관계형 데이터베이스를 이용한 XPath Accelerator: 구현과 튜닝

신 진 호[†] · 나 갑 주[††] · 이 상 원[†††]

요 약

XML은 데이터 표현과 교환을 위한 표준으로 급격히 자리잡아가고 있으며, XML문서는 다양한 응용분야에 도입되고 있다. 1990년대 후반부터, XML 전용 DBMS(Database Management Systems)들이 개발되어왔고, 최근 들어서는 상용 관계형 DBMS 벤더들도 XML 기능을 자사 제품들(예를 들어 오라클, IBM DB2, 그리고 MS SQL Server)에서 지원하기 시작했다. 본 논문에서는 XML 저장과 인덱싱 기법의 하나인 XPath Accelerator을 특정 관계형 DBMS상에 구현하고 이를 최적으로 튜닝하는 방안을 설명한다. 본 논문의 기여사항은 1) XPath Accelerator의 자세한 구현 방안과 2) 상용 관계형 DBMS의 최신 질의 처리 기법들을 활용한 튜닝 방법이다.

An XPath Accelerator on Relational Databases: An Implementation and Its Tuning

Jin-Ho Shin[†] · Gap-Joo Na[††] · Sang-Won Lee[†††]

ABSTRACT

XML is rapidly becoming the standard for data representation and exchange, and XML documents are being adopted in various applications. Since the late 1990s, some native XML database management systems(DBMSs) have been developed. More recently, commercial relational DBMS vendors try to incorporate full functionalities of XML into their products, such as Oracle, MS SQL and IBM DB2. In this paper, we implement a well-known RDBMS-based XML data storage and indexing technique, called XPath Accelerator, and tune it in an industry-leading RDBMS. Our contributions are two-folds: 1) an in-depth implementation of the XPath Accelerator technique and 2) its tuning to exploit the advanced query processing techniques of an RDBMS.

키워드 : XML, XPath Accelerator, RDBMS, 튜닝(Tuning)

1. 서 론

XML(EXtensible Markup Language)은 데이터 표현(data representation)과 문서 교환(data exchange)의 표준으로 빠르게 지정되고 있으며 최근에는 XML 문서 정의를 위한 XSchema와 XML 질의를 위한 XPath, XQuery, XQL 등의 XML 관련 표준들도 제안되었다.

현재 관계형 데이터베이스의 데이터 형식과 XML 형식의 데이터를 다루는 상용 데이터베이스로는 ORACLE, MS SQL, IBM DB2등이 대표적으로 사용되고 있으며 그 밖의 관계형 혹은 객체 관계형 데이터베이스에서도 XML의 지원이 이루어 지고 있다. 또한, XML 데이터만을 다루는 XML 전용 데이터베이스의 개발도 꾸준히 이루어지고 있다.

관계형 데이터베이스 내에서 XML 데이터의 저장과 질의 방안들에 대해 많은 연구와 논의가 있지만 본 논문에서는 그 중 기존에 제안된 XPath Accelerator[1]라는 XML 인덱스 기법을 실제 관계형 데이터베이스 시스템에 구현하고, 해당 데이터베이스 시스템에서 XPath Accelerator 적용 시 발생하는 문제점과 튜닝방안에 대해서도 실험을 통해 제시하고자 한다.

본 논문에서는 XML 저장과 인덱싱 기법의 하나인 XPath Accelerator를 대표적인 관계형 데이터베이스인 Oracle 상에 구현하였다. 참고로, Oracle에서는 자사의 객체관계형 기능을 활용해서 XML 기능(이를 Oracle XML DB라 부른다)을 지원하고 있는데[12], 본 논문의 내용은 이 기능과는 무관하다. 본 논문의 기여사항은 1) XPath Accelerator의 자세한 구현 방안과 2) 상용 관계형 데이터베이스의 질의 처리 기법들을 활용한 튜닝 방법이다.

본 논문의 구성은 다음과 같다. 먼저, 2장에서 관계형 데

※ 본 연구는 한국과학재단 목적기초연구(과제번호 : R05-2003-000-11943-0) 지원으로 수행되었음.
† 준 회 원 : ANYDATA 근무
†† 준 회 원 : 성균관대학교 대학원 컴퓨터공학과 석사과정
††† 정 회 원 : 성균관대학교 조교수
논문접수 : 2005년 1월 24일, 심사완료 : 2005년 3월 11일

이터베이스에서 구현 가능한 XML 인덱스 기법의 하나인 XPath Accelerator에 대해 알아보고, 3장에서는 이를 대표적인 데이터베이스인 Oracle에 직접 구현하는 과정을 보이겠다. 또한, 4장에서는 구현된 XPath Accelerator의 문제점을 찾아내고 그 튜닝방안에 대해 설명하겠다. 마지막으로 5장에서 결론을 내리고 향후 연구 과제를 기술하겠다.

2. XPath Accelerator

XPath Accelerator는 관계형 데이터베이스 상에서 XML 데이터를 효율적으로 저장하고 질의하기 위한 방법이다. 기본 개념은 XML 문서가 트리 형태의 데이터구조를 가지고 있다는 것을 이용하여 XML 문서의 각 element를 트리의 한 노드로 정의하고, XPath를 여러 개의 축으로 나누어 찾고자 하는 노드를 XPath 축 상에서 찾아내는 방식이다.

2.1 노드 기술자(Node Descriptor)

노드 기술자는 Tree 형태의 XML데이터를 노드단위로 분리 하였을 때 각 노드를 구별할 수 있는 노드 정보에 대한 데이터 집합이다. [1]에서 제시하고 있는 XPath Accelerator는 먼저 XML의 구조가 트리 형태로 되어 있기 때문에 각 엘리먼트를 트리의 한 노드로 생각하고 다른 노드와 구별할 수 있게 각 노드 별로 특정 값을 부여 한다. 각 노드에 특정 값은 일반적인 트리 검색 방식인 전위 탐색과 후위 탐색을 이용하여 탐색이 이루어지는 순서대로 노드에 값을 부여 한다.

(그림 1)에서와 같이 전위 탐색 시 노드를 방문하는 순서대로 노드에 pre값을 정의하고 후위 탐색의 경우 post에 값을 정의한다. 이 결과로 얻어진 (pre-rank, post-rank)의 쌍을 pre, post 축으로 이루어진 평면상에 표시할 경우 (그림 2)와 같이 각 노드는 좌표평면의 점으로 표현할 수 있다.

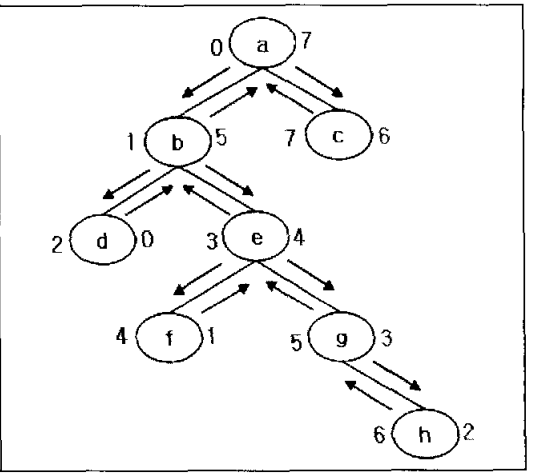
노드 기술자는 pre, post 값 이외에도 다음과 같이 5개의 값들의 집합으로 구성한다.

descriptor(v) = (pre(v), post(v), par(v), attr, tag)
pre(v) =노드 v의 전위 탐색 순위 값
post(v) =노드 v의 후위 탐색 순위 값
par(v) =노드 v의 부모 노드의 전위 탐색 순위 값
par(v)=pre(v')
attr =노드 v가 속성 값인지를 결정(true, false로 구성)
tag =노드 v의 실제 element 이름
예 : ND(e) = (3, 4, 1, false, e)

2.2 XPath의 축과 질의 윈도우(Query Window)

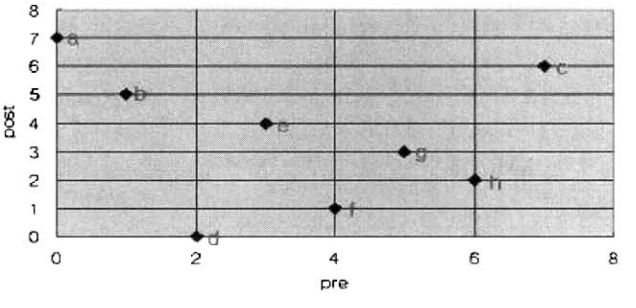
XPath는 XML 문서내에서 각 요소들을 노드(node) 개념으로 접근하여 소스트리의 정확한 위치를 찾는 경로 지정에

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlnx:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <xs:element name="a">
    <xs:element name="b">
      <xs:element name="d"/>
      <xs:element name="e">
        <xs:elemant name="f"></xs:element>
        <xs:element name="g">
          <xs:element name="h"/>
        </xs:element>
      </xs:element>
    <xs:element>
    <xs:element name="c"/>
  </xs:element>
</xs:schema>
```



(그림 1) 트리 탐색

tag	pre	post
a	0	7
b	1	5
c	7	6
d	2	0
e	3	4
f	4	1
g	5	3
h	6	2



(그림 2) 노드에 대한 (pre, post) 표현 및 평면 사상

관한 문법이다.

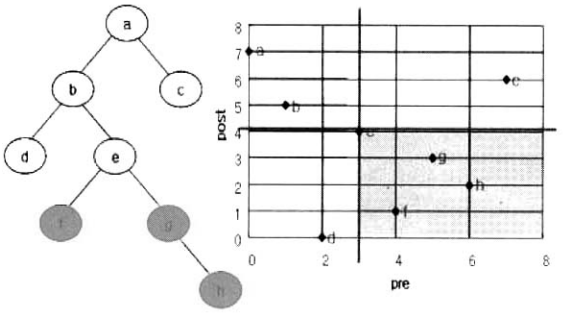
예 : /authors/author[first_name='furious decoys']

XPath Accelerator에서는 XPath를 분석하여 pre, post, par 값들의 관계로 표현하도록 함으로써 관계형 데이터베이스베이스 시스템에서 사용 가능한 SQL문으로 재구성 한다. XPath에서는 각 엘리먼트 사이의 관계를 '/'로 표현 하고 있으며 위의 예의 경우 엘리먼트 'author'는 하나의 엘리먼트이며'authors'의 child 엘리먼트이다. 또한 엘리먼트 'first_name'은 'authors'의 descendant 엘리먼트이며 동시에 엘리먼트'author'의 child 엘리먼트이다. 특정 노드에 대해 XPath 상에서의 서로 다른 엘리먼트 사이의 관계는 XPath의 축이라 정의 한다.

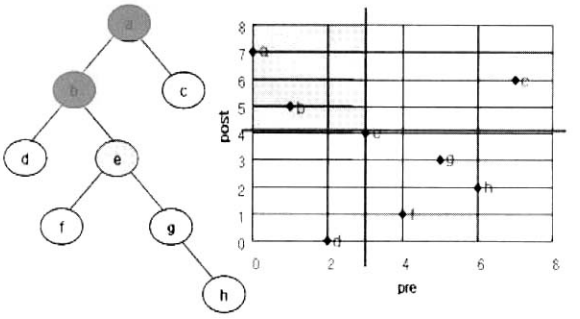
(그림 2)와 같이 각 노드를 평면상에 표시할 경우 해당 노드의 pre, post값을 기준으로 4 개의 평면으로 나누어지므로 해당 노드와 각 평면에 위치한 노드들간의 관계를 알 수 있다. XPath 축의 경우 (그림 3)의 (a), (b), (c), (d)에서와 같이 descendant, ancestor, following, preceding 4가지의 기본 축으로 구분되며, 특정 노드 v와 v'과의 pre, post관계를 는 <표 1>에 나타내었다.

〈표 1〉 축에 대한 pre, post 값과의 관계

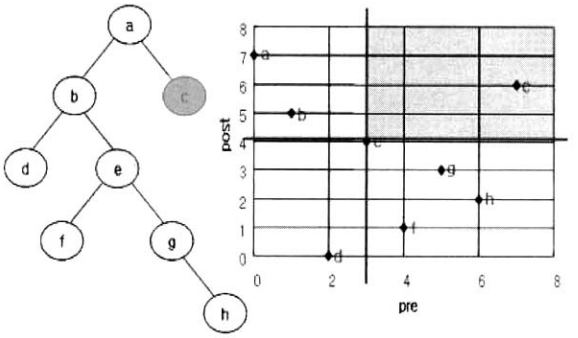
v/decsendant::v' Pre(v)<pre(v')∧post(v')<post(v)	v/ancestor::v' pre(v')<pre(v)∧post(v)<post(v')
v/preceding::v' Pre(v')<pre(v)∧post(v')<post(v)	v/following::v' pre(v)<pre(v')∧post(v)<post(v')



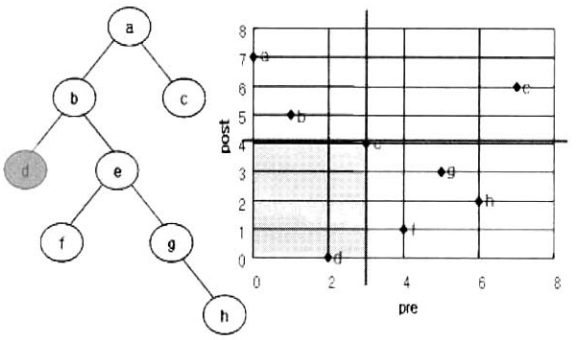
(a) e 노드의 descendant (e/descendant::*)



(b) e 노드의 ancestor (e/ancestor::*)



(c) e 노드의 following (e/following::*)



(d) e 노드의 preceding (e/preceding::*)

(그림 3) 노드 e에 대한 4개의 XPath 축

〈표 2〉 질의 윈도우

Axis a	Query window(a,v)				
	pre	post	par	att	tag
child	<(pre(v), ∞)	, [0,post(v))	, pre(v)	, false	,* >
descendant	<(pre(v), ∞)	, [0,post(v)),	, *	, false	,* >
descendant-or-self	<[pre(v), ∞)	, [0,post(v))	, *	, false	,* >
parent	<[par(v), par(v)]	, [0,post(v)]	, *	, false	,* >
ancestor	<[0,pre(v))	, (post(v), ∞)	, *	, false	,* >
Ancestor-or-self	<[0,pre(v)]	, [post(v), ∞)	, *	, false	,* >
following	<(pre(v), ∞)	, (post(v), ∞)	, *	, false	,* >
preceding	<(0,pre(v))	, (0,post(v))	, *	, false	,* >
Following-sibling	<(pre(v), ∞)	, (post(v), ∞)	par()	, false	,* >
Preceding-sibling	<(0,pre(v))	, (0,post(v))	, par()	, false	,* >
attribute	<(pre(v), ∞)	,[0,post(v))	, pre(v)	, true	,* >

특정 노드를 기준으로 노드와 축과의 관계를 정의한 것을 질의 윈도우라고 한다. 질의 윈도우에는 기본 4개축 이외에 11개의 축으로 pre, post값의 관계에 의해 나타내었고 이는 <표 2>에 나타내었다.

2.3 XPath에서 SQL로의 재구성

관계형 데이터베이스에서는 SQL을 통해 질의를 하므로 XPath를 SQL로 재구성하는 과정을 거쳐야 한다. 노드 기술자 정보를 담은 accel 테이블을 통해 노드 기술자와 질의 윈도우를 이용하여 아래와 같이 XPath를 SQL로 재구성할 수 있다.

XPath를 포함한 SQL :

```
SELECT EXTRACT (I.object_value, '/item')
FROM item I
WHERE EXISTSNODE (I.object_value, '/item[@id="I1"]') = 1

// 명령어 중심으로 기술
FIND node 'item'
FIND child of 'item' 'id'
      IF value of 'id'=="I1"
      FIND descendant of 'item'
return all descendants value

// 명령문 형식으로 표현 한 후 노드 기술자와 질의 윈도우를
 이용하여 다음과 같이 변환하다.
FIND IF(tag=='item')
FIND IF(pre of 'item'== par && tag='id')
      IF(value of 'id'=="I1")
      FIND(pre>pre of 'item' && post<post of 'item')
Return

// 관계형 데이터베이스에서 사용하는 SQL문으로 최종
변환한다.
SQL :
SELECT d2.pre, a3.tag, d2.val
FROM accel a1, accel a2, accel a3, xdata d1, xdata d2
WHERE a1.tag='item'
AND a1.pre=a2.par AND a2.tag='id'
AND a2.pre=d1.pre AND d1.val='I1'
AND a1.pre<a3.pre AND a1.post>a3.post
AND a3.pre=d2.pre
```

3. XPath Accelerator의 구현

XPath Accelerator의 구현을 위해 먼저, XML 문서를 노드 기술자와 각 엘리먼트들의 데이터값을 관계형 데이터베이스에 저장 하였다. 데이터의 저장 후 XPath를 SQL로 재구성하여 질의를 수행하는 실험을 하였다. 본 논문에서는 [1]에서 제시된 내용을 기초로 XML 문서를 JAVA 기반의 SAX Parser를 통해 파싱 하였고, 파싱된 데이터파일을 ORACLE 데이터베이스에 저장하여 XPath Accelerator를 구현하였다. ORACLE에 저장하는 방법은 대용량의 데이터에 대해 insert 속도의 향상을 위해 ORACLE에서 제공되는 SQL*LOADER를 이용하여 데이터를 로딩하였다. 또한, 데이터의 입력이 완료된 후에는 Analyze 명령을 수행하여 테이블과 인덱스에 대한 기초적인 통계정보를 수집하였다.

3.1 XML 문서 파싱(Document Parsing)

XML 문서 파싱 과정을 통해 노드 기술자를 구성하는데 필요한 정보와 각 노드의 실제 데이터 값을 얻을 수 있다. 파싱을 위해 본 논문에서는 JDK 1.4.2에 내장되어 있는 SAX Parser[2]를 사용하였다.

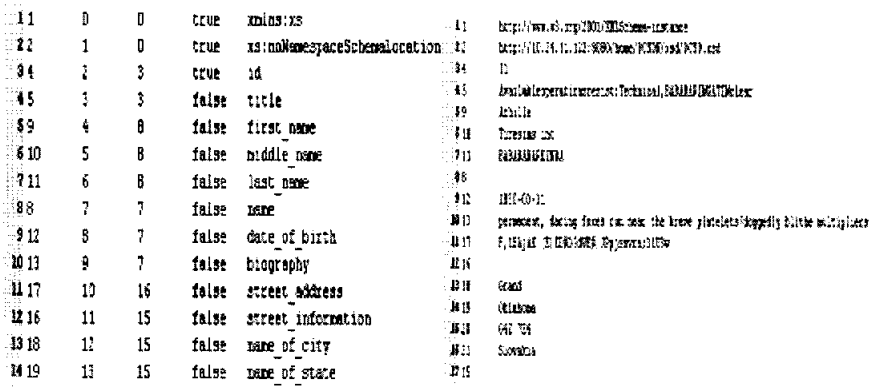
XML 문서가 최종적을 파싱 되어 생성되는 노드 기술자는 (그림 4)의 좌측에 나타나 있고, 각 노드가 지니는 데이터 값은 (그림 4)의 우측에 나타난 형태로 결과 파일이 생성하였다.

3.2 데이터베이스 구축

파싱을 통해 생성된 파일은 accel.dat, xdata.dat의 두 개의 파일을 ORACLE 데이터베이스에 다음 스키마를 이용하여 생성된 accel, xdata 테이블에 각각 저장된다.

```
CREATE TABLE accel(
      pre INTEGER,
      post INTEGER,
      par INTEGER,
      attr VARCHAR2(10),
      tag VARCHAR2(40)
)
```

```
CREATE TABLE xdata(
      pre INTEGER,
      val VARCHAR2(4000)
)
```



(그림 4) 파싱을 통한 (좌측)와 데이터(우측) 파일 내용

〈표 3〉 각 테이블에 대한 통계 정보

TABLE _NAME	BLOCKS	EMPTY _BLOCKS	AVG _SPACE	AVG_ROW _LEN	NDV (pre)	NDV (post)	NDV (par)	NDV (tag)	NDV (val)
XDATA	9974	1127	892	28	2390112				686935
ACCEL	12877	1207	845	38	2390112	2390112	846471	54	

〈표 4〉 인덱스에 대한 통계 정보

INDEX_NAME	LEAF_BLOCKS	DISTINCT_KEYS	CLUSTERING_FACTORY	NUM_ROWS
IDX_XDATA_VAL	8972	687364	1487277	1616183
IDX_XDATA_PRE	5520	2390112	59057	2390112
IDX_ACCEL_PAR	5510	846471	89249	2390112
IDX_ACCEL_POST	5520	2390112	12873	2390112
IDX_ACCEL_PRE	5520	2390112	82716	2390112

각 테이블에 데이터를 입력하는 방법은 ORACLE이 제공하는 SQL*LOADER를 이용하여 입력 시간을 최소화 하였다.

accel 테이블에 XML 문서의 정보를 입력하고 xdata 테이블에 각 노드의 데이터 저장이 완료된 후 accel 테이블의 *pre, post, par* 컬럼과 *xdata*의 모든 컬럼에 대하여 인덱스를 생성한다. 실험에 이용된 데이터에 대한 데이터 로딩을 마친 후의 각 테이블에 대한 통계정보는 <표 3>, <표 4>과 같다.

4. XPath Accelerator 튜닝

4.1 XPath Accelerator SQL의 문제점

*XPath Accelerator*의 구현 후 SQL문을 통해 XML 데이터의 질의 수행을 해본 결과 공통적인 문제점들이 나타났다. 관계형 데이터베이스 상에서 수행계획을 수립하는 과정에서 최적화기(Optimizer)가 비효율적인 수행계획을 선택함으로써 발생하는 문제이다. 이를 발생시키는 원인은 SQL과 노드 정보를 담는 accel 테이블과 노드별 데이터를 담는 xdata 테이블에 저장되어 있는 데이터들의 특성에 의해서 발생된다. 그 결과 잘못된 예측 카디널리티(cardinality)에 의해 최적화기는 테이블의 데이터 블록에 접근하기 위해 필요한 비용과 조인에 필요한 비용을 올바르게 산정하지 못하게 되어 다음과 같은 과정으로 비용 산정 오류를 일으킨다.

Descendant -→ 잘못된 cardinality 예측 -→ 비용 산정 오류
예 : a1.post > a2.post -→ Cardinality = 119k, Selectivity = 1/20 -→ 비용 산정 오류

위의 예는 한 노드의 descendant를 구하기 위한 첫 번째 조건식 a1.post>a2.post에서 이를 만족하는 데이터를 구하는 단계에서부터 선택도(selectivity)에 오류가 발생하고 이는 다시 카디널리티를 예측하는데 오류를 발생시켜 최종적으로 이후 수행해야 할 연산 단계에서의 올바른 비용 산정에 실패 하게 된다.

이러한 문제점들은 두 가지의 경우로 나누어 생각해 볼 수 있다. 첫 번째 경우는 질의 수행 과정 중 최종 단계의 조인에서의 해시조인(Hash Join)을 선택하는 경우이며, 이 경우는 최적화기가 예상한 카디널리티가 실제의 값보다 많다고 예측하는 과대 추정(over estimation)의 문제이다. 두 번째 경우는 과대 추정과 반대로 실제 카디널리티보다 더 적게 예상하는 과소 추정(under estimation)의 문제이다. 과소 추정 의 경우는 조건식에 알맞은 중간 노드를 찾고자 하는 경우에 대해서 주로 발생하는 문제이다.

본 논문에서는 이러한 과대 추정과 과소 추정에 대해 자세히 살펴보고 각 문제점을 개선할 수 있는 튜닝 방안을 모색하였다.

실험을 위해 XBench[3]라는 XML 데이터베이스를 위한 벤치마크를 이용하였으며, 데이터의 크기는 100M의 단일 문서의 데이터 파일(DCSD)를 사용하였다.

4.2 과대추정(Over Estimation) 문제

<표 5>에 제시된 Query1을 수행하면 (그림 5)와 같은 수행계획으로 질의가 수행된다. Query1의 경우 수행 과정 중 과대추정이 발생하였으며 그 결과 마지막 수행 단계에서 해시조인을 유발하게 된다.

Query 1을 수행 한 결과 100개의 결과 로우를 패치했으며 약 20초 정도의 수행 시간이 걸렸다. 이러한 결과는 관계형 데이터베이스에서 Query1이 효과적으로 수행 되지 못했음을 알 수 있다.

〈표 5〉 Query 1

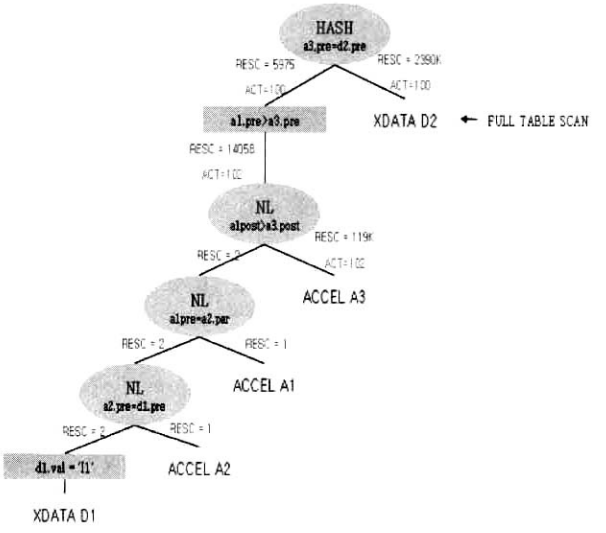
XPath를 포함한 SQL :

```
SELECT extract(Lobject_value, '/item')
FROM item I
WHERE existsNode(Lobject_value, '/item[@id="I1"]') = 1;
```

SQL :

```
SELECT d2.pre, a3.tag, d2.val
FROM accel a1, accel a2, accel a3, xdata d1, xdata d2
WHERE a1.tag='item'
AND a1.pre=a2.par AND a2.tag='id'
AND a2.pre=d1.pre AND d1.val='I1'
AND a1.pre<a3.pre AND a1.post>a3.post
AND a3.pre=d2.pre;
```

수행 완료 후 결과 로우수 : 100
질의 수행 시간 : 00:00:19.64



(그림 5) Query1 수행 계획

(그림 5)과 같이 a1.post>a3.post의 조건을 이용한 조인 과정은 예측 카디널리티와 실제 카디널리티가 많은 차이를 보였으며 이러한 차이의 원인은 a1.post>a3.post에 대하여 수행 계획 단계에서 카디널리티를 산정하는데 있어서 기준이 될 수 있는 a1.post값을 알 수 없기 때문이다. 최적화기는 a1.post의 값을 변수와 같은 형식으로 인식하기 때문에 올바른 선택도를 산정 할 수 없고 최적화기는 선택도를 1/20로 가정한다. 따라서 실제와 다른 선택도를 가정하기 때문에 올바른 카디널리티를 예측 할 없는 문제점을 보였다.

최적화기가 수행계획을 설정하는 단계에서 접근 방식에 따른 비용과 조인 방식에 따른 비용산정을 참고문헌 [4]에서 제시된 공식을 통해 계산하여 수행 계획과 비교해 보았다. [4]에 제시되어 있는 공식에 각각의 값을 대입하여 보면 마지막 해시 조인의 비용은 10374이며 만약 중첩루프조인(Nested Loop Join)을 선택할 경우의 비용은 44142가 되므로 최적화기는 더 적은 비용을 가지는 해시 조인을 선택한다. 따라서, 102개의 카디널리티를 최적화기의 과대 추정으로 인해 해시 조인을 발생 시키며 이로 인해 성능 저하를 가져 오게 됨을 알 수 있다.

이를 해결하기 위해서는 1/20로 가정된 선택도를 올바르게 수정 하면 쉽게 해결이 가능 하다. 그러나 이 경우 선택도에 대한 가정을 최적화기 내부에서 변수 값으로 처리하므로 선택도의 수정은 불가능하다.

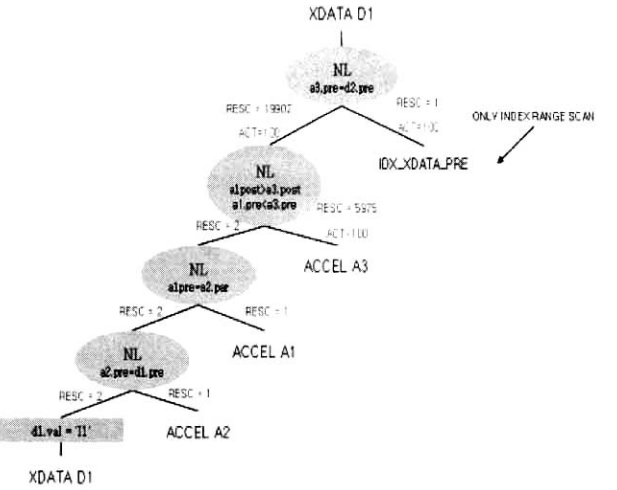
4.3 과대 추정의 튜닝 - B+ 트리 인덱스

앞에서 1/20에 대한 선택도 수정이 불가능 함을 알 수 있었다. 따라서 다른 방식을 이용한 튜닝이 필요하다. 중간 단계에 대한 카디널리티 가 102개 이므로 전체 테이블 스캔이 필요한 해시 조인을 이용하는 것 보다 인덱스를 이용한 중첩 루프 조인을 이용하는 것이 더 좋은 수행 능력을 보일 것을 알 수 있다. 따라서 인덱스를 이용한 접근 방식을 선택할 수 있게 튜닝을 시도 하였다.

인덱스를 이용하도록 하는 기법은 여러 가지가 존재 하지만 본 논문에서는 Oracle의 파라미터인 optimizer_index_caching[5], ptimizer_index_cost_adj[5]에 값을 조정하여 인덱스를 이용하도록 하였다. optimizer_index_caching=100의 값으로 세팅하여, 모든 인덱스블록이 버퍼에 존재하고 있는 것으로 가정하고 optimizer_index_cost_adj=1 의 값으로 세팅하여 인덱스를 접근 방식으로 선택하였을 경우의 비용이 최적화기 가정하고 있는 비용의 1%로 하였다. 두 개의 파라미터를 조정함으로써 인덱스를 이용할 경우의 비용이 실제보다 매우 작게 가정하여 최적화기가 해시 조인을 피하고 인덱스를 이용한 중첩 루프 조인을 택하도록 유도하였다.

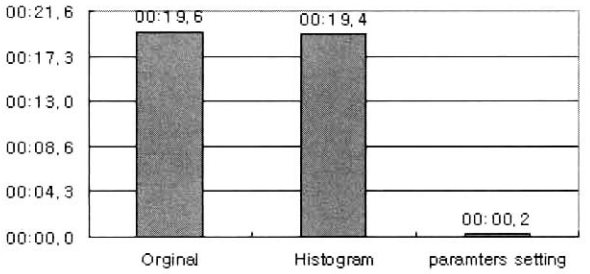
중첩 루프 조인을 선택하여 수행 할 경우의 실행 계획은 (그림 6)과 같고, 수행 시간은 다음과 같이 나타났다.

결과 로우수	수행 시간
100	00:00:00.02



(그림 6) 인덱스를 접근 방식으로 선택하였을 경우의 질의 수행계획

일반적으로 하나의 노드에 대한 하위 노드들을 모두 찾아야 하는 경우 전체 노드수의 비해서 매우 작은 양이다. Query1에서도 알 수 있듯이 2390112개의 전체 노드들 중에서 찾고자 하는 노드는 100개로 적은 개수이다. 따라서 마지막 최종 결과를 얻기 위해서는 인덱스를 이용하여 노드를 검색하는 것이 효과적이다. 즉, 하나의 노드에 대한 descendant 노드들을 찾는 검색 방법에는 인덱스를 이용하는 것이 효과적임을 알 수 있다.

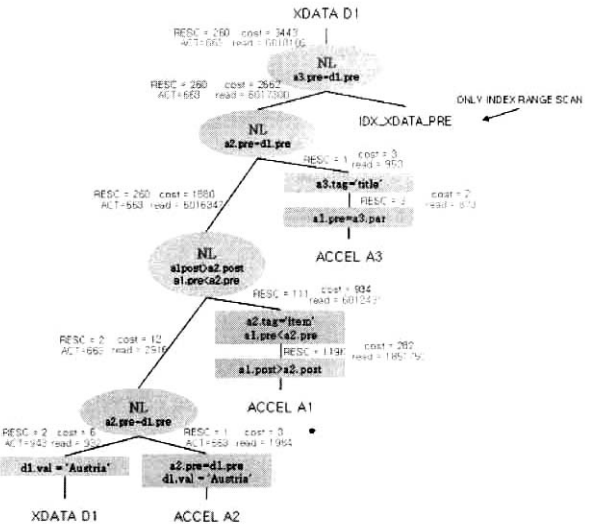


(그림 7) Query1의 각 경우에 대한 수행시간 비교

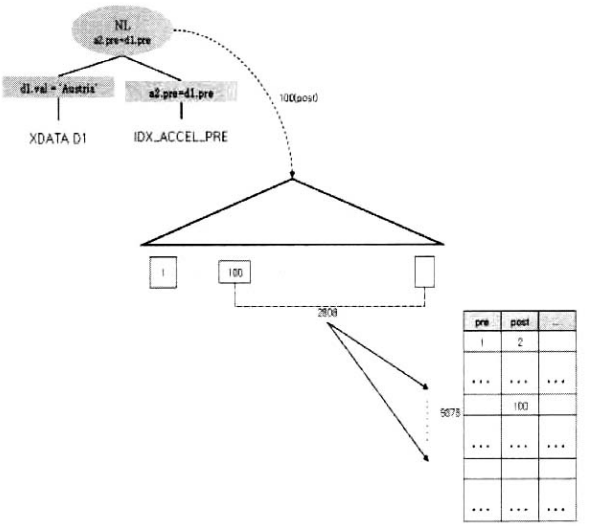
4.4 과소추정(Under Estimation) 문제

특정 노드의 descendant 노드들 중에서 값을 만족하는 노드를 찾는 경우가 있다. 이 경우 두 개의 accel 테이블이 서로 조건을 만족하는 노드를 찾기 위해 조인을 수행하는 단계에서 과대 추정과 반대로 과소 추정 의 문제가 발생할 수 있다. <표 6>의 Query2 질의문을 통해 과소 추정이 발생함을 살펴보고 이로 인한 성능 저하의 해결책을 모색하여 보았다.

수행계획은 (그림 8)과 같고, descendant를 구하는 과정에서 과소추정의 문제를 가지고 있었다. a1.post>a2.post를 구하는 과정에서 전 과정을 통해 구해진 하나의 a2.post값에 대해 a1.post>a2.post값을 만족하는 a1.post 값은 매우 많으며, 실제 디스크 블록에 접근 하는 횟수를 이용하여 계산을 하여 보면 (그림 9)와 같다.



(그림 8) Query2의 수행계획



(그림 9) post 값에 따른 인덱스와 데이터블록 접근

〈표 6〉 Query2

XPath를 포함한 SQL :

```
SELECT extrace(value(i), '/item/title')
FROM item i
WHERE existsNode(value(i), '/item/../[name_of_country="Austria")=1;
```

SQL :

```
SELECT d2.pre, a3.tag, d2.val
FROM accel a1, accel a2, accel a3, xdata d1, xdata d2
WHERE a1.pre>0 and a1.tag='item'
AND a1.pre<a2.pre and a1.post>a2.post and a2.tag='name_of_country'
AND a2.pre=d1.pre and d1.val='Austria'
AND a1.pre=a3.par and a3.tag='title'
AND a3.pre=d2.pre;

수행 완료 후 결과 로우수 : 663
질의 수행 시간 : 03:06:22.63
```

〈표 7〉 힌트를 추가한 Query2

```
SELECT /*+ FULL(a1) */d2.pre, a3.tag, d2.val
FROM accel a1, accel a2, accel a3, xdata d1, xdata d2
WHERE a1.pre>0 and a1.tag='item'
AND a1.pre<a2.pre and a1.post>a2.post and a2.tag='name_of_country'
AND a2.pre=d1.pre and d1.val='Austria'
AND a1.pre=a3.par and a3.tag='title'
AND a3.pre=d2.pre;
```

결국 하나의 a2.post 값을 대상으로 볼 때 선택도는 약 1/2정도가 되어야 하지만 최적화기는 1/20로 가정하여 카디널리티를 예측하게 되었으므로 약 10배정도의 카디널리티에 대한 과소 추정이 발생하게 되었다.

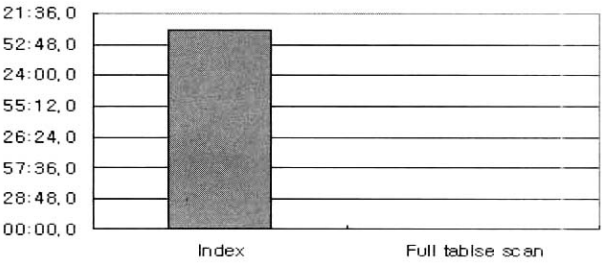
4.5 과소추정(Under Estimation)의 튜닝 - 전체 테이블 스캔 이용

과소 추정의 경우 실제 약 1/2의 선택도를 가진 조건식을 1/20로 추정함으로써 발생한다. 이와 같은 과소추정된 선택도 때문에 최적화기는 인덱스 스캔을 선택한다. 하지만, 실제 데이터 분포를 고려하면 전체 테이블 스캔(full table scan)을 이용하는 것이 보다 효과적이다. 전체 테이블 스캔을 수행하기 위한 방법으로 Query2에 힌트를 추가하여 Query를 재 수행하였다.

수행 계획은 (그림 10)과 같고 수행 시간은 다음과 같다.

결과 로우수	수행 시간
663	00:01:02.61

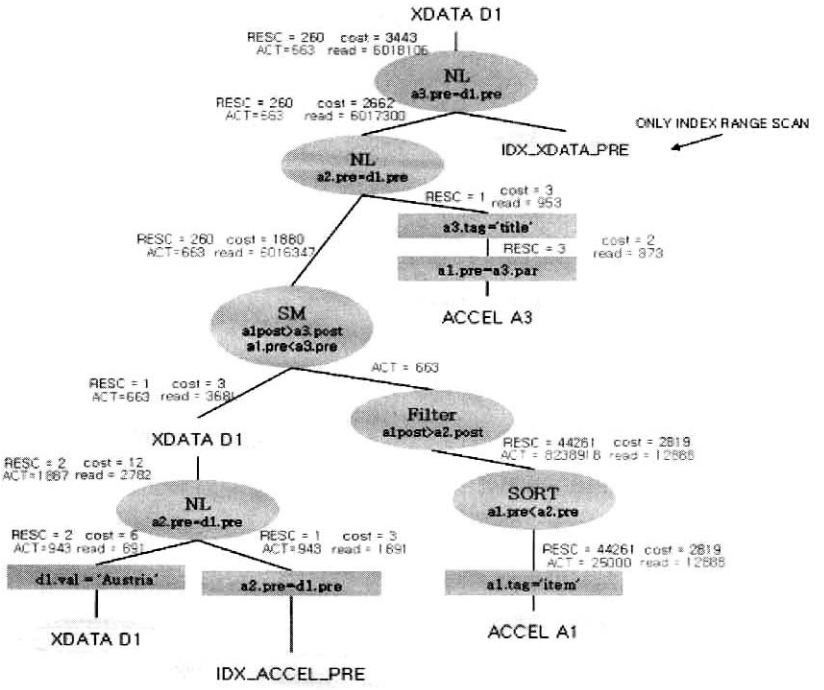
수행 시간의 경우 3시간이 넘던 Query가 1분 정도로 줄어 수행 시간이 줄었다. 또한 이 경우의 수행계획도 중첩 루프조인이 아닌 정렬합병(Sort Merge) 조인으로 바뀌었다.



(그림 11) Query2의 튜닝 결과 그래프

5. 결론 및 향후 연구방향

XPath Accelerator를 구현하여 XML 데이터를 관계형 데이터베이스 시스템에 저장하고 SQL을 통해 질의하는 과정



(그림 10) Query3의 수행계획

을 알아보았다. 또한, 이 경우 발생되는 SQL문의 수행상의 문제점을 분석하고 문제점 해결을 위한 튜닝 방안도 제시해 보았다. 결론적으로XPath Accelerator의 수행시 발행하는 두 가지 문제점은 과대 추정(over estimation)과 과소 추정(under estimation)으로 나눌 수 있고 각각의 해결책은 다음과 같다.

- 과대추정의 해결책: accel, xdata 테이블에 대하여 **인덱스 스캔** 선택
- 과소추정의 해결책: 대상 테이블에 대하여 **전체 테이블 스캔** 선택

그러나, 두 가지 경우 모두 수행 시간이 지연되는 원인은 최적화기의 내부적으로 볼 때 동일하다. 수행계획을 수립하는 단계에서 카디널리티와 비용의 예측이 실제 값과 차이가 나기 때문이다 특히 카디널리티 예측의 오류로인해 비용 예측의 오류가 유발된다.

이는 최적화기가 XML 데이터의 트리 구조와 대응되는 accel 테이블의 데이터 특성을 파악하지 못하고 단순한 관계형 테이블로 인식하기 때문이다. 이로 인해 잘못된 카디널리티를 예측하게 되고 결국 최적화된 수행계획을 생성하지 못하게 되는 것이다. 특히 descendant와 같은 Range 조건을 필요로 할 경우 정확한 선택도를 계산하기 위한 근거를 마련하지 못함으로 1/20로 선택도를 가정함을 알 수 있었다. 그러나 최적화기내부에서 발생되는 잘못된 예측은 현실적으로 수정이 어렵다. 따라서 본 논문에서는 과대 추정, 과소 추정과 같은 문제 발생시 해결하기 위한 튜닝 방안을 제시하였다.

본 논문의 의미는 앞으로 널리 사용될 XML데이터에 대한 접근 방식으로 제시된 많은 방법들 중 하나를 선택하여 구현하고 실험을 통해 문제점과 개선점을 제시함으로써 관계형 데이터베이스를 이용하여 XML 데이터를 저장/관리하기 위한 새로운 방식의 제안으로 볼 수 있다. 또한, 트리 형태의 데이터를 관계형 데이터베이스로 저장하고 질의 하는 과정에서 발생되는 질의 최적화기의 문제점을 살펴보고 이를 극복하는 방법을 제시하였다. 실질적인 활용을 위해서는 보다 많은 종류의 데이터와 다양한 크기의 데이터에 대한 실험을 통해 성능평가를 해보아야 할 것이다.

본 논문에서는 Join 발생은 그대로 두고 각 조인에 대한 문제점을 찾아 조인 방식을 바꾸는 방법을 제시하였지만 Query2에서도 볼 수 있듯이 663개의 데이터를 얻어 오기 위해 1분 정도의 시간을 결리는 점에서 성능 개선을 위한 튜닝의 여지가 많음을 알 수 있다. XPath Accelerator가 보다 좋은 성능을 유지하기 위해서는 본 논문에서 제시한 방법과 다른 관점의 접근인 Staircase Join[10]에서 제시된 방법처럼 Join 수행 자체를 줄이는 방법이 있을 수 있다. 그리고 또 다른 방법으로 고려해 볼 수 있는 사항은 논문에서 사용된 B+ Tree Index 이외에 XPath Accelerator에 보다 적합한 R-tree Index등의 다양한 Index 구조에 대한 연구도 필요로 할 것이다. 마지막으로, XML과 같이 전형적인 트리 구조와 문자열을 조건 식으로 많이 포함하는 데이터들에 대해, 관계형 최적화기에서 선택도 추정 등의 측면에서 추가적인 연구[13,14]가 필요하다.

참 고 문 헌

[1] Torsten Grust, "Accelerating XPath Location Steps", SIGMOD 2002

[2] http://www.saxproject.org/

[3] B. B. Yao, M. T. Özsu, and N. Khandelwal, "XBench Benchmark and Performance Testing of XML DBMSs", 20th ICDE 2004, pp.621-632, 2004

[4] Wolfgang Breitling, "A Look Under the Hood of CBO: The 10053 Event", Centrex Consulting Corporation, 2004

[5] Oracle Database Documentation Library, "Reference", Oracle Documentation, pp.1-99

[6] World Wide Web Consortium "http://www.w3c.org/XML/"

[7] 김찬웅, "(IT expert)자바 개발자를 위한 XML 프로그래밍", 한빛미디어, 2002

[8] Oracle Database Documentation Library, "Oracle XML DB Developer's Guide", Oracle Documentation, 2003

[9] Guy Harrison, "Orracle SQL High-Performance Tuning second edition", Prentice Hall PTR, 2001

[10] Torsten Grust, "Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps", 29th VLDB, 2003

[11] Shankar Pal, "Indexing XML Data stored in a Realtional Database", 30th VLDB, 2004

[12] M. Krishnaprasad et al., "Query Rewrite for XML in Oracle XML DB", 30th VLDB, 2004

[13] Ashraf Aboulnaga et al., "Estimating the Selectivity of XML Path Expressions for Internet Scale Applications", 27th VLDB, 2001

[14] Surajit Chaudhuri et al., "Selectivity Estimation for String Predicates: Overcoming the Underestimation Problem", 20th ICDE, 2004

신 진 호

e-mail : sjho14@nate.com

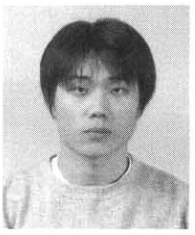
2003년 성균관대학교 정보통신공학부 (학사)

2005년 성균관대학교 대학원 컴퓨터공학과(공학석사)

2005년~현재 ANYDATA 근무

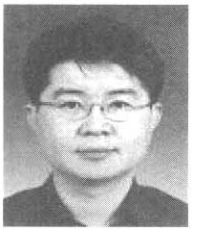
관심분야 : XML, DB 튜닝, 모바일 인터넷

나 갑 주



e-mail : factory@skku.edu
2003년 한국항공대학교 항공전자공학과 (학사)
2004년~현재 성균관대학교 대학원 컴퓨터공학과 석사과정
관심분야 : XML, DB 튜닝, 인터넷 프로그래밍

이 상 원



e-mail : wonlee@ece.skku.ac.kr
1991년 서울대학교(학사)
1994년 서울대학교(석사)
1999년 서울대학교(박사)
1999년~2001년 한국오라클
2001년~2002년 이화여대
2002년~현재 성균관대학교 조교수
관심분야 : XML, DB 튜닝, Data Warehouse/Data Mining