれぞれのモジュールについて説明を行い，5 章では，実験と実験結果について述べる．最後に，6 章でまとめと今後の課題を述べる．

2. 前提事項と関連研究

2.1 従来のキャッシング手法

従来，WWW や分散環境で問合せ最適化技術のために利用されていたキャッシュ管理手法にはタプルキャッシング，ページキャッシングやセマンティックキャッシングなどがあった．

ページキャッシングモデル[5),13)] は伝送単位とキャッシュアイテムがディスクページである．クエリに対するセマンティック情報が保持されないため，キャッシュ管理時にユーザ統計値を利用することはできない．要求されたページがキャッシュにないとディスクから求める．キャッシュ交換アルゴリズムとして LRU や MRU で管理される．

タプルキャッシングは多くの点においてページキャッシングと類似しているが，キャッシュはページではなく，関係データベースのタプル単位で構成される．通常，タプルのサイズはページより小さいため頻繁に行うメッセージのやりとり費用が生じる場合がある．その無駄を避けるため，多数のタプルをブロックにまとめて送受信を行うのが普通である．キャッシュ交換アルゴリズムとして LRU や MRU で管理される．

セマンティックキャッシングは様々なところで研究[1),3),7),10),12),16)] されている．centralized システムではディスクアクセスを縮める目的で，クライアント・サーバシステムではネットワークの競合とシステムスケーラビリティを向上させる目的で用いられる．キャッシュにはクエリのセマンティック情報とその結果が保持される．つまり，キャッシュ再利用の粒度がクエリレベルにあり，クエリとキャッシュセグメント各々の間に包含関係の判定が不可欠な技術要素であるといえる．キャッシュをクエリ単位で扱うため，個々のセグメントサイズに柔軟性があるという特徴を持つ．

2.2　XML クエリに基づくセマンティックキャッシュ

本研究では，クエリ処理時に既存の研究である応答可能性を用いる．XPath や XQuery セマンティックキャッシュを実現するためには，キャッシュからクエリの結果を求めることができるのかというクエリ・キャッシュ応答可能性の判定が重要である．クエリ・キャッシュ応答可能性は包含関係とは異なる．たとえば，キャッシュビューを V，クエリを Q とすると，V = /a/b と Q = /a[x]/b の関係は，Q は V に含まれるが V からの応答可能性の判断には情報が不十分である．なぜならば，V の結果はノード b をルートノードとした XML 部分木であり，その XML 部分木から結果を求めようとする Q の検索条件 '[x]' が b をルートノードにする XML 部分木の範囲外にあるからである．

Mandhani ら[3] の研究では，XML セマンティックキャッシュとして実体化 XPath ビューを提案している．あるときセマンティックキャッシュが含むビューの状態が V = {V1,V2,V3···} とすると，新しいクエリ Q に対して V で検索を行う．Q と V は XPath 表現である．V の結果に対して Q の結果を得られるクエリ C が存在すれば Q は V から結果を求めることができる．C・V = Q を満たすクエリ C を composing クエリ（CQ）という．クエリ・キャッシュ応答可能性[3] は以下のように定義される．

定義 1 Query Axis と Query Depth
Query Axis（クエリ軸）とはルートノードから結果ノードまでの path を表す．その path の上にあるノードを axis node（軸ノード），それ以外は predicate node（述語ノード）と呼ぶ．Query depth（クエリの深さ）は axis node（軸ノード）の数で表す．

定義 2 Prefix（Q,k）
k 番目の axis node（軸ノード）でクエリ Q を枝切りして得るものを指す．ただし，k 番目の axis node（軸ノード）は含むが，その predicate（述語）は含まない．

定義 3 Preds（Q,k）
クエリ Q の k 番目 axis node（軸ノード）の predicate（述語）集合を表す．

定義 4 ConPreds（Q,k）
クエリ Q の k 番目の axis node（軸ノード）の predicate を含むすべての predicate（述語）集合を表す．

XPath の例をあげながら上述の定義を説明する．たとえば，クエリ Q = a[v]/b[x/y//z]//c[z] とする．まず，**定義 1** による Query Axis と Query Depth はそれぞれ a[v]/b[x/y//z]//c と 3 である．そして，**定義 2** で k が 2 とすると Prefix（Q,2）は a[v]/b となる．そのときに**定義 3** の Preds（Q,2）は {[x/y//z]} になる．そして，**定義 4** ConPreds（Q,k）とは，Preds（Q,2）から求めた [x/y//z] を含むすべての predicate の集合であるため，{[x],[x/y],[x/y//z],[x//y],[x//y//z],[x//z],[.//y],[.//y//z],[.//z]} となる．

Mandhani ら[3] は上記の定義 1 から定義 4 をまとめて以下の 2 つの条件を満たせばキャッシュからクエリの結果が得ることができるクエリビュー応答可能性を提案した．

（1）　Prefix（V,k）= Prefix（Q,k），k は V の depth
（2）　Preds（V,k）⊆ ConPreds（Q,k）

Mandhani らは実体化ビューの定義によって生成したビューをビューキャッシュと定義して，応答可能性を用いてクエリとビューキャッシュ間のクエリ処理について述べた．一方，本研究では 2 つの手法を提案する．1 つはユーザからのクエリとその結果に対する動的なキャッシュテーブルを生成して，そこで行われる削除や挿入による更新手法であり，次にレコード間に生じる重複問題，断片化問題に対する手法である．以上を前提にしたキャッシュ管理手法について 3 章で述べる．

3.　ノードセマンティックキャッシング

XML データを構成する要素（element）は様々なノードで構成されている．ノードとは意味を持つ最小単位である．XML データはノードとノードの間に親子，兄弟や子孫と先祖という構造的な従属性を持つことが関係データのタプルと異なるところである．そこで，関連があるノード間の構造的特徴を活かしたセマンティックキャッシングがより重要になる．従来の研究では，構造的な特徴に基づく XPath 間の包含関係に注目したクエリ処理に対する最適化がさかんであった．同時にそれはキャッシュ側でクエリ処理にも利用されていた．しかし，包含関係に基いたクエリ処理はクエリに対する結果ノードのみにアクセスの意味を持たせる．その結果，従来 XML に関するセマンティックキャッシュでは時間が経過するほど，次第にキャッシュセグメント間の重複が多くなる．このような現状に基づく XML データに適したキャッシュ管理アルゴリズムが必要である．本章では XML データの構造的な特徴を反映したノードアクセス統計値に基づくセマンティックキャッシュ管理手法について述べる．

3.1　データ構造

提案するセマンティックキャッシュ管理手法を実現するためには，ノードごとにユーザのアクセス統計値を蓄積する必要がある．キャッシュされたクエリとその結果に対するセマンティック情報を保持し，ノードことにアクセス情報を持つことにより，より多くの情報に基づく評価関数が使用できる．これにキャッシュの交換管理のために使われてきた LRU と MRU を加えることもできる．

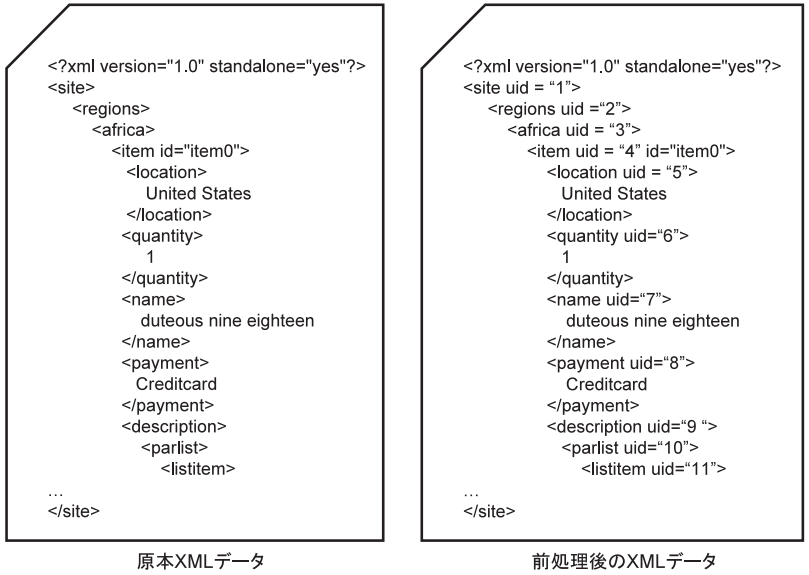

図 1　XML データ
Fig. 1　XML data.

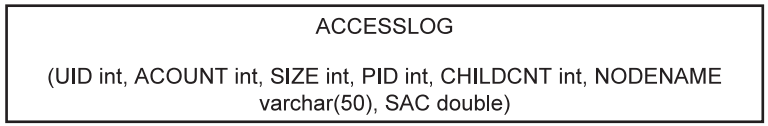

図 2　ACCESSLOG テーブルのスキーマ
Fig. 2　ACCESSLOG table schema.

ノードごとのアクセス情報を格納するためにノードを 1 つ 1 つ区別しなければならない．そこで原本 XML データに対する前処理が必要になる．**図 1** のように原本 XML データを SAX パーサを用いて要素ごとに分けイベントが発生するたびに要素の第 1 番目属性としてグローバル番号を振っていく．番号の振り方は深さ優先順位に従う．新たな XML データに対して，ノードに関する様々なメタ情報を格納するために**図 2** のような ACCESSLOG テーブルを用意する．ACCESSLOG テーブルは，SAX パーサを用いて XML データをデータベースへローディングして作られるノードの集合である．これは DTD グラフやスキーマと異なり，ノードごとにセマンティック情報を保持することができる．ノード間はユニークなノード ID で区別できる．ユーザからのクエリが XML データのあるノードにアクセスされるたびに，そのノードと同一のノード ID を持つ ACCESSLOG テーブルのレコードにアクセス情報を格納していく．

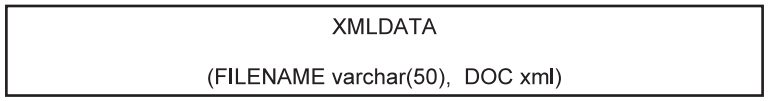

図 3　XMLDATA テーブルのスキーマ
Fig. 3　XMLDATA table schema.

ACCESSLOG テーブルは以下の属性で構成される．各ノードを区別するユニークな ID が UID であり，親ノードの UID を表すのが PID である．アクセスされるたびに該当レコードのみに 1 ずつヒットカウントを格納（従来のヒットカウント法）するための属性が ACOUNT であり，そのノードをルートとした XML 部分木の大きさがサイズ属性である．CHILDCNT はそのノードが持つ子ノードの数を表す．ACCESSLOG テーブル属性の 1 つとして，従来のヒットカウント - アクセスカウントの応用である SAC を取り入れる．SAC については次節で詳しく述べる．キャッシュマネージャは ACCESSLOG テーブルに格納された情報を参照しながらキャッシュ管理を行う．次に，本来の XML データが格納される XMLDATA テーブルと実際のクエリとその結果を格納するキャッシュテーブルについて述べる．XMLDATA テーブルは**図 3** のように，FILENAME と DOC の 2 つの属性で構成される．DOC は XML の原本データである．本研究で使うキャッシュテーブル群を**図 4** に示す．CACHE，PREFIX，PREDICATE 3 つのテーブルで構成される．

CACHE テーブルの属性 QUERY はユーザが検索で用いる XPath である．RESULT は XMLDATA テーブルから QUERY で得られた XML データ型の XML 部分木を格納する．その結果の大きさを SIZE で表す．ACCESSTIME は timestamp 型で本研究ではまだ利用されていないが，今後 LRU アルゴリズムなどを用いて時間的局所性を反映させるための属性値である．ACOUNT 属性は ACCESSLOG テーブルのそれと同様である．PREFIX テーブルの PREFIX 属性は定義 2 の Prefix（Q,k）から求める．PREDICATE テーブルの PREDS と ALLPREDS 属性はそれぞれ定義 3 の Preds（Q,k）と定義 4 の ConPreds（Q,k）から求める．

以上述べたデータ構造に基づいてクエリの検索過程について説明する．

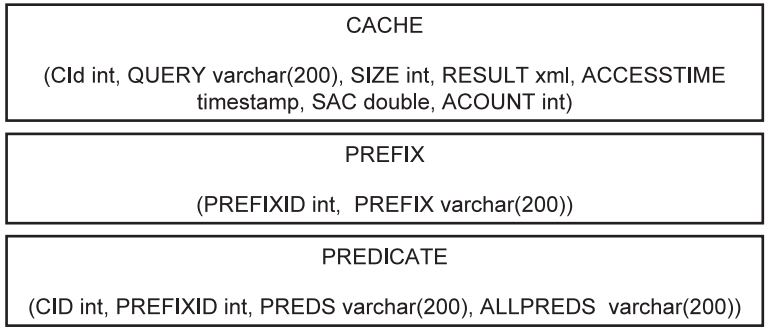

図 4　キャッシュ関連テーブルのスキーマ
Fig. 4　Schema of concerned cache.

3.2　データ構造に基づくクエリ検索

ユーザからのクエリ（XPath）は応答可能性の可否によってキャッシュルックアップ（**図 5**）とデータベース検索に分かれる．最初のクエリは，応答可能性判定の前に CACHE テーブルの QUERY 属性値と比較を行い，同一クエリがあるか判定を行う．これを complete マッチングと呼ぶ．見つからなかった場合は応答可能性判定を行う．キャッシュルックアップアルゴリズムは図 5 に示す．応答可能性を満たしたクエリに関しては，クエリの再作成を行わなければならない（getQueryRewriter()）．なぜなら CACHE テーブルの RESULT フィールドには，⟨Root⟩⟨c⟩ . . . ⟨/c⟩⟨/Root⟩ ようにダミールートノードが入っているからである．それは，検索による XMLDATA テーブルからの結果値としてルートノードを持たないノードリストを返すことがあるからである．たとえば，/a/b/c[g][h]（ユーザクエリ）に対して，CACHE テーブルの prefix が/a/b/c だと reWriQuery（14 行目）は/Root/c[g][h] になる．そして，再作成されたクエリを用いて最後に結果を求める．

キャッシュルックアップの結果が空値で戻った場合，データベース検索を行う．本研究では，プロトタイプとして IBM DB2 Express-C [19] を利用したため，DB2 で XML データ検索について提供している db2-fn:xmlcolumn() 関数を用いて以下の構文で検索を行う．

$$executeQuery(XQuerydb2-fn:xmlcolumn(XMLDATA.DOC)+Uquery);$$

キャッシュルックアップとデータベース検索のどちらからアクセスされるノードに対しても，ACCESSLOG テーブルにアクセス情報を集積する作業が必要になる．更新された ACCESSLOG との整合性を保つために CACHE テーブルに更新を行う．こうして更新された統計値を基に検索処理後，キャッシュフル時のみキャッシュマネージャが CACHE テーブルに対し交換作業を行う．キャッシュ再編成を実行するタイミングについては工夫が必要になる．上述した作業はデータベースサーバのバックエンドで行われる．次節では，CACHE テーブルに対する交換と再編成の作業時に基準値となる SAC の概念と更新方法について述べる．

```
Algorithm cacheLookUp(Uquery)
Input: a user query Uquery
output: an XML partial tree Pxml
▪complete match
01:  Pxml =  executeQuery(select  C.result
02:                              from   CACHE C
03:                              where C.query = Uquery)
04:  if (Pxml  != null)
05:      Return Pxml
06:
▪answerability check
07:  else if (Pxml == null)
08:      <C.result, P.prefix> = executeQuery( select C.result, P.prefix
09:                                 from CACHE C, PREFIX P, PREDICATE D
10:                                 where P.prefix = Prefix(Uquery, k) and
11:                                 P. prefixId = D. prefixId and
12:                                 (D.preds in NULL or  D.preds in (ConPreds(Uquery, k)))
13:                                 D.cId = C.cId
14:      reWriQuery   = getQueryRewriter(Uquery, P.preifix)
15:      Pxml = executeQuery(XQuery db2-fn:xmlcolumn(C.result) + reWriQuery)
16:      Return Pxml
17:  end if
```

図 5　キャッシュルックアップアルゴリズム
Fig. 5　Cache lookup algorithm.

3.3　セマンティックアクセスカウント（SAC）

本節では，ACCESSLOG テーブルの属性として用いられる SAC について従来のアクセスカウントと比較して述べる．まず，従来のアクセントカウントを用いた場合に生じる現象について述べる．**図 6** と**図 7** を用いて説明を行う．XPath（/a/b/c）によってノードにアクセスが行われるとノード c に対しアクセス情報としてヒットカウントが上がる．キャッシュテーブルは図 6 の i) のようになる．残りの属性値は説明簡略化のため省略する．それから/a/b/c/g に対するアクセスがキャッシュに来ると/a/b/c レコードから結果が得られるため，その分/a/b/c にアクセスカウントが上がる（図 6 の ii)）．ノード g にアクセスされるにもかかわらず，/a/b/c から結果を求めたので/a/b/c のカウントが上がる．その後/a/b/c/g

i)

QUERY	SIZE	RESULT	ACOUNT	…
/a/b/c	…	…	1	…

ii)

QUERY	SIZE	RESULT	ACOUNT	…
/a/b/c	…	…	2	…

iii)

QUERY	SIZE	RESULT	ACOUNT	…
/a/b/c	…	…	1	…
/a/b/c/g	…	…	1	…

図 6　従来キャッシュテーブルの例
Fig. 6　Example of cache table.

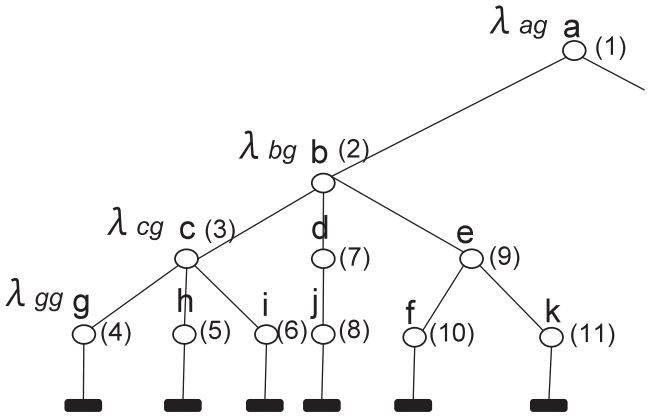

図 7　XML データの一部に対する木構造
Fig. 7　Tree structured XML data fragment.

UID	PID	CHILDCNT	SAC	…
1	0	2	0	
2	1	3	0	
3	2	3	0	
4	3	0	0	
5	3	0	0	
…				

図 8　図 7 に対する ACCESSLOG テーブル
Fig. 8　ACCESSLOG table about Fig. 7.

が来る場合，そのつど/a/b/c を持つことによって空間スペースを消費してしまうことになる．それは，/a/b/c の結果値の中に余計にノード h とノード i が含まれるからである．また，重複を許容しキャッシュテーブルにレコード/a/b/c と/a/b/c/g を両方持つことになる場合もある（図 6 の iii)）．これは/a/b/c と/a/b/c/g の間は応答可能性を持つ包含関係であるため，このような重複によるスペースの消費が起こるものである．以上の問題点を解決しより高度なキャッシュ管理を行うために SAC を提案する．

図 7 と**図 8** を用いて SAC に関する説明を行う．図 7 は全体 XML データに対して一部を木構造で表している．そしてそれに対する ACCESSLOG テーブルを図 8 に示す．ここで，SAC を定義する．

定義 5 セマンティックアクセスカウント（SAC）
あるノードを n とし，n に対して先祖関係であるノードの集合を $M = \{m_1, m_2, \ldots, m_i\}$ と仮定する．そこで，ノード n にアクセスが行われた際に，ノード n と M の元素ノードに対して，下記の式で導き出される $\lambda_{m_i n}$ を SAC とする．

$$\lambda_{m_i n} = \prod_{h=1}^{H} \frac{1}{\beta_h}, \lambda_{nn} = 1 \tag{1}$$

H：m_i の高さ $-n$ の高さ
β_h：ノード n と h レベル差にある先祖ノードが持つ子ノード数

たとえば ‘/a/b/c/g’ というクエリに対し従来のアクセスカウントでは，アクセス対象になるノード ‘g’ の集合のみヒットカウントが加えられるが，SAC では ‘g’ に至るまでの各ノードにも式 (1) の計算方法によって値を求めてノードごとに格納するという点が異なる．ノード ‘g’ に対してノード ‘c’ と ‘b’ と ‘a’ はそれぞれ親・先祖関係があるため，SAC は XML データノード間にある構造的密接性を反映させる手法だといえる．つまり，各々のノードが XML データの中でどれぐらいの領域を占有しているかを反映した応用アクセスカウントである．式 (1) に用いられる β_h 値はアクセスされるノードとの高さの差が h である先祖ノードが持つ子ノードの数のことをいう．

式による計算過程がアルゴリズム**図 9** によって ACCESSLOG テーブルのルートレコードに至るまで遡りながら SAC 属性の更新が行われる．ACCESSLOG テーブルに更新が行われると CACHE にもその情報を反映しなければならない．それについては**図 10** に示す．

データベース検索やキャッシュ検索のどちらかで検索が行われてもアクセスされるノードの UID を用いて ACCESSLOG にあるメタ情報が更新される．そのうち SAC に対する更

```
Algorithm updateParentSAC(uid)
input:  an accessed node's id  uid
 01:  pid = executeQuery(select  PID  from   ACCESSLOG   where UID = uid)
 02:  if (pid != 0)
 03:       cnt = executeQuery(select  CHILDCNT  from  ACCESSLOG where UID = pid)
 04:       sac = 1/cnt
 05:       executeUpdate(update ACCESSLOG set SAC = SAC + sac  where UID = pid)
 06:       updateParentSAC(pid)
 07:  end if
 08:  Return
```

図 9　ACCESSLOG 側の SAC 更新アルゴリズム
Fig. 9　Update SAC algorithm at ACCESSLOG.

```
Algorithm updateCacheSAC(query)
input a query
 01:  result = executeQuery(select  RESULT  from  CACHE
 02:                              where QUERY = query)
 03:  uid  = getUid(result)
 04:  sac = executeQuery(select SAC from  ACCESSLOG where UID = uid)
 05   executeUpdate(update CACHE set SAC = sac  where QUERY = query)
```

図 10　CACHE 側の SAC 更新アルゴリズム
Fig. 10　Update SAC algorithm at CACHE.

新について簡単な例をあげる．図 7 に対してクエリ/a/b/c/g による検索を行うと，ノード g をルートとした結果（XML 部分木）とそのノードの UID が得られる．得られた UID（4）を引数にし，アルゴリズムによる ACCESSLOG テーブルの更新を行う．そのとき CACHE テーブルと ACCESSLOG テーブルは**図 11** の状態になる．説明の便宜のために属性の一部のみ表す．

‘/a/b/c/g’ というクエリが与えられる場合，λ による計算が行われその値をそれぞれの親ノードの SAC に加えることで更新を行う．

計算例）‘/a/b/c/g’ に対する SAC の計算

（1）　`/a/b/c (UID = 3)`, $\lambda_{gg} = 1$, $\lambda_{cg} = 1/$ (1+1+1) * 1 = 0.33

（2）　`/a/b (UID = 2)`, $\lambda_{bg} = 1$ / (1+1+1) * 0.33 = 0.11

（3）　`/a (UID = 1)` $\lambda_{ag} = 1/(1+1)$* 0.11 = 0.06

その後再びクエリ/a/b/c/g が来ると，同一計算により更新後の状態は**図 12** のようになる．そして/a/b/c/h というクエリ（UID = 5）が来ると，それぞれの SAC 値に λ_{gg}（/a/b/c/h），λ_{cg}（/a/b/c），λ_{bg}（/a/b），λ_{ag}（/a）の計算値を加えて**図 13** のように更新

UID	PID	CHILDCNT	SAC	…
1	0	2	0.06	
2	1	3	0.11	
3	2	3	0.33	
4	3	0	1	
5	3	0	0	
…				

QUERY	SAC	ACOUNT	…
/a/b/c/g	1	1	

図 11　/a/b/c/g による更新後の状態
Fig. 11　CACHE and ACCESSLOG table state after /a/b/c/g inserted.

UID	PID	CHILDCNT	SAC	…
1	0	2	0.12	
2	1	3	0.22	
3	2	3	0.66	
4	3	0	2	
5	3	0	0	
…				

QUERY	SAC	ACOUNT	…
/a/b/c/g	2	2	

図 12　2 度目の/a/b/c/g による更新後の状態
Fig. 12　CACHE and ACCESSLOG table state after 2nd /a/b/c/g.

UID	PID	CHILDCNT	SAC	…
1	0	2	0.18	
2	1	3	0.33	
3	2	3	0.99	
4	3	0	2	
5	3	0	1	
…				

QUERY	SAC	ACOUNT	…
/a/b/c/g	2	2	
/a/b/c/h	1	1	

図 13　/a/b/c/h による更新後の状態
Fig. 13　CACHE and ACCESSLOG table state after /a/b/c/h inserted.

UID	PID	CHILDCNT	SAC	…
1	0	2	0.34	
2	1	3	0.66	
3	2	3	1.99	
4	3	0	2	
5	3	0	1	
…				

QUERY	SAC	ACOUNT	…
/a/b/c/g	2	2	
/a/b/c/h	1	1	
/a/b/c	1.99	1	

図 14　/a/b/c による更新後の状態
Fig. 14　CACHE and ACCESSLOG table state after /a/b/c inserted.

が行われる．また，図 13 の状態でクエリ/a/b/c が来ると，キャッシュテーブルに/a/b/c が挿入される．ただし，その SAC 値は 1 ではなく ACCESSLOG テーブルのノード c（UID = 3）から 1.99 になる．そしてノード b とノード a が式によってそれぞれ 0.66 と 0.34 に更新される（**図 14**）．

実際，クエリ/a/b/c/g と/a/b/c/h がノード g と h にアクセスすると，ACCESSLOG テーブルではそれらと親，先祖という関連を持つすべてのノードに対しても更新が行われるのが分かる．たとえば，ノード c の子ノードである g，h，i に均等にアクセスするとその分ノード c にも SAC が 1/3 ずつ上がっていくため，ノード c とそれぞれの子ノードである g，h，i の SAC との比較によりノード c をルートノードにする XML 部分木を結果として含む方が有効性があると思われる．つまり，/a/b/c[g]/h や /a/b/c[h]/g など潜在的なクエリにも応答可能だということである．一方，CACHE テーブルに/a/b/c のクエリの結果が格納されていると仮定する．それに対して，子ノードである g のみにアクセスされるクエリ（/a/b/c/g）が頻繁に来ると ACESSLOG テーブルにはノード g の SAC が 1 ずつ増加されるのに対してノード c は 1/3 ずつ増加される．すると，その差が徐々に増していくため，CACHE テーブルにあるレコード/a/b/c の代わりに g をルートにする部分木に入れ替えることが望ましい．したがって，CACHE テーブル/a/b/c のレコードに応答可能である/a/b/c/g によるアクセスが来た場合，従来のヒットカウントのように/a/b/c には直接 SAC を 1 カウント増やさないようにする．

以上の例では，/a/b/c タイプのように Child 軸（/）だけ持つシンプルクエリに対して説明を述べたが，それ以外にも Descendant 軸（//）や条件（[]）を持つクエリにも対応できる．たとえば，/a//f の場合，ノード f から親・先祖であるすべてのノード（ノード d（9）と b（2））を ACCESSLOG から探して更新を行う．しかし，クエリに明示的に書かれたノード（a と f）のみ限定で更新を行うことも考えられる．本研究では前者の手法をとる．/a/b[d]/c の場合，分岐ノード（b（2））を基準として/a/b/d と/a/b/c の 2 つのパス表現に分けてそれぞれ SAC の更新を行う．

従来のアクセスカウントの代わりに λ による SAC を取り入れるのは，あるノードがアクセスされるときそのノードにだけではなく，それに至るまでの経路にあるノード（親，先祖ノード）にも式 (1) による重みを与えることである．SAC は CACHE テーブルのレコード交換における基準値の 1 つであり，ACCESSLOG テーブルを参照してレコード間の比較による再編成にも利用される．次節ではキャッシュ再編成のとき，その判断の基準値を求める手法について述べる．

3.4　SAC による局所性

一般的に局所性とは，プログラムの実行やデータアクセスなどがある一部分に集中することをいう．本節では，キャッシュ再編性時，2 つのレコードを比べて削除対象を定めるために 2 つのノード間の相対的なアクセス局所性について述べる．前提として，比較対象になる 2 つのノードはそれぞれのクエリ（XPath 表現）の端末ノードにする．つまり，CACHE テーブルの RESULT フィールドに格納されている XML 部分木のルートノードのことである．そして，そのクエリ間は応答可能関係にあることにする．キャッシュ再編成が応答可能関係にある 2 つのクエリの二者択一を目的にしているからである．図 7 を用いて例をあげながら説明を行う．たとえば，/a/b/c と/a/b/c/g がキャッシュに格納されているとする．/a/b/c/g が/a/b/c に対して応答可能であるのは自明である．ここで 2 つのノードのうち，ノード g のみにアクセスすると，ノード g の SAC はノード c の 3（1/λ）倍になることが分かる．ノード c に対するノード g のアクセス局所性の最大値はノード c の SAC∗1/λ である．一方，ノード c の子ノードに均等にアクセスすると，ノード g の SAC とノード c の SAC は同じ値であることも分かる．そして，ノード g 以外のノードや親ノードのみにアクセスすると，その値が 0 に徐々に近づいていくのが分かる．つまり，以上に述べた特徴から以下の式を導く．

キャッシュに格納されている任意の 2 つのクエリ（XPath 表現）Q1，Q2 に対して，Q1 が Q2 から応答可能であるとき，それぞれの端末ノードを n，m とする．n が m に対するアクセス局所性値を δ_{nm} とすると，δ_{nm} の範囲は以下のようになる．

$0 < \delta_{nm} \leq n$ の SAC $* 1/\lambda_{mn}$ (2)

これにより，キャッシュ再編成時，δnm 値を用いて 2 つのレコードの比較を行うことができる．次節では，SAC と δ に基づくキャッシュ管理について述べる．

3.5　キャッシュの管理

キャッシュの管理とは制限されたキャッシュサイズという制約のうえ，ヒットレートが落ちない範囲でキャッシュスペースの利用率を高めることである．本研究では，DB2 Express-C9.7[19] ハイブリッド型データベースのテーブルを用いて実装を行っているが，キャッシュテーブルに格納される XML データ型を DOM[20] Tree に入れ替えれば，同様のアルゴリズムでメモリ上において利用できる．本節ではキャッシュセグメントの挿入と削除による更新やキャッシュ全体の再編成について述べる．読みやすくするために本節中のアルゴリズム内容では一部属性を省略することにする．

3.5.1　挿入と削除によるキャッシュの更新

従来キャッシュ交換を行う際，LRU や MRU など様々なキャッシュ交換関数が使用されてきたが，本研究では SAC のみに基づく LFU を用いることにする．これに時間的局所性を表す LRU と MRU を加えることもできる．

本研究ではキャッシュテーブルレコードの間に重複許容の可否によって 2 つの管理手法を提案する．対象になるクエリパターンは，応答可能な関係を持つ 2 つのクエリが包含（/a/b/c），被包含（/a/b/c/h）順で来る場合に限定する．まず，重複を許容した場合，キャッシュテーブルのレコードから応答可能性判定によって結果が求められるにもかかわらず，キャッシュスペースに余裕があれば新しいクエリとその結果を格納する．たとえば，キャッシュテーブルに/a/b/c とその結果がすでに入っているとする．その後応答可能であるクエリ/a/b/c/h が来ても，クエリとその結果をキャッシュテーブルに入れてしまう．再び，/a/b/c/h クエリが来るとキャッシュルックアップアルゴリズム（図 5）から complete マッチングで済むため，その分検索速度に利点があるといえる．しかし，時間の経過によりレコード間に重複が多くなるためスペースの消費が起こる．この現象に対してキャッシュ再編成アルゴリズムを提案する．また，キャッシュスペースに余裕があるにもかかわらず重複が許容されない場合，キャッシュの利用率を高めることができる．しかし，上に述べたパターンが逆だと（/a/b/c/g, /a/b/c）重複が生じる場合があるのでキャッシュ利用率を高めるためにはキャッシュ再編成アルゴリズムを実行する．実行するタイミングについては，ヒットレートの低下を観察する方法が考えられる．

図 15 のキャッシュ更新アルゴリズムについて述べる．1 行目から 9 行目に至る cache-

Algorithm Cache Replacement Algorithm

```
▪cacheLookUp / searchDB
01: if completeMatch
02:     return(query, result)
03: else if answerability
04:    return(query, result)
05:     insertCache(query, result)
06: else if searchDB
07:    return(query, result)
08:    insertCache(query, result)
09: else return null
 ▪insertCache
// n is cache table count, C is record at cache, rSize is remain cache size
11: if result.size ≤ rSize
12:     input(query, result)
13: else for (i = 0; i < n; i ++)
14:   // order by sac asce
15:       if (result.sac > Ci.sac)
16:          if (result.size ≤ Ci.size)
17:            delete Ci
18:            input(query, resultXmlData)
19:            return
20:         else
21:            rSize = rSize + Ci.size
22:             if(result.size ≤ rSize )
23:                 for(j = i; j < 0; j --)
24:                    delete Ci
25:                end for
26:                input(query, result)
27:                return
28:       end if
29: end if
```

図 15　挿入と削除によるキャッシュ更新アルゴリズム
Fig. 15　Cache update algorithm by delete and insert.

LookUp/searchDB の部分には，実際に SAC に関する ACCESSLOG や CACHE に更新が行われているが省略されている．クエリ結果が answerability（3 行目）処理によって得られた場合，insertCache 処理は上述した重複許容の可否による．searchDB 処理から得られた結果に対しては insertCache 処理を行う．キャッシュに余裕スペースがあればそのまま挿入（11，12 行目）されるが，キャッシュの状態が不十分な場合，キャッシュに対する削除と挿入などの更新が行われる．

まず，SAC の昇順にソートされたキャッシュレコードの SAC と比較を行う（15 行目）．条件を満たすと，次に互いのサイズの比較を行う．その条件まで満たした場合，delete と input 処理（17，18 行目）を行う．しかし，サイズの条件を満たさなければ次のレコードと

Algorithm Cache Reorganization algorithm

```
cacheTable1 = {ar1, ar2, ar3, ... arn} orderby sac, query desc
cacheTable2 = {br1, br2, br3, ... brm} orderby sac, query desc

01: for(i = 1; i ≤ n; i ++)
02:     for(j = i+1 ; j ≤ m; j ++)
03:         if(ari is answerable from brj )
04:             if( ari.sac  - 0.75*brj.sac / λ  > 0)
05:                 pId  =  executeQuery( select D.prefixId
06:                                       from PREDICATE D
07:                                       where D.cId = brj.cId)
08:                 executeQuery(delete from PREDICATE D, CACHE C
09:                                 where D.cId = brj.cId)
10:                 executeQuery(delete from PREFIX  P where P.prefixId  = pId)
11:                 executeQuery(delete from CACHE C where C.cId = brj.cId)
12:             end if
13:         else if(brj is answerable from ari )
                ....
14:         end if
15:     end for
16: end for
```

図 16　キャッシュ再編成アルゴリズム
Fig. 16　Cache reorganization algorithm.

の SAC の比較を行い，満たすと，そのレコードとのサイズの比較を行う．そして，サイズの比較も満たすと，そのレコードと入れ替える．しかし，比較対象 2 番目レコードとのサイズ条件を満たさなかった場合，1 番目のレコードのサイズと合算（21 行目）してから再び比較（22 行目）を行う．その条件を満たした場合，（23〜26 行目）処理によって 1，2 番目のレコードを削除して新たなレコードを挿入する．また，満たさなかった場合は上述した過程を繰り返していく．次にキャッシュの重複問題に対するキャッシュの再編成について述べる．

3.5.2　キャッシュの再編成

時間の経過に従って，キャッシュレコードの間には包含関係による重複や断片化が生じる．本項ではその現象を防ぐためのキャッシュの再構成について述べる．キャッシュ再編成によるアルゴリズムは**図 16** のようになる．

キャッシュテーブルの再編成は SAC 値と QUERY 値の降順に CACHE テーブルをソートすることから始まる．まず，応答可能性判定（3，13 行目）を行う．CACHE テーブルの中に格納されている 2 つのレコードに対する比較を行うので，1 番目の判定で失敗した場合，順序を変えて再び行わなければならない．成功した場合，どちらのレコードを残すかという判定（4 行目）を行う．比較判定の基準値として 3.4 節から求めた δnm を用いる．ここで δnm の係数 p（$0 < p <= 1$）を取り入れると，p の値によって再編成の効率に変化が生じると考えられる．つまり，クエリの出現パターンに対して，適した p の値を定めることがキャッシュ再編成に重要である．条件（4 行目）を満たすと該当レコードを削除する．ここで，CACHE テーブルからレコードを削除する前にそれを参照している PREDICATE テーブルから PREFIXID を求めておく（5 行目）．それから，PREDICATE テーブルのレコードを削除する（8 行目）．その後，PREFIX と CACHE テーブルの該当レコードの削除を行う．再編成作業はキャッシュの中に格納されているすべてのレコードを比較しなければならないためその計算量は $O(n^2)$ である．

4. 実　　装

3 章で説明した手法をもとに，XML キャッシュ管理システムを実装した．本章ではその実装について述べる．

本キャッシュシステムは Java1.6 で実装を行い，XML データ，ACCESSLOG テーブルと CACHE テーブル群の格納のために DB2 Express-C9.7 ハイブリッドデータベースを用いた．実装したプロトタイプシステムを**図 17** に示す．プロトタイプシステムは図 17 のようにいくつかのモジュールで構成される．

まず，XML データをデータベースに格納することにともない，SAX パーサを通じて XML データが読み込まれる．そして，発生した各イベントがすべてのノード情報を ACCESSLOG テーブル（図 2）に格納する．そのノード情報とは，ユニークな番号（UID），ノードをルートとした XML 部分木のサイズ（SIZE），そのノードの親番号（PID），子ノードの数（CHILDCNT），セマンティックアクセスカウント（SAC）などノードに関する基本情報である．

ユーザからクエリ（XPath）が来ると最初に ‘XPath Complete Match Checker’（図 2）によってマッチングを行う．それは，ユーザのクエリと CACHE 群の CACHE テーブル（図 4）レコードに格納されているクエリとの complete マッチングである．complete マッチングに成功すると CACHE 群からクエリ結果を返す．CACHE 群は CACHE と PREFIX と PREDICATE の 3 つのテーブルで構成される．一方，complete マッチングに失敗すると次は ‘XPath Answerability Checker’（図 5）によって応答可能性判定を行う．応答可能性の判定は 2.2 節で述べた 2 つの条件に基づいて行う．その判定を満たさなかった場合，CACHE 群からクエリに対する結果を求めることができない．ゆえに，XML データが格納

されている XMLDATA テーブル（図 3）で検索を行う．

一方，応答可能性判定条件を満たした場合，CACHE 群の CACHE テーブルの RESULT から結果を求めるため，‘XPath Rewriter’ によって適したクエリを再作成する．クエリ実行が終わるとそれに対する更新が行われる．アクセスしたノードだけでなく，その先祖関係にあるすべてのノードに関する情報が ACCESSLOG 側（図 9）で更新される．それがユーザからのアクセス統計値であり，本研究では SAC 値のみ更新を行うことにする．

この累積値を用いて ‘Semantic Cache Manager’ がキャッシュ管理を行う．‘Semantic Cache Manager’ は 2 つのモジュールで構成される．キャッシュフルの状態時，キャッシュに対する削除と挿入による更新が要求されると，‘Replacement Manager’ が SAC 値を基準値にして更新アルゴリズム（図 15）に従い更新を行う．また，時間の経過によってキャッシュレコード間に重複問題や断片化問題が生じることがある．その問題に対しては ‘Reorganization Manager’ がキャッシュ再編成（図 16）アルゴリズムを用いてキャッシュ管理を行う．実験では，キャッシュの再編成を行うとき，3.4 節から求めた δ のパラメータ値として 0.75 を用いる．

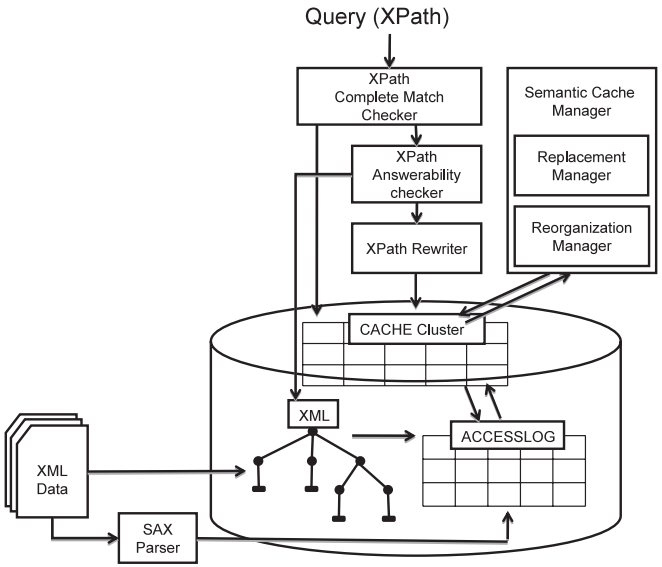

図 17　プロトタイプ XML キャッシュシステム
Fig. 17　Prototype XML cache system.

5. 実験と評価

本章では，提案する XML セマンティックキャッシュ管理システムの有用性を示すため，Non Cache と Naïve Cache システムとの比較実験を行う．比較システムの特徴は**表 1** に示す．Non Cache はキャッシュシステムを持たないデータベースそのもののみである．そして，Naïve Cache システムのうち，Complete Cache はクエリとキャッシュの間に応答可能性判定機能を持たない従来型のセマンティックキャッシュのことを表す．つまり，クエリとキャッシュレコードに格納されているクエリとの complete マッチング判定のみ行うことができるものである．もう一方の Containment Cache は応答可能性判定によってクエリ処理を行う．上述のシステムではキャッシュレコードの削除と挿入などの更新を行う際，通常のアクセスカウントを基準値とする LFU アルゴリズムを利用する．一方，本システムは SAC を更新の基準値として用い，キャッシュ再編成（図 16）機能の有無によって本システム SAC と本システム SAC+Reorg に分ける．実験によって新しく提案するキャッシュマネジメントがヒットレートにもたらす影響による有用性を示す．

実験にはデータベース上に実装したプロトタイプシステムを用いる．また，対象は様々なサイズを持つ Auction データセットである．実験に使われるクエリとして，XPath でシンプルクエリ（Child 軸のみ持つ），//クエリ（Descendant 軸包含），枝分かれクエリ（[] 条件付き）を用いる．ただし，実験で使われている DB2 Express-C 9.7 は XML データの更新を行う場合，1 つの XML データ全体を入れ替えることしかできない．ノードの部分更新，ノードの追加，ノードの部分削除を行うことはできないため，実験の前提として原本 XML データに関する更新は生じないと仮定する．

5.1 実験環境

実験は CPU：Pentium4 2.60 Ghz，メモリ：2 GB，OS：Window XP のマシンで行い，

表 1　比較対象システムの特徴
Table 1　Compared system for experiment.

比較システム	特徴
Non Cache	キャッシュ機能を持たないデータベース
Complete Cache	応答可能性を持たないキャッシュ
Containment Cache	応答可能性を持つキャッシュ
本システム SAC	SAC による Replacement Manager を持つキャッシュ
本システム SAC+Reorg	Replacement・Reorganization Manager を両方持つキャッシュ

データベースは IBM DB2 Express-C V9.7 を利用した．

5.1.1　データセット

実験用のデータセットとして Auction[15] データを用いた．このデータはオークションの情報が記述され，XMark[15] ベンチマークプロジェクトが提供するツールを用いて人工的に生成される．最大深さが 12 で比較的深さのあるデータセットとなる．本実験ではそのツールを用いてサイズ 100 M から 700 M まで 7 つのオークションデータを生成して実験の対象にする．

5.1.2　クエリセット

クエリの種類はシンプルクエリ，//クエリ，枝分かれクエリの 3 種類を用意する．シンプルパスは ‘/site/region/asia/item/name’ のように属性を持たない単純パスであり，//クエリは ‘/site/region/africa//name’ のようにクエリの中に//（Descendant 軸）を含む．そして，枝分かれクエリは/site/region/europe/item[location=‘Uzbekistan’]/name’ のように属性を持つクエリのことをいう．

上述した 3 種類で構成される実験用クエリセットを作る．まず，シンプルクエリを抽出する．SAX パーサを通じて原本 Auction データの読み込みを行う際，ユニークなパスを取り出す．その際各ノードに対する子ノードを格納しておく．ただし，子ノードがテキストコンテンツ（#PCDATA）を持つ場合（端末ノードの場合），‘ノード名 = “値”’ で格納する．そして属性ノードに関しては ‘@属性名 = “属性値”’ で格納する．これらはシンプルクエリ作成後に，枝分かれクエリを作るときに用いられる．抽出されたシンプルクエリは Auction データ 100 MB から 700 MB までそれぞれ等しく，総数が 514 個であり，パスの長さは 1 から 12 までの深さを持つ．深さによるパスの分布は**表 2** に示す．

次にシンプルクエリを用いて枝分かれクエリを生成する．この際，predicate として 2 つの種類があげられる．1 つは属性 (@)predicate であり，もう 1 つは子ノードで構成されるパス predicate である．シンプルパスを取り出す際に，パスのノードごとに格納しておいた子ノードと属性ノードを用いて predicate を作る．たとえば，シンプルパス/a/b/c から枝分かれクエリ/a/b[d][e]/c を求めるとき，先に求めておいたパス/a/b/c の b ノードの子ノードの集合を c，d，e と@f = ‘g’ とすると，シンプルパス/a/b/c の b ノードの子ノードである c ノード以外から選ぶ．ただしパス predicate の長さは 1 とする．

最後に//クエリの作り方について述べる．上述したようにできあがったシンプルクエリと枝分かれクエリからパスステップを 1 つずつ抜き取ることで作られる．たとえば，‘/a/b[d]/c[h=‘6’]/g’ というクエリに対して ‘/a//c[h=‘6’]/g’，‘/a/b[d]//g’，‘/a//g’ のように作り出せる．以上のような生成方法から作られたクエリのうち，データベースからクエリの結果が求められるクエリのみ用意する．

表 2　Auction データのユニークなパス情報
Table 2　Unique path about auction data.

深さ	数	比率
1	1	0.2
2	6	1.1
3	16	3.1
4	52	9.1
5	78	11.6
6	26	2.9
7	64	12.6
8	84	16.4
9	77	15
10	62	12
11	33	6.4
12	49	9.6
合計	514	100

5.2　パフォーマンスの比較

5.2.1　クエリの種類におけるパフォーマンスの比較

最初の実験として，本システムと Non Cache（DB）とのクエリの処理時間を測った．実験に利用されているクエリを**表 3** に示す．本実験の目的は 3 種（シンプル，//，枝分かれクエリ）のクエリがキャッシュでヒットしたのを前提にデータベースでのクエリ処理速度と比較することである．3 種のクエリの complete マッチングを測定するためキャッシュにはすでに同様のクエリとその結果が格納されている．そして，containment マッチングの処理速度を測るため，シンプルクエリの場合 ‘/site/people/person’，//クエリの場合 ‘/site/regions/asia’，枝分かれクエリの場合 ‘/site/regions/africa’ と ‘/site/regions/samerica/item/mailbox’ が格納されている．処理時間はクエリの実行回数を 10 回行った平均値で表す．そして，実験の CACHE テーブルサイズはそれぞれデータサイズの 10%に定める．Non Cache の場合，キャッシュシステム（CACHE 群）を持たないためすべてのクエリに対して XMLDATA テーブル（図 3）から結果を求める．

本システムでは 2 つに分けてクエリの処理を行う．与えられたクエリに対してキャッシュレコードのクエリとの complete マッチングを実行する方法と，応答可能性判定にかけ，レコードの中から検索（containment 検索と呼ぶ）する方法がそれである．それぞれ測定時間

表 3　実験のためのクエリタイプ
Table 3　Query type for experiment.

クエリタイプ	種類	結果サイズ（byte）						
		AUC100	AUC200	AUC300	AUC400	AUC500	AUC600	AUC700
シンプルタイプ	/site/people/person/address/street	511,790	1,006,796	1,509,441	2,015,710	2,569,246	3,081,246	3,591,246
	/site/people/person/address/country	524,882	1,033,249	1,549,631	2,068,948	2,634,281	3,123,961	3,654,288
	/site/people/person/address/zipcode	407,306	796,525	1,194,683	1,595,597	2,056,709	2,348,984	2,727,468
//タイプ	/site/regions/asia/item//from	112,971	223,272	339,912	448,686	548,147	668,425	778,693
	/site/regions/asia/item//to	105,919	209,138	318,551	421,565	514,812	627,685	731,477
	/site/regions/asia/item//mail	1,645,781	3,235,953	4,821,734	6,417,605	8,010,846	9,574,097	10,137,348
枝分かれタイプ	/site/regions/africa/item[name=“blind remainder brook caesar”]/location	48	48	48	48	48	48	48
	/site/regions/samerica/item/mailbox/mail [from=“Rachad Vrancken mailto:Vrancken@ask.com”]/to	67	67	67	67	67	67	67

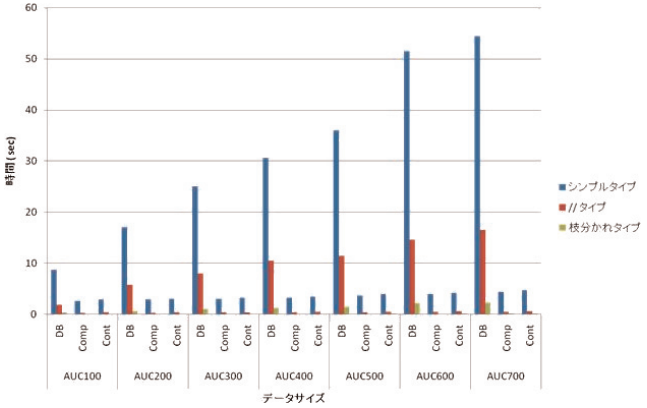

図 18　データサイズによるクエリ実行時間（Non Cache と本システム）
Fig. 18　Execution time about various data size (Non Cache vs. Prototype System).

を**図 18** の中で Comp と Cont で表す．すべてのクエリにおいて Comp，Cont，DB 順にクエリ実行時間が速いのが分かる．また，データサイズの増加にともないクエリ実行にかかる時間が線形的に増えていくのも分かる．実験結果により，サイズが CACHE テーブルより 10 倍大きい XMLDATA からのクエリ実行パフォーマンスが劣っているのが分かる．実験で使われているデータベースの場合，データベースからクエリの結果を求めるとき XMLDATA テーブル（図 3）の DOC フィールドから検索を行う．該当 DOC フィールドを取得するために，本体自体から提供される XML 領域索引（XML regions index）を用いる．ここで求めた DOC フィールドの最初のノードから始めすべてのノードを巡回（XSCAN）し，クエリを満たす結果を返す．一方，キャッシュでその結果を求めるときには，CacheLookUp アルゴリズム（図 5）の complete matching や answerability check の後，CACHE テーブルの RESULT フィールドから検索を行う．complete matching の場合，RESULT フィールドに格納されている XML データをそのまま返す．answerability check の場合は RESULT フィールドの最初のノードから始めすべてのノードを巡回（XSCAN）し，クエリを満たす結果を返す．つまり，クエリに対する XSCAN の処理コストは検索対象になる XML データサイズに比例するため，本システムのクエリ処理速度が Non Cache（DB）に比べ速い．

また，complete マッチングの場合は CACHE テーブルの QUERY 属性に付いているハッシュ演算によってマッチを行う．一方，containment 検索の場合，応答可能性判定のために CACHE 群の PREFIX テーブルと PREDICATE テーブルの間に結合演算を行った後 CACHE テーブルの RESULT 属性から結果を求める．つまり，Comp と Cont のクエリ実行時間の差はそれぞれが行う演算によるものと考えられる．

5.2.2　クエリパターンにおけるヒットレートの比較

本項ではクエリパターンによるヒットレートの比較を行う．対象データとして 5.2.1 項で使われたデータのうち，700 MB の AUCTION データ（AUC700）を使用した．AUC700 データに対して ACCESSLOG テーブルのサイズは 332 MB である．そして，キャッシュサイズを 10 MB から始め 5 MB ずつ足しながら，最大 70 MB に至るまでヒットレートとクエリ実行時間を測定する．5.1.2 項に用意されたクエリセットには，その結果が数百 MB（‘/site/people/person’ のような深さ 3 のシンプルクエリ）など最大 700 MB（‘/site’ のようなルートクエリ）にまで至るクエリがあるため，クエリ結果の最大サイズは最初のキャッシュサイズ（10 MB）を超えないようなクエリのみにする．すると，クエリの深さは 3（枝

表 4　タイプ別クエリパターン
Table 4　Query pattern.

タイプ	クエリパターン	総クエリ結果サイズ（MB）
タイプ 1	/a/b/c/d〜 /a/b/c/d[]/e/f[]/g/h[]/i/j[]/k/l	668
タイプ 2	/a/b/c/d[][][] 〜 /a/b/c/d[][][]/e[][][]/f[][][]/g[][][]/h/i/j/k/l	578
タイプ 3	/a/b/c/d〜 /a/b/c/d/e/f/g/h/i[][][]/j[][][]/k[][][]/l[][][]	482

分かれクエリ）から 12 の範囲内になる．クエリパターンによる比較を行うため 5.1.2 項のクエリセットから**表 4** にあるような 3 パターンのクエリを用意する．1 つのノードに与える属性 (@)predicate とパス predicate の数はそれぞれ 1 つと 2 つまでとする．タイプ 1 のパスの特徴は深さ 4 から 12 の範囲に predicate が散在している．タイプ 2 の場合は 4 から 7 までに predicate を与える．すると，それらのパスがアクセスされる XML データの部分がルートノードからの距離が 4 から 7 の範囲に集中する．タイプ 3 の場合，9 から 12 までに predicate を与える．それにより端末ノードと predicate を持つノードにアクセスが集中するといえる．

用意したクエリセットからそれぞれクエリパターンに合わせたクエリを 10,000 個ずつ抽出する．10,000 個のクエリを 2,000 個と 8,000 個に分け，それぞれ A セットと B セットにする．そして，A セットを 80 倍に増やした後，B セットを 20 倍に増やし，2 つのセットを混合させクエリセットを生成する．クエリセットを 10 回繰り返して実行する．ただし，毎回実行前にクエリの順序をランダムに定める．

すべてのクエリに対する検索結果のサイズは表 4 のようにそれぞれ 668 MB，578 MB，482 MB になる．キャッシュ再編成アルゴリズムの有用性を試験するために，本システム SAC+Reorg のみクエリセットを 5 回実行した後，キャッシュ再編成アルゴリズムによる再編成を行ってから実行を続けることにする．クエリパターンによる実験結果はそれぞれ**図 19** に示す．タイプ 1 から 3 までの実験結果を通してクエリパターンがそれぞれのシステムに与える影響が分かる．

まず，図 19 のタイプ 1 において，Complete Cache の場合，キャッシュサイズが 20 MB から 30 MB の区間と 40 MB から 55 MB の区間などクエリに対するヒットレートの変化がグラフ全体において少ないのが分かる．応答可能性判定ができないので，キャッシュテーブル側のクエリ実行が complete マッチングのみに依存するしかない．したがって，アクセスカウントが高くサイズが大きいクエリの結果がキャッシュレコードに格納されてしまうと，そこから利用できるキャッシュスペースが少なくなる．そのため図 19 のような傾向が現れ

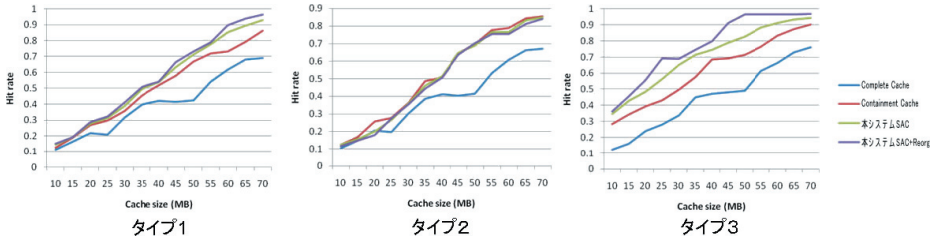

図 19　キャッシュサイズによるヒットレート
Fig. 19　Hit rate about various cache size.

る．つまり，Containment Cache と本システム SAC，本システム SAC+Reorg の場合，全般的に Complete Cache よりヒットレートが優れているのが分かる．これは，ユーザからのクエリとキャッシュに格納されているクエリとの間に応答可能性判定が可能だからである．そして，この 3 つの中から本システム SAC+Reorg，本システム SAC，Containment Cache の順にヒットレートが高いのが分かる．キャッシュサイズ 20 MB まではほぼ同じだが，その起点からキャッシュサイズ 60 MB までは本システムと Containment Cache との差が徐々に広がっていく．キャッシュサイズ 60 MB の箇所で本システム SAC との差は 11%，本システム SAC+Reorg との差は 17%になる．これは提案した SAC の特徴による結果である．

図 19 のタイプ 2 の場合はキャッシュサイズの全区間で Containment Cache の方が優れているのが分かる．つまり，本システム SAC と本システム SAC+Reorg で用いられるキャッシュ交換と再編成が効率的に機能していないといえる．これはクエリパターンによるものである．タイプ 2 では predicate が 4 から 7 の範囲に集中するためパスの端末ノードだけでなく，predicate を持つノードやその親ノードにも SAC が蓄積されていく傾向がある．つまり，λ の計算方法に基づいて深さが 3 や 4 のノードに SAC が累積してしまう．その原因は以下のように考えられる．XML データにおいてルートノードとの距離が近いほど，そのノードをルートノードとする XML 部分木のサイズが大きいため，クエリ結果の最大サイズ（10 MB）を超えてしまい，SAC が高いノードをルートノードに持つ XML 部分木がキャッシュに反映されないからである．本システム SAC と本システム SAC+Reorg に対するグラフの傾向が類似するのは本システム SAC+Reorg による再編成が機能していないせいだともいえる．

図 19 のタイプ 3 は 3 つのクエリパターンのうち，本研究で提案するキャッシュ管理が最

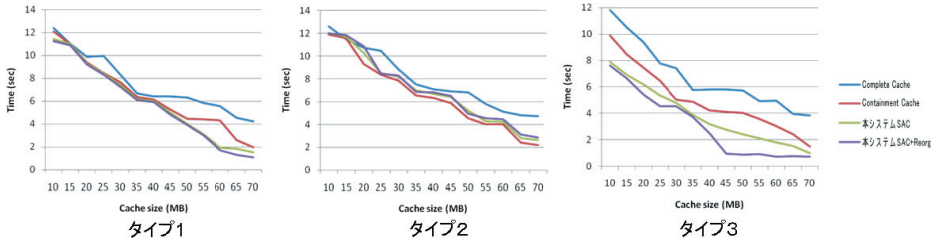

図 20　キャッシュサイズによるクエリ実行時間
Fig. 20　Execution time about various cache size.

も効率的に機能するケースであるといえる．クエリパターンの特徴上，クエリの predicate が深さ 9 から 12 の範囲に集中している．するとクエリ結果ごとに重複や包含関係が起こりやすいため再編性の方が効率良く機能する．最初は 10 MB の箇所から本システムのヒットレートが高いのが分かる．キャッシュサイズ 25 MB で本システム SAC+Reorg のグラフの数値が急増するのが分かる．そのとき，本システム SAC+Reorg と Complete Cache の差が 41%，本システム SAC+Reorg と Containment Cache との差が 26%になる．これはキャッシュ再編成によるヒットレートが上がるからである．キャッシュサイズ 50 MB の箇所でヒットレートが最大値に近接するのが分かる．そのとき，本システム SAC+Reorg と Complete Cache の差が 47%，本システム SAC+Reorg と Containment Cache との差が 25%になる．本システム SAC+Reorg を用いるとキャッシュを 50 M だけ用いることで，70 MB のキャッシュを持つ Containment Cache のヒットレートを上回るといえる．

5.2.3　ヒットレートにともなうクエリ速度の比較

次にヒットレートの影響によるクエリ実行時間について述べる．**図 20** はタイプ別に比較対象ごとに平均クエリ実行時間を表す．比較対象やクエリセットなど実験環境は 5.2.2 項のヒットレート測定の場合と同様である．実験結果によると，タイプと関係なくキャッシュサイズの増大にともなってクエリ実行時間が徐々に減少していくのが分かる．これは 5.2.1 項の実験結果と同様に，サイズが膨大な XMLDATA テーブルよりクエリとその結果がすでに格納されている CACHE テーブルから結果を求める方が検索速度が速いためと考えられる．

まず，図 19 のタイプ 1 でヒットレートの高い順と同様に図 20 の本システム SAC+Reorg，本システム SAC と Containment Cache，Complete Cache の順にパフォーマンスも良いのが分かる．本システム SAC，本システム SAC+Reorg と Containment Cache の場合，キャッシュサイズが 40 MB の箇所からクエリ実行時間の差が出始め徐々に広がっていくのが分かる．そして，キャッシュサイズ 60 MB で本システムと Containment Cache の差が最大になるのが分かる．

また図 20 のタイプ 2 の場合，図 19 のタイプ 2 のグラフに見られるように，Complete Cache 以外の 3 つのシステムはヒットレートの上昇にともなった速度の変化パターンが類似している一方で，Containment Cache の方が全般的に良いのが分かる．それは 5.2.2 項に述べたように，タイプ 2 のクエリパターンに対して本システム SAC，本システム SAC+Reorg のキャッシュ交換と再編成がうまく機能しなかったためである．特に，本システム SAC と本システム SAC+Reorg のグラフからキャッシュサイズ 15 から 20 MB と 55 から 70 MB の 2 つの区間では再編成が実行されるにもかかわらず，本システム SAC の方が本システム SAC+Reorg よりも若干速度が速いのが分かる．それは再編成の実行時，応答可能である 2 つのレコードのうち，生き残った方に対する選択の誤りで起きる現状である．

図 20 のタイプ 3 の場合，グラフに見られるようにヒットレートの上昇にともない，速度においても本システム SAC と本システム SAC+Reorg の方が優れているのが分かる．特にヒットレートのグラフにおいてヒットレートが急激に上昇したところ（キャッシュサイズ 25 MB と 45 MB）では本システムが Complete Cache，Containment Cache に比べて速度に格段と差がつくのが分かる．ここで特筆すべきは，タイプ 3 のようなクエリパターンに対してはキャッシュサイズ 50 MB のみで他のシステムの性能を上回ることである．図 19 と図 20 の実験結果によって，ヒットレートとクエリ実行時間はお互いに関係性を持っていることが分かる．つまり，クエリパターンによるキャッシュ管理がクエリ実行時間に影響を与えるといえる．

5.2.4　クエリパターンにおけるキャッシュ再編成

次に，キャッシュ再編成アルゴリズムの有用性を調べるため，再編成アルゴリズム実行後の CACHE テーブルのサイズを測る．ただし，Complete Cache と Containment Cache はキャッシュ更新の基準値として，従来のアクセスカウントを用いるので比較対象外にする．つまり，本システム SAC と本システム SAC+Reorg のみの比較を行うことで再編成アルゴリズムの有用性を調べる．本システム SAC の場合はクエリパターンによるタイプ別クエリセットをそれぞれ 10 回ずつ実行した後の残りサイズを測る．そして，本システム SAC+Reorg の場合，同様のクエリセットを 5 回実行し再編成アルゴリズムを行った後に残ったサイズを測る．**図 21** はタイプ別にそれぞれの CACHE テーブルの残りサイズを表す．

XML データの構造的特徴上，キャッシュには時間の経過に従い，キャッシュレコード間に包含関係による重複や断片化が生じる傾向がある．キャッシュ再編成アルゴリズムはこの

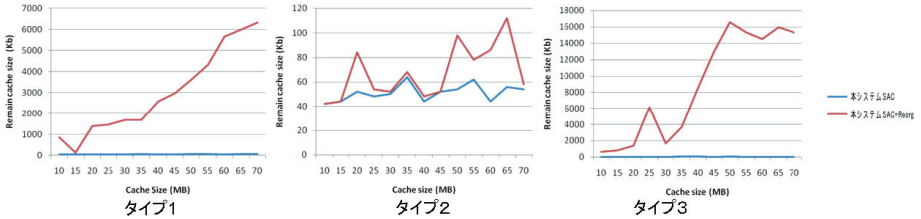

図 21　キャッシュ再編成による余裕キャッシュサイズ
Fig. 21　Remain cache size by cache reorganization management.

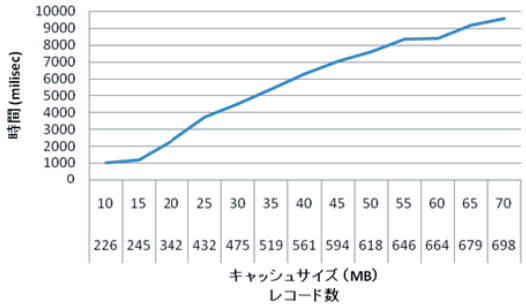

図 22　キャッシュ再編成にかかる時間
Fig. 22　Cache reorganization time.

問題を防ぐため，キャッシュに重複が多いと再編性後の余剰スペースを多くすると考えられる．図 21 の実験結果によると再編成アルゴリズムの有用性がクエリパターンの影響を受けているのが分かる．再編成アルゴリズムはタイプ 3 のようにクエリ結果ごとに重複や包含関係にある頻度が高いクエリパターンに対して有用であると考えられる．図 21 タイプ 3 の場合，キャッシュサイズ 45 MB から 70 MB までの区間でキャッシュ再編成後のサイズが他に比べて大きいのが分かる．つまり，キャッシュ再編成前の状態が他に比べて重複が多く利用率が低かったことが考えられる．また，再編成後のサイズが急増した 25 MB と 50 MB 部分と図 19 で同サイズの再編成後のヒットレートを調べると，同じ部分でヒットレートが向上したのが分かる．つまり，キャッシュ再編成アルゴリズムがヒットレートの向上につながるのである．

次に，キャッシュ再編成にかかる時間を測定した．**図 22** はクエリパターンに対しキャッシュサイズごとに行った平均再編時間の結果を表す．図 16 のキャッシュ再編成のアルゴリズムから再編成実行にかかる計算量はキャッシュのレコード数の増加によって指数的に増えている．ところが，キャッシュに格納されるクエリ間が包含関係にある頻度が高いため，実際に格納されるレコード数は log 関数に従ってキャッシュサイズが大きくなる．すると，再編成にかかる計算量は図 22 のグラフのように線形的に増加する．

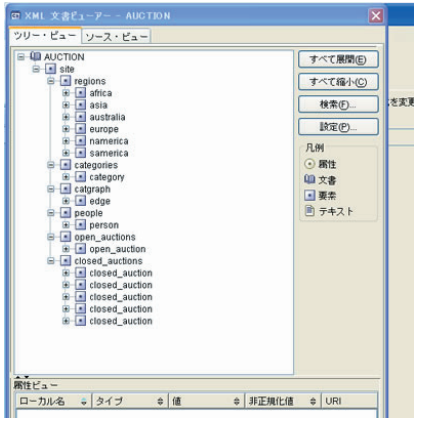

図 23　Auction データのディレクトリ構造
Fig. 23　Structure of Auction data.

5.2.5　クエリの偏りによるパフォーマンスの比較

本項ではクエリの偏りによるヒットレートの比較を行う．比較対象は 5.2.2 項のヒットレート測定の場合と同様である．ただし，本項の実験ではタイプ 3 のクエリパターンから偏りがあるクエリを用意する．**図 23** は AUC700 データのディレクトリ構造を表す．

図 23 から分かるように Auction データは site をルートノードとしてその下に regions, categories や catgraph などの子ノードを持っている．そのノード regions の子ノードであるノード asia に含まれる部分にアクセスされるクエリを 800 個用意する．それと同時に，ノード asia 以外にアクセスされるクエリ 200 個を用意する．総 1,000 個のクエリに対する結果のサイズは 103 MB である．このクエリを混合させ 100 回繰り返して実行する．

図 24 のグラフ全般にかけて，本システム SAC と Containment Cache のグラフはその数値が類似しているのが分かる．それらと比較すると，本システム SAC+Reorg のグラフからは再編成アルゴリズムの効果があることが読み取れる．これは相対的に狭い範囲にクエ

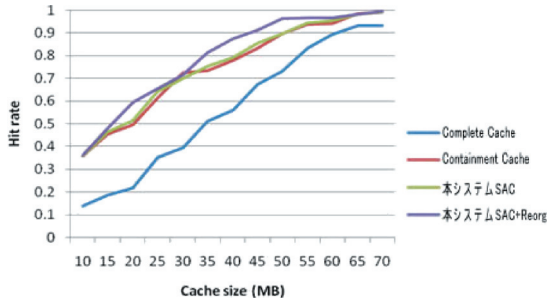

図 24　クエリの偏りによるヒットレート
Fig. 24　Hit rate about locality.

リのアクセスが集中すると XML データの構造的特徴上，重複や包含関係が生じやすくなるためである．

6. ま と め

本論文では XML データにおける新しい XML セマンティックキャッシュ管理手法を提案した．提案した管理手法ではノードに関する基本情報を総合的に管理するため，ACCESSLOG テーブルを用意した．ACCESSLOG テーブルはノード情報（構造情報）とそのノードのアクセス情報などを格納しキャッシュを管理するとき，有用な情報として用いることを可能とした．またノードの構造情報とアクセス情報からセマンティックアクセスカウント - SAC を考案した．SAC とは従来のアクセスカウントと異なり，XML が木構造であることに着目し，ユーザクエリによって最終的にアクセスされるノードだけでなく，そのノードと構造的密接性があるノードにも λ 値を付加することである．SAC の導入により精密にアクセスをカウントすることを可能とした．そして，削除と挿入時に SAC 値を基準値としたキャッシュ更新アルゴリズムを提案した．その SAC をもとに，応答可能関係にある 2 つのノード間に，アクセスに対する局所性を表す δ を求め，その δ を用いたアルゴリズムを使ってキャッシュを再編成することも提案した．

これらをふまえ，提案手法に基づきキャッシュ管理システムを実装し，下記の 3 種類の実験を行った．1 つ目は，クエリの種類に対してデータサイズによるクエリ実行実験であり，次に多様なクエリパターンに対してキャッシュサイズによるヒットレートとクエリ実行実験，そして偏りのあるクエリに対してキャッシュサイズによるヒットレートの 3 つである．

実験の結果，ヒットレートとクエリ実行実験について提案したキャッシュ管理手法が従来の手法より全般的に優れていることが分かった．特に，あるクエリパターン（実験でタイプ 3）と偏りのあるクエリにおいては，ヒットレートが従来の手法より 25 から 47%まで向上したことを示した．これにより，クエリ実行時間においても優れていることが証明された．そして，クエリパターン 3 の場合，再編成作業の実行によって，キャッシュサイズが 50 MB の部分で最大 16.6 MB まで削減可能であることを示した．

以上を総括して，提案した SAC 値を用いるキャッシュ更新アルゴリズムと SAC 値から求めた δ 値を用いたキャッシュ再編成アルゴリズムが有効であることを示した．今後の課題としては，SAC 値を求める際の子ノード数および，そのノードサイズがもたらす影響について検討していきたい．

参 考 文 献

1) Balmin, A., Ozcan, F., Beyer, K., Cochrane, R. and Pirahesh, H.: A Framework for Using Materialized XPath Views in XML Query Processing, *Proc. VLDB*, pp.60–71 (2004).
2) Berglund, A., Boag, S., Chemberlin, D., Fernandez, M.F., Kay, M., Robie, J. and Simon, J.: XML path language (XPath) 2.0, Technical Report, World Wide Web Consortium (2005).
3) Mandhani, B. and Suciu, D.: Query Caching and View Selection for XML Databases, *Proc. VLDB*, pp.469–480 (2005).
4) Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E. and Yergeau, F.: Extensible Markup Language (XML) 1.0 (3rd Edition), Technical report, World Wide Web Consortium (2004).
5) DeWitt, D., Futtersack, P., Maier, D. and Velez, F.: A study of three alternative workstation-server architectures for object-oriented database systems, *Proc. VLDB*, pp.107–121 (1990).
6) Miklau, G. and Suciu, D.: Containment and Equivalence for an XPath Fragment, *Proc. PODS*, pp.65–76 (2002).
7) Hu, H., Xu, J., Wong, W.S., Zheng, B., Lee, D.L. and Lee, W.C.: Proactive Caching for Spatial Queries in Mobile Environments, *Proc. IEEE ICDE*, pp.403–414 (2005).
8) Wang, H., Park, S., Fan, W. and Yu, P.S.: ViST: A Dynamic Index Method for Querying XML Data by Tree Structures, *Proc. ACM SIGMOD*, pp.110–121 (2003).
9) Tatarinov, I., Viglas, S.D., Beyer, K., Shanmugasundaram, J., Shekita, E. and Zhang, C.: Storing and Querying Ordered XML Using a Relational Database System, *Proc. ACM SIGMOD*, pp.204–215 (2002).

10) Ken, C., Lee, K., Leong, H.V. and Si, A.: Semantic Query Caching in a Mobile Environment, *ACM SIGMOBILE Mobile Computing and Communications Review*, Vol.3, Issue 2, pp.28–36 (1999).
11) Chen, L. and Rundensteiner, E.A.: A CachE-aware XQuery Answering System, *Proc. WebDB*, pp.31–36 (2002).
12) Chen, L., Rundensteiner, E.A. and Wang, S.: XCache-A Semantic Caching System for XML Queries, *Proc. ACM SIGMOD*, pp.11–12 (2002).
13) Carey, M., Franklin, M. and Zaharioudakis, M.: Fine-grained sharing in page server database systems, *Proc. ACM SIGMOD*, pp.359–370 (1994).
14) Ren, Q., Dunham, M.H. and Kumar, V.: Semantic Caching and Query, *Proc. IEEE Transactions on Knowledge and Data Engineering*, pp.192–210 (2003).
15) Schmidt, A., Waas, F., Kersten, M., Florescu, D., Manolescu, I., Carey, M.J. and Busse, R.: The XML Benchmark Project, Technical report, Centrum voor Wiskunde en Informatica (2001).
16) Dar, S., Franklin, M.J., Jonsson, B.T., Srivastava, D. and Tan, M.: Semantic Data Caching and Replacement, *Proc. VLDB*, pp.330–341 (1996).
17) Boag, S., Chemberlin, D., Fermandez, M.F., Florescu, D., Robie, J. and Simon, J.: XQuery 1.0: An XML Query Language, Technical Report, World Wide Web Consortium (2005).
18) Bottcher, S. and Steinmetz, R.: A DTD Graph Based XPath Query Subsumption Test, *Proc. 1st International XML Database Symposium*, pp.85–99 (2003).
19) DB2 Express-C 9.7. http://www.ibm.com/db2/express/
20) Document Object Model (DOM). http://www.w3.org/DOM/

(平成 21 年 9 月 20 日受付)
(平成 22 年 1 月 4 日採録)

(担当編集委員　山根 康男)

朴　　大一（学生会員）
平成 14 年慶應義塾大学大学院開放環境科学専攻修士課程修了．現在，同大学院開放環境科学専攻博士課程在学中．データベース，XML の研究に従事．

遠山　元道（正会員）
慶應義塾大学理工学部情報工学科准教授．昭和 54 年慶應義塾大学工学部管理工学科卒業．昭和 59 年同大学大学院博士課程修了後，管理工学科助手，専任講師を経て現職．博士（工学）．平成 8 年 Oregon Graduate Institute 客員研究員．平成 10〜13 年科学技術振興事業団さきがけ研究 21「情報と知」領域研究員．主にデータベースの研究に従事．電子情報通信学会，日本ソフトウェア科学会，IEEE，ACM 各会員．