제22회 한국정보처리학회 추계학술발표대회 논문집 제11권 제2호 (2004. 11)

XPath Accelerator: 구현 및 튜닝

신진호, 이상원
성균관 대학교 컴퓨터 공학과
VLDB 연구실
e-mail : sjho14@skku.edu

XPath Accelerator: An Implementation and its Tuning

Jin-Ho Shin, Sang-Won Lee
VLDB Laboratory, Dept. of Computer Engineering, Sungkyunkwan University

요 약

XML 은 데이터 저장과 전송을 위한 수단으로 자리잡아 가고 있으며, 관계형 DBMS 를 이용해서 효과적으로 대용량의 XML 데이터의 저장과 검색에 관한 연구가 진행 되고 있다. 본 논문에서는 기 제안된 XPath Accelarator 라는 XML 데이터 인덱스 메커니즘을 상용 관계형 DBMS 를 활용해서 구현하고, 이를 해당 DBMS 상에서 최대한의 성능을 위해 튜닝하는 방안을 기술한다. 이를 위해 XBench 라는 XML 전용 벤치마크 데이터를 활용해서 XPath Accelerator 의 문제점과 향후 개선 방안에 대해서도 논의한다.

1. 서론

XML 은 최근 데이터 표현 및 전송을 위한 표준으로 급속히 자리잡아 가고 있으며, XSchema, XPath, XQuery 등의 다양한 관련 표준들이 제정되고 있다. 향후 더 많은 데이터들이 XML 형태로 생성/저장/관리될 필요가 있다. 따라서, 이러한 XML 데이터를 데이터베이스에 효과적으로 저장/검색 등의 지원을 위한 많은 방안들이 제시되었다.

하지만, 관계형 DBMS 가 실제적으로 차지하는 비중이나 시장상황을 고려했을 때, 관계형 DBMS 기반의 XML 지원이 주류가 될 것이다. 본 논문에서는 기존에 제안된 XPath Accelerator[1]라는 XML 인덱스 방안을 실제로 상용 DBMS 상에서 구현해보고, 해당 DBMS 에서 최대의 성능을 보장하기 위해 튜닝하는 방안을 제시한다. 또한, XBench 라는 XML 전용 벤치마크 데이터를 이용해서 구현한 XPath Accelerator 의 문제점과 개선점에 대해서도 논의한다.

논문 구성은 다음과 같다. 2 장에서는 XPath Accelerator 의 개념적 원리를 간단히 설명하고, 3, 4 에서는 이를 상용 DBMS 를 이용해서 구현한 내용과 이를 해당 DBMS 에서 튜닝 방안을 기술한다. 마지막으로, 5 장에서 결론과 향후 연구방안을 제시한다.

2. Accelerating XPath Location Steps

이 장에서는 [1]에서 제시한 XPath Accelerator 의 개념적 동작원리를 간단히 설명한다..

[1]에서 제시하고 있는 XPath Accelerator 는 먼저 XML 의 구조가 트리 형태로 되어 있기 때문에 각 엘레멘트를 트리의 한 노드로 생각하고 그 노드를 다른 노드와 구별 할 수 있는 값을 부여 한다. 이 특정한 값을 트리 검색 방법들 중에서 전위 탐색과 후위 탐색을 이용하여 값을 결정한다.

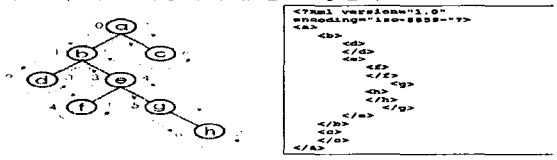


그림 1 Tree Traversal

우리는 이를 Node Descriptor 라고 표현 한다. Node Descriptor 가 포함 하고 있는 내용은 다음과 같다. 임의의 노드 v 에 대하여 v'을 v 의 parent 노드라고 하면 아래와 같이 표현 할 수 있다.

descriptor(v) = (pre(v), post(v), par(v), attr, tag)
par(v) = pre(v')
attr = true or false

임의의 노드 v 에 대하여 XPath 는 크게 Ancestor, descendant, preceding, following 4 가지로 구분할 수 있다. 그 외에도 세분화된 축들이 있지만 이는 본 논문에서 다루고자 주제는 아니므로 [1]을 참고하도록 한다.

Node Descriptor 를 구하게 되면 XPath 의 축과 Node Descriptor 의 정보를 이용하여 다음과 같은 XPath 를 SQL 로 재구성하는 것이 가능하다.

```
select extract(I.object_value, '/item')
from item I
where existsNode(I.object_value, '/item[@id="I1"]') = 1
```

이 XPath query 가 의미하는 내용은 다음과 같다.

node 'item'의 child 중 id 인 노드를 찾고 그 노드의 값이 'I1' 인 노드 'item'의 하위 노드를 모두 찾아라

이 내용과 같은 XPath Accelerator 에 대한 SQL 로 재구성 하면 다음과 같다.

```
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
```

이 SQL 를 설명하면

```
SELECT d2.pre, a3.tag, d2.val
FROM accel a1, accel a2, accel a3, xdata d1,
xdata d2
WHERE a1.tag='item' <-------- find Node 'item'
AND a1.pre=a2.par AND a2.tag='id' <-------- find Node 'id' child of 'item'
AND a2.pre=d1.pre AND d1.val='I1' <-------- find value of node 'id' , 'I1'
AND a1.pre<a3.pre AND a1.post>a3.post <-------- find nodes descendant of 'item'
AND a3.pre=d2.pre <-------- find values descendant nodes of 'item'
```

가 되며 이러한 방식을 이용하여 우리는 모든 XPath 를 이용한 질의 문을 SQL 로 재구성 할 수 있다.

3. XPath Accelerator 의 구현

서론에도 언급했듯이 XPath Accelerator 에 대한 자세한 구현내용을 찾기는 힘들다. 따라서 논문의 내용을 바탕으로 직접 구현한 내용을 간단히 언급해 보기로 한다. 실험에서는 Oracle 데이터베이스를 이용하였음을 밝혀 둔다.

XPath Accelerator 를 실험하기 위해서 가장 중요한 것은 XML Document 로부터 Node Descriptor 정보를 얻어 오는 것이다. 이 정보를 얻기 위해 SAX Parser 를 이용하여 XML Document 를 파싱하였다. SAX Parser 를 이용한 파싱방법과 구현내용은 웹상에서 향후 제공할 예정이다.

Oracle 상에는 특정 사용자를 생성하고 Node Descriptor 정보와 data 정보를 저장하기 위해 accel, xdata 라고 하는 테이블을 생성한다. accel 테이블에는 Node Descriptor 가 가지고 있는 정보를 그대로 저장하기 때문에 5 개의 컬럼이 필요하며, xdata 테이블은 pre 값을 저장하는 컬럼과 실제 데이터 값을 저장하기 위한 val 컬럼만이 존재한다.

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

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

XPath Accelerator 를 수행하기 위해 XPath 를 SQL 으로 재구성하는 경우 accel 테이블의 pre, post, par 와 xdata 테이블의 pre, val 컬럼은 가장 폭 넓게 쓰이게 되기 때문에 우리는 5 개의 컬럼에 각각 인덱스를 생성하였다. 원문에서는 pre 와 post 컬럼에만 인덱스를 생성하도록 하였지만 데이터가 커지면서 두 개의 인덱스만으로 수행할 경우 심각한 성능상의 문제를 초래하게 된다. 따라서 이는 튜닝의 요소라기 보다는 필요한 경우 인덱스를 이용할 수 있도록 하는 것이기 때문에 기본 스키마로 생각 하는 것이 바람직 하다.

그림 2 는 생성된 테이블과 인덱스에 대한 간단한 정보를 보여 주고 있다.

4. XPath Accelerator 튜닝

XPath Accelerator 의 개선방향으로 Staircase Join 과 같이 XML 데이터가 Tree 형태를 가지고 있다는 점에서 착안되는 방식이 있다.

이 논문에서는 XPath 가 SQL 로 재구성 되었기 때문에 결국은 XML 데이터에 대한 질의 수행은 RDB 방식이 되므로 이미 RDBMS 관점에서의 개선방향을 생각해 보고자 한다. 실험에 사용되는 데이터베이스는 Oracle 이며 데이터는 XBench 에서 제공하는 DCSD 데이터를 이용한다.

4-1 SQL query 수행의 문제점

query 의 수행상에서 속도의 저하를 속도저하의 원인을 분석하기 위해서는 수행계획을 먼저 살펴볼 살펴보자 XPath 를 XPath Accelerator 에 제시한 SQL 로 재구성하게 되면 가장 많이 나타나는 관계는 parent-child 관계와, 특정 노드의 descendant 를 구하는 질의 문이 된다. parent- child 관계에 있는 두 노드를 구하는 것은 SQL 로 재구성 하더라도 인덱스를 이용하여 매우 좋은 성능을 보이고 있다. 그러나 descendant 와 같은 조건을 가지는 쿼리는 공통적으로 Hash 조인이나 Sort 조인을 수행하면서 질의 문을 수행하는데 많은 시간이 걸리게 된다.

앞에서 예를 들었던 query 를 살펴 보면

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	

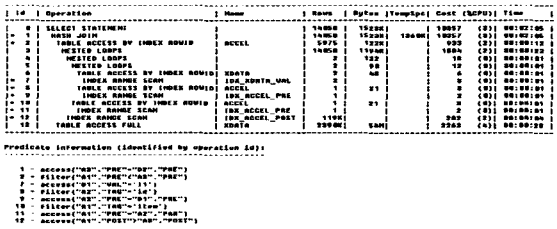
INDEX_NAME	LEAF_BLOCKS	DISTINCT_KEYS	CLUSTERING_FACTOR	NUM_ROWS
IDX_XDATA_VAL	8972	687364	1487277	1616183
IDX_XDATA_PRE	5520	2390112	59057	2390112
IDX_ACCEL_PAR	5510	846471	89249	2390112
IDX_ACCEL_POST	5520	2390112	12873	2390112
IDX_ACCEL_PRE	5520	2390112	82716	2390112

그림 2 Schema

```
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
```

테이블은 accel, xdata 두 개이며 index 는 각각 accel 의 pre, post, par 와 xdata 의 pre, val 컬럼을 참조하여 생성하였다.

이 쿼리의 수행시간은 테이블과 생성된 인덱스를 모두 analyze 하였을 경우에는 결과 튜플 수는 100 개가 되며 00:00:19:64 정도의 시간이 걸린다. 그러나 실제 수행된 시간은 실험 환경에 따라 변화할 수 있기 때문에 의미가 있는 것은 아니다. 수행계획을 살펴 보면 총 4 번의 조인이 발생하는데 그 중에 마지막 조인은 Hash 조인이 발생하게 된다. Hash 조인이 올바른 수행계획으로 선택하게 되는 원인은 a1.post>a3.post 조건의 cardinality 가 올바르지 않기 때문이다.



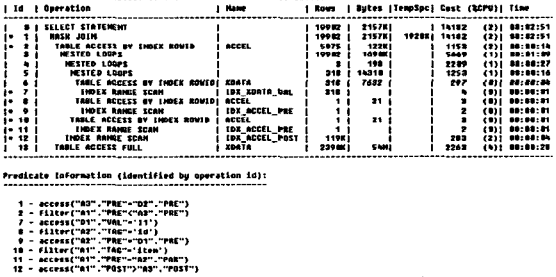
수행계획 1 Analyze 만 실행 한 경우의 수행계획

근본적으로 Oracle 의 최적화기가 Hash 조인을 선택하게 되는 원인은 위 그림에서 나와 있는 수행계획상의 ID12 인 ' INDEX RANGE SCAN' 에서의 cardinality 예측이 틀리기 때문이다. 최적화기는 119000 개의 튜플 수를 예측하고 있지만 실제로는 106 개가 올바른 튜플 수이다. 119000 개로 예측되었기 때문에 인덱스가 아닌 Full Table Scan 을 접근 방식으로 결정하게 되고 그 경우 Nested Loop 조인 보다 Hash 조인이 좋은 결과를 낼 수 있다고 판단하는 근거가 되는 것이다.

그러나 실제로 원하는 튜플 수가 106 개이므로 Hash 조인을 이용하는 것은 효과적이라고 할 수 없다.

4-2 XPath Accelerator 의 성능 개선

cardinality 가 정확하게 예측 되지 못하는 문제를 극복하기 위해 먼저 Histogram 을 생성하는 방법을 사용해 보았으나 수행시간이나 수행 계획 측면에서 효과를 볼 수 없었다. 수행시간 00:00:19.40 이었으며 수행계획도 전혀 변함이 없었다. 다음은 히스토그램을 생성하였을 경우에 대한 수행계획이다.



수행계획 2 Histogram 을 생성하였을 경우의 수행계획

따라서 다른 방법을 이용하여 수행시간을 단축시킬 필요가 있다. 우리가 한가지 알고 있는 사실 중에 하나는 실제로 우리가 원하는 결과로써 원하는 튜플 수는 100 여 개라는 사실이다. 이로부터 우리는 Full Table Scan 을 이용하는 것 보다 인덱스를 이용해 보는 것이 더 좋으며, 인덱스를 이용하게 된다면 Hash 조인 보다는 Nested Loop 조인을 이용하게 될 것이라고 예측해 볼 수 있다. 그러므로 Oracle 이 접근 방법으로 인덱스를 이용하도록 해보자. Oracle 이 접근 방법을 인덱스로 선택하게 하는 방법으로는 SQL 상에 직접 힌트를 이용하는 방법도 있지만 이번 논문에서는 Oracle 이 가지고 있는 parameter 들 중에서 optimizer_index_caching, optimizer_index_cost_adj 라는 두 개의 parameter 를 이용하였다.

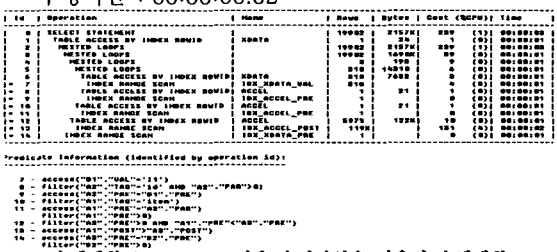
optimizer_index_caching 은 Oracle 은 인덱스를 이용할 경우의 비용을 계산할 때 인덱스 데이터 블록들은 하드디스크에서 하나씩 불러 온다고 가정한다. 그러나 이 parameter 를 100%라는 값을 저장하면 Oracle 은 인덱스의 모든 데이터 블록이

이미 버퍼에 올라와 있다고 가정하게 된다. 값으로는 50% ,30%도 가능하다.

optimizer_index_cost_adj parameter 는 인덱스를 이용할 때의 비용이 100 이라고 한다면 100 에 비해 어느 정도 비용이 인덱스를 이용할 때 필요한 것인가를 결정하게 된다. 만약 10 이라고 한다면 인덱스를 이용할 경우의 비용이 100%가 아닌 10%만 필요하다고 최적화기에게 알려 주는 것이다.

이번 논문에서는 조금 값을 과장해서 optimizer _index_cahcing 은 100 으로, optimizer_index _cost_adj 는 1 이라는 값을 저장하였다. 그 결과 다음과 같은 결과를 얻을 수 있었다.

수행시간 : 00:00:00.02



수행계획 3 parameter 값을 수정하였을 경우의 수행계획

수행계획도 마지막 Hash 조인이 없어 지고 Nested Loop 조인으로 바뀌었으며 조건 식의 수행 순서도 바뀌었다. 즉 Oracle 이 인덱스를 이용하였을 경우에 가장 최적의 비용이 든다고 가정하고 그 경우에 생성할 수 있는 최적의 수행계획을 결정한 것이다.

따라서 XPath 로 이루어진 질의 문을 XPath Accelerator 에서 제안한 방식으로 SQL 로 재구성하였을 경우 일반적인 equality 조건 식만으로 적용된 쿼리는 매우 빠른 속도로 결과를 얻어 올 수 있으나 그렇지 못한 descendant 와 같은 range 조건을 필요로 하는 경우에는 인덱스를 이용한 Nested Loop 조인을 이용할 수 있도록 하는 것이 보다 수행 시간을 단축 시킬 수 방법 중 하나이다.

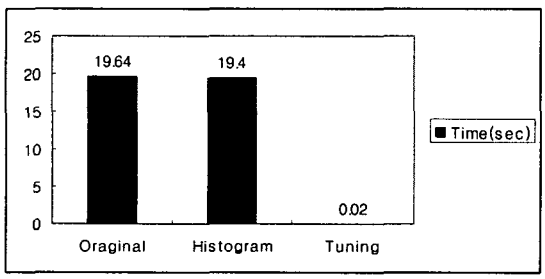


그림 3 수행 시간 결과

5. 결론 및 향후 연구

이번 실험이 모든 XPath 를 포함하고 있을 수는 없으나 제안된 XPath Accelerator 가 self 조인을 다수 포함할 수 밖에 없다는 점에서 descendant 를 구하는 것과 같은 range 조건을 포함한 SQL 문에서는 모두 공통적인 문제점을 내포 하고 있었다. 이 문제를 극복하기 위하여 보다 하나의 노드가 다른 자신의 자신과 관련된 특정 노드를 찾는 경우 전체 튜플 수에 비해 찾고자 하는 튜플 수가 작다는 점을 이용하여 INDEX 를 이용한 튜닝 방법을 제안한 것이다. 그 결과로 보다 향상된 수행속도를 얻을 수 있었다.

지금까지는 주어진 SQL 에 대한 튜닝의 관점이 포함되었다면 앞으로는 조금 더 폭 넓은 연구가 필요하다 첫 째로 인덱스 방식의 변화인데 Oracle 에서 제공하는 B+ Tree 외에 [6]에 기술된 MS 에서 제공하는 XML 데이터를 위한 인덱스 방식에 대한 연구가 이루어져야 한다. 다음으로 Stair Case Join[4]과 같은 XML Data 의 의미를 잘 이해하고 특성을 이용한 방식에 연구도 필요하다. 마지막으로, [5]에서 제시한 것 처럼, XML 데이터의 시맨틱을 활용해서 SQL 에서의 불필요한 조인 횟수를 줄이는 방안에 대한 연구도 필요하다.

참고 문헌

[1]. Torsten Grust, "Accelerating XPath Location Steps", In Proc. of the 21st Int'l ACM SIGMOD Conference, 2002.

[2]. Torsten Grust, Maurice Van Keulen, Jens Teubner, "Accelerating XPath Evaluation in Any RDBMS", TODS, 2004

[3]. Trsen Grust, Maurice van Keulen, Jens Teubner, "Staircase Join: Teach a relational DBMS to Watch its (Axis) Steps", VLDB, 2003

[4]. Torsten Grust, Maurice van Keulen, Jens Teubner. "Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps" VLDB, 2003.

[5]. Rajasekar, Raghav Kaushik, Jeffrey F Naughton, "Effeeicient XML-to-SQL Qeury Translation: Where to Add the Interligence?", VLDB, 2004

[6]. Shankar Pal, Istvan Cseri, Oliver Seeliger, Gideon Schaller, Leo Giakoumakis, Vasili Zolotovm, "Indexing XML Data Stored in a Relational Database", VLDB, 2004

[7]. Guy Harrison, "Oracle SQL High-Perfromance Tuning", Presntice Hall PTR.

[8]. SAX (Simple API for XML).http://sax.sourceforge.net/.

[9]. Oracle Co. "Everything You Always Wanted to Know about the Optimizer", Oracle Publisher

[10]. Ravi Murthy, Sandeepan Banerjee, "XML Schemas in Oracle XML DB", VLDB, 2003

[11]. the W3C XML Schema Standard (Schema Working Group), http://www.w3.org/XML/Schema

[12]. Benjamin Bin Yao, M.Tamer Ô zsu, Nitin Khandelwal, "XBench Benchmark and Performance Testing of XML DBMSs", Data Engineering, 2004