스트리밍 XML 데이타를 위한 효율적인 다중 질의 처리 기법

(An Efficient Multi-Query Evaluation Technique for Streaming XML Data)

민 준 기 † 박 명 제 †† 정 진 완 †††
(Jun-Ki Min) (Myung-Jae Park) (Chin-Wan Chung)

요 약 현재 스트리밍 XML 데이타에 대한 관심이 점차 증가한다. 스트리밍 XML 데이타에 대한 대부분의 연구는 XML 데이타를 효율적으로 여과하는 기법에 초점을 맞추었다. 이러한 XML 여과 시스템들은 사용자들이 관심 있는 XML 문서 전체를 사용자들에게 제공한다. 이 경우, 제공된 XML 문서들로부터 관심 있는 부분만을 추출하는 부담이 사용자에게 남겨지게 된다. 따라서, 스트리밍 XML 데이타에 대하여 직접적으로 질의 처리를 수행하여 관심 있는 XML 부분만을 추출하는 스트리밍 XML 질의 처리 기법들이 제안되었다. 그러나, 기존의 스트리밍 XML 질의 처리 기법들은 제안된 XPath 질의 만을 지원하며 복수 개의 질의 문을 처리하지는 못하고 있다.

본 논문에서는 스트리밍 데이타의 한 번 읽는 특성에 따라 XML 데이타를 한 번 읽으면서 복수 개의 질의들을 동시에 처리하는 XTREAM을 제안하고자 한다. 또한, XTREAM은 기존의 기법들에 비하여 순서 기반 프리디케이트 등 다양한 종류의 XPath 질의 기능들을 지원한다. 실제 XML 데이타와 합성 XML 데이타를 통한 실험 결과들은 XTREAM의 효율성과 확장성을 보인다.

키워드 : XML, 스트리밍 XML 데이타, 다중 질의 처리

Abstract Recently, there has been growing interest in streaming XML data. Much of the work on streaming XML data has been focused on efficient filtering of XML data. Such XML filtering systems deliver XML documents to interested users. The burden of extracting the XML fragments of interest from XML documents is placed on users. As a result, several evaluation techniques for streaming XML data, which only extract interested XML fragments by directly evaluating XML queries on streaming XML data, have been proposed. However, existing evaluation techniques for streaming XML data only support the restricted subset of XPath queries, and multiple queries cannot be evaluated by such evaluation techniques.

In this paper, we propose XTREAM which evaluates multiple queries in conjunction with the read-once nature of streaming data. In contrast to the previous work, XTREAM supports a wide class of XPath queries including order based predicates and so on. Experimental results with real-life and synthetic XML data demonstrate the efficiency and scalability of XTREAM.

Key words : XML, Streaming XML Data, Multi-Query Evaluation

1. 서 론

DBMS(DataBase Management System)와 같은 전형적인 데이타 처리 시스템들은 안정된 저장소를 통해 데이타를 관리하고 데이타의 수명(lifetime) 동안 데이타에 대한 질의를 처리한다. 하지만, 인터넷과 인트라넷의 확산에 따라, 스트리밍 데이타 처리(streaming data processing)와 같은 새로운 분야가 등장하게 되었다. 스트리밍 데이타 처리 분야에서는 정적인 데이타를 여러 번 반복하여 처리할 수 있는 기존의 분야와는 달리 데

· Acknowledge 본 연구는 정보통신부 및 정보통신연구진흥원의 대학 IT 연구센터 육성·지원 사업(IITA-2006-C1090-0603-0031)의 연구결과로 수행되었음

† 정 회 원 : 한국기술교육대학교 인터넷미디어 교수
jkmin@kut.ac.kr
†† 학생회원 : KAIST 전자전산학과
jpark@islab.kaist.ac.kr
††† 종신회원 : KAIST 전자전산학과 교수
chungcw@cs.kaist.ac.kr

논문접수 : 2006년 2월 16일
심사완료 : 2007년 3월 2일

이타가 연속적으로 끊임없이 입력으로 주어진다. 이 때, 사용자들은 스트리밍 데이타에 대한 사용자들의 관심사를 논리적으로 표현하여 먼저 등록하고, 질의 처리기는 사용자에 의해 논리적으로 표현된 관심사에 따라 연속적으로 주어지는 스트리밍 데이타를 처리한다.

다양한 애플리케이션(application) 분야를 통해 수집된 데이타의 통합과 공유에 따라, 정보 처리 상호 운용(inter-operability)에 대한 필요성이 증가하였으며, 그 결과, W3C에서는 XML(eXtensible Markup Language)[1]을 제안하였다. XML은 유연하고 자기설명적(self-describing)인 특성에 따라, 인터넷 상에서의 데이타 표현 및 교환의 표준으로 부각되었다.

최근에는, XML을 기반으로 하는 스트리밍 데이타(즉, 스트리밍 XML 데이타) 처리에 대한 연구가 진행되었다[2-8]. 다양한 XML 질의 언어 중에서 XPath[9]는 간결하면서도 XML 문서의 어떠한 부분도 표현하기에 충분하므로 사용자의 관심사를 표현하는 일반적인 질의 언어로 사용된다. 스트리밍 XML 데이타 처리에 관한 대부분의 기존 연구들은 제한된 XPath 표현식(expression)을 기반으로 XML 문서들을 여과하는(filtering) 기법에 초점을 맞추었다[2]. XML 여과 시스템은 관심 있는 사용자들에게 여과한 XML 문서들을 제공한다. 따라서, 제공된 XML 문서들로부터 필요한 부분을 추출하는 부담은 사용자들에게 부여된다. 또한 현재의 XML 여과 시스템들은 제한된 XPath 질의를 지원한다. 따라서, 브랜칭 경로 표현식(branching path expression), 순서 기반 프리디케이트(order based predicate), 그리고 중첩된 프리디케이트(nested predicate)와 같은 유용한 XPath 질의들은 지원하지 않는다.

위에서 언급한 바와 같이, 스트리밍 XML 여과 시스템은 문서 전체를 사용자에게 전달한다. 따라서, 스트리밍 XML 데이타에 대하여 질의를 처리하여 질의 결과 부분 만을 추출하는 질의 처리 시스템이 제안되었다. XSL[10]은 다양한 종류의 XPath 질의를 지원하지 않는다. 그리고, XSQ[8,11]는 한 번에 하나씩 제한된 형태의 XPath 질의만 처리한다. 그러나, 사용자들의 관심사는 다양하다. 예를 들면, 한 사용자는 삼성 전자의 현재 주식 시세와 인텔의 현재 주식 시세를 동시에 알기를 원한다. 따라서, 사용자의 다양한 관심사를 표현한 여러 XPath 질의들을 동시에 처리할 수 있는 다중 XPath 질의 처리기가 필요하다. 이러한 다중 질의 처리는 프로세서 할당 방식에 따라 질의마다 쓰레드(thread)를 할당할 수 있지만, 현재 알려진 그 어떠한 시스템도 광대한 수의 쓰레드를 충분히 처리하지 못하므로 스트리밍 XML 데이타 분야에 적용하는 것은 부적절하다.

본 논문에서는 XTREAM이라 불리는 스트리밍 XML 데이타 처리기를 제안하고자 한다. 기존의 연구와는 달리, XTREAM은 스트리밍 데이타의 한 번 읽는 특성에 따라 XML 데이타를 한 번 읽으면서 다양한 종류의 XPath 질의들을 동시에 처리한다. 본 논문의 특징은 다음과 같다.

- 다중 질의 처리: 앞에서 언급한 바와 같이, 기존의 연구들은 스트리밍 XML 데이타에 대해 한 번에 단일 질의를 처리하는 데에 초점을 맞추었다. 하지만, 사용자의 다양한 관심사를 만족시키기 위해서는 한 번에 다중 질의를 동시에 처리하는 기법이 필요하다. 따라서, XTREAM은 EStack과 LStack이라 불리는 두 스택(stack)을 사용하여 다중 질의 처리를 지원한다.
- 다양한 종류의 XPath 질의 지원: 일부 XML 여과 시스템들은 XPath 질의의 종류를 제한하였다. 하지만, 사용자의 다양한 관심사를 표현하기 위해서는 다양한 형태의 XPath 질의가 지원되어야 한다. XTREAM은 와일드카드(wildcard), 클로져(closure), 순서 기반 프리디케이트, 그리고 중첩된 프리디케이트를 포함한 다양한 종류의 XPath 질의를 지원한다.

XTREAM의 효율성과 확장성(scalability)을 검증하기 위해, 실생활에서 사용하는 XML 데이타와 인위적인 XML 데이타를 사용하여 실험을 수행하였다. 단일 질의 처리의 성능 비교는 단일 질의를 지원하는 기존의 스트리밍 XML 질의 처리기들과 단일 질의를 지원하는 XTREAM인 XNAIVE의 성능을 비교하였다. 그 결과, XNAIVE는 기존의 스트리밍 XML 질의 처리기보다 1.5배에서 7.5배 빠른 질의 처리 성능을 보였다. 다중 질의 처리 환경에서는 XNAIVE보다 다중 질의를 지원하는 XTREAM이 38.5배 빠른 질의 처리 성능을 보였다.

2. 관련연구

본 장에서는 XML 데이타의 기본적인 데이타 모델과 SAX(Simple API for XML)라 불리는 이벤트 기반 XML 파서에 대해서 설명한다. 그리고, 본 논문에서 지원하는 다양한 종류의 XPath 질의에 대해서 설명한다. 또한, 스트리밍 XML 데이타를 위한 기존의 여과 기법과 질의 처리기에 대하여 살펴본다.

2.1 XML 데이타 모델과 SAX 파서

스트리밍 XML 데이타는 질의 처리기가 XML 문서의 파싱 작업이 완전히 끝나기 전에 질의 처리를 수행할 수 있으므로, SAX 파서가 발생하는 이벤트들의 순서를 기반으로 XML 데이타를 모델링한다.

본 논문에서는 수정한 SAX 파서를 사용하여 XML 문서를 파싱한다. XML 문서 파싱의 시작 부분과 끝 부분에서 SAX 파서는 startDocument()와 endDocument() 이벤트를 각각 발생한다. 엘리먼트(element)의 시작 태그

(tag)와 끝 태그에 대해서, SAX 파서는 startElement (*level, T, offset*)와 endElement(*level, T, offset*) 이벤트를 각각 발생한다. 이때, *level*은 루트 노드(root node)로부터 해당 엘리먼트 노드까지의 깊이(depth)를, *T*는 태그 이름을, *offset*은 태그의 위치를 각각 의미한다. 마지막으로, SAX 파서는 데이타 값에 대해서 text (*level, V, soffset, eoffset*) 이벤트를 발생하며, *level*은 루트 노드로부터 데이타 값을 표현하는 말단 노드(leaf node)까지의 깊이를, *V*는 해당 데이타 값을, 그리고 *soffset*과 *eoffset*은 데이타 값의 시작 위치와 끝 위치를 각각 의미한다. 그림 1은 예제 XML 문서와 SAX 파서가 파싱하여 발생한 SAX 이벤트들을 보여준다.

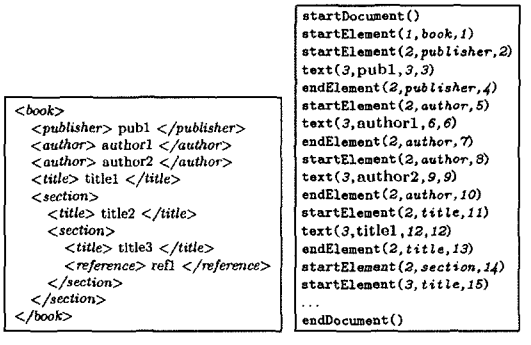


그림 1 XML 데이타 예와 SAX 이벤트

2.2 XPath

그림 2는 본 논문에서 지원하는 XPath의 구문들을 표현한 BNF이며, 이를 Minimal XPath라 명한다. Minimal XPath를 위한 문법은 축약된 XPath 구문을 기반으로 한다.

```
Minimal_XPath ::= ('/'|'//') RelPath
RelPath       ::= Step ('/'|'//') RelPath | Step
Step          ::= '.' | NodeTest Predicate*
NodeTest      ::= label | '*' | 'text()'
Predicate     ::= '['RelPath | Comp_Val | Comp_Pos']'
Comp_Val      ::= RelPath CompOP string
Comp_Pos      ::= number
CompOp        ::= '=' | '<' | '>' | '≤' | '≥'
```

그림 2 *Minimal XPath* 구문

Minimal XPath는 child와 descendant 축(axis), 엘리먼트와 애트리뷰트 라벨(label), 와일드카드, 그리고 중첩된 프리디케이트를 포함한 프리디케이트를 모두 지원한다. 각 스텝(step)은 '/'에 의해 연결되고, 스텝의 기본(default) 축은 child이다. 하지만, 축약된 구문에서는 표현하지 않는다. '//'는 descendant 축을 의미한다.[1] 프리디케이트는 존재 한정사 (existential quantifier) 로 사용한다. 예를 들면, //book[.//title]은 title 엘리먼트를 자손으로 가지는 모든 book 엘리먼트를 추출한다. 또한, Minimal XPath는 '[title=XML]' 형태의 값 기반(value based) 프리디케이트와 '[2]' 형태의 순서 기반 프리디케이트를 지원한다.

2.3 스트리밍 XML 여과 시스템

다양한 XML 여과 시스템들이 제안되었다. XFilter [2]는 처음으로 스트리밍 XML 데이타를 처리한 XML 여과 시스템이다. XFilter는 각 XPath 질의를 분리된 DFA(Deterministic Finite Automaton)로 변환한다. YFilter[5]는 접두사(prefix)를 공유하면서 XPath 표현식 집합(set)을 하나의 NFA(Non-deterministic Finite Automaton)로 변환한다.

XTier[4]는 프리디케이트를 포함한 트리 기반의 XPath 질의를 지원한다. XTier는 XPath 질의를 라벨 경로(sequence)의 서브 스트링(substring)들로 분해하고, 각 서브 스트링은 Substring-Table이란 테이블에 저장한다. 테이블 내에 있는 정보들은 부분 매칭(partial matching)을 확인하기 위해 사용한다. XPush [6]는 각 XPath 질의를 NFA로 변환하고, NFA 집합 내에 포함된 상태(state)들을 배합(grouping)하여 하나의 DPA(*Deterministic Pushdown Automaton*)를 생성한다. 이것은 NFA를 DFA로 변환하는 알고리즘[12]과 유사하다. 따라서, DPA에 포함된 상태 수는 대응하는 NFA 집합 내에 포함된 상태 수를 초과할 수 있다.

2.4 스트리밍 XML 질의 처리 시스템

스트리밍 XML 데이타에서 질의 처리를 수행하는 일부 질의 처리기들이 제안되었다. 하지만, XTREAM에 비교하여 볼 때, 제안된 대부분의 질의 처리기들은 제한된 XPath 구문을 지원한다. 다음의 표 1은 다양한 스트리밍 XML 데이타 질의 처리기들과 본 논문에서 제안하는 XTREAM의 지원 가능 기능들을 정리한 것이다.

XMLTK[13]는 위에서 언급한 XPush와 유사하게 입력 받은 XPath 문을 NFA로 변환하고 이를 다시 DFA로 변환하여 처리한다. 이때, DFA의 상태 수가 NFA에 비하여 지수적으로 증가할 수 있으므로 질의 처리 전에 DFA로 변환하는 것이 아니라 질의 처리 도중에 필요에 따라서 DFA로 변환하는 방법을 적용하고 있다. 현재 XMLTK는 특수한 경우의 순서 기반 프리디케이트와 애트리뷰트를 포함하는 프리디케이트만을 지원하고 이외의 프리디케이트는 지원하지 않는다.

XSQ[8,11]는 입력된 XPath 문은 푸시-다운 오토마

1) 정확하게 표현하면, '//'는 /descendant or self::node()/의 축약된 구문이다. 하지만, 기본 축(i.e., child)에 따라, '//'는 일반적으로 descendant 축으로 해석한다.

표 1 다양한 스트리밍 XML 질의 처리 시스템 비교

	다중 질의	'/' 지원	'//' 지원	애트리뷰트	값 기반 프리디케이트	순서 기반 프리디케이트	프리디케이트
XSM[10]	X	O	X	X	O	X	X
XMLTK[13]	X	O	O	O	X	O	X
XSQ[8,11]	X	O	O	O	O	X	△
SPEX[14]	X	O	O	X	X	X	△
ViteX[15]	X	O	O	X	X	X	△
XTREAM	O	O	O	O	O	O	O

타(push-down automata)로 변환한다. 이 변환에 있어서, 입력된 XPath 문을 스텝 별로 분리하고 각 스텝에 대하여 미리 정의된 템플레이트(template) 오타마타들로 대응 시킨다. 그 후 변환된 오토마타들을 조합하여 최종 질의 수행을 위한 푸시-다운 오토마타로 변환하고 스트리밍 XML 데이타의 SAX 파서로부터 전달되는 이벤트에 따라서 푸시-다운 오토마타의 상태 전이를 발생시키면서 질의를 수행하게 된다. XSQ는 제한된 프리디케이트를 포함한 XPath 질의를 지원한다. 하지만, XSQ는 한 번에 하나의 XPath 질의만 처리한다. 또한, XSQ는 순서 기반 프리디케이트와 프리디케이트 내의 descendant 축과 같은 다양한 XPath 질의들을 지원하지 않는다. 더욱이, XSQ는 질의 결과일 수도, 그렇지 않을 수도 있는 XML 부분(fragment)을 유지하기 위해서 대량의 메모리를 사용한다. 최근, SPEX[14]과 ViteX[15] 역시 제안되었지만, 두 질의 처리기 모두 다중 질의 처리를 지원하지 않는다. 또한, 두 질의 처리기 모두 순서 및 값 기반 프리디케이트를 지원하지 않으며 제안된 형태의(예. [title])의 프리디케이트만을 지원한다.

Bruno 등[16]은 다중 질의 처리에 대한 연구를 수행하였다. 그들의 인덱스 기반 방식은 IR(Information Retrieval) 분야에서 연구된 역 인덱스(inverted index)를 사용하여 질의 처리를 수행하므로 스트리밍 XML 데이타 분야의 한 번 읽는 특성을 정확하게 반영하지 못한다.

기존 연구와는 달리, XTREAM은 스트리밍 XML 데이타에 대해 다중으로 중첩된 프리디케이트 및 순서 기반 프리디케이트를 포함한 다양한 종류의 XPath 질의들을 처리한다. 또한, XTREAM은 사용자들의 다양한 관심사에 따라, 동시에 다중 질의를 처리하는 질의 처리 기법을 제공한다.

3. XTREAM

3.1 Minimal XPath를 위한 표현 기법

XTREAM에서는 2.2절에서 언급한 Minimal XPath 질의를 MXT(Minimal XPath Tree)라 불리는 트리 기반의 NFA로 표현한다. MXT의 각 노드는 XPath 표현식의 스텝이나 값 비교(value comparison) 부분을 나타내며 유일한 노드 식별자(*nid*)와 질의 식별자(*qid*)를 가진다. MXT에서 스텝 내의 프리티케이트에 대응되는 노드를 프리디케이트 노드(predicate node)라고 한다. 나머지 노드들은 탐색 노드(navigational node)라고 하며, 질의의 마지막 스텝에 대한 탐색 노드를 결과 노드(output node)라고 한다.

예제 1은 일부 Minimal XPath 질의들에 대한 MXT들을 보여준다.

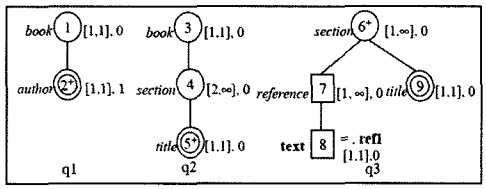
예제 1. 그림 1에서의 예제 XML 문서에 대해 다음과 같은 세 개의 Minimal XPath 질의들을 처리하고자 한다.

q1 = /book/author[1]

q2 = /book/*/section/title

q3 = //section[.//reference=ref1]/title

다음은 위에서 언급한 세 개의 Minimal XPath 질의들에 대한 MXT들이다.



예제 1에서, 프리디케이트 노드는 네모로 표현하고, 탐색 노드는 원으로 표현한다. 그리고, 이중 원은 결과 노드를 의미한다.

결과 노드는 Output Generator(3.2절 참조)가 생성한 잠재적인 질의 결과들을 생성한다. 잠재적인 질의 결과들이 실재 질의 결과인지에 대한 판단과 관심 있는 사용자들에게 질의 결과를 전달하는 과정은 판단 노드(decision node)에서 이루어진다. 예제 1에서와 같이, '+'에 의해서 표시된 판단 노드는 프리디케이트가 존재하는 질의에서 프리디케이트를 가지는 첫 번째 스텝과 관련이 있다.

q2와 같이 프리디케이트를 포함하지 않은 질의의 경우, 실재 질의 결과인지에 대한 판단이 필요하지 않으므로 결과 노드와 판단 노드는 동일하다. 하지만, 프리디케이트를 포함한 질의의 경우, 결과 노드가 생성한

XML 형태의 결과가 실제 결과인지를 결과 노드 상에서 판단할 수 없으므로, 판단 노드는 결과 노드와 다를 수 있다. 예를 들어 q3과 같은 경우, Query Evaluator (3.2절 참조)가 두 번째 title 엘리먼트를 만나면, q3의 결과 노드인 노드 9는 해당 엘리먼트를 결과로 생성한다. 하지만, q3의 첫 번째 스텝이 가지는 '[.//reference=ref1]' 프리디케이트는 아직 만족되지 않았다. 따라서, 질의 결과에 대한 판단은 첫 번째 section 엘리먼트의 끝 태그에 대한 이벤트가 발생할 때까지 미룬다.

예제 1에서 살펴본 바와 같이, 각 노드는 스텝에 관련된 정보를 표현하고자 다음과 같은 *type, label, level_gap,* 그리고 *position* 항목들을 가진다.

type : 노드의 종류(예, 프리디케이트 노드와 결과 노드)를 의미.

label : 스텝의 이름 테스트(NameTest)를 위해 필요.

level_gap : 항목은 축을 표현하고자 사용. XML 문서를 표현한 트리에서, 부모와 자식 간의 레벨 차이는 1이므로, child 축의 *level_gap*은 [1,1]로 표현하며 조상과 자손 간의 레벨 차이는 1과 동일하거나 1보다 크므로 descendant 축의 *level_gap*은 [1,∞]임.

position : '[2]'와 같은 순서 기반 프리디케이트를 처리하기 위해 필요. 순서 기반 프리디케이트를 가지지 않는 스텝들의 *position* 값들은 0임.

이외에 값 비교를 위한 노드는 비교 연산자(예, '='), 문자열 값, 그리고 *level_gap* 항목을 가진다. 값 비교 노드에 대한 *label*은 text이다.

3.2 XTREAM의 구조

XTREAM의 전체 구조는 그림 3에서 보여주며, XTREAM의 핵심 부분은 Query Evaluator이다. 개념적으로, XML Parser가 발생한 이벤트들에 따라, Query Evaluator는 XPath Parser가 생성한 오토마타의 스테이트들을 변경한다.

그림 3에서 알 수 있듯이, XTREAM은 다음과 같은

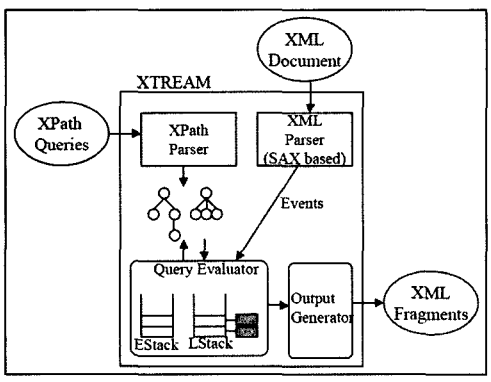


그림 3 XTREAM 구조도

XPath Parser, XML Parser, Query Evaluator, 그리고, Output Generator로 구성된다.

XPath Parser: 3.1절에서 설명한 바와 같이, 2.2절에서 언급한 Minimal XPath 질의를 파싱하여 트리 기반의 NFA인 MXT를 생성함.

XML Parser: 2.1절에서 설명한 바와 같이, 수정한 SAX 파서를 사용하여 XML 문서를 파싱하여 SAX 이벤트들을 순서대로 발생함.

Query Evaluator: EStack(Evaluated Stack)과 LStack (Look-Forward Stack)이라 불리는 두 스택들을 사용하여 수정한 SAX 파서가 발생한 이벤트들에 따라 MXT들을 탐색하면서 질의 처리를 수행함.

Output Generator: XML 형태로 표현된 Minimal XPath 질의들의 결과들을 생성함.

XTREAM이 동시에 다중 질의를 처리함에 따라, 다중 스테이트들은 이벤트에 의해 활성화(activate)된다. 일부 스트리밍 XML 데이타 처리기들[5,6,8,16]은 현재 활성화되는 스테이트들을 유지하기 위해 하나의 스택을 사용한다. 이와 반대로, XTREAM은 EStack과 LStack이라 불리는 두 스택들을 사용한다. EStack은 활성화되는 스테이트들의 정보를 유지하고, LStack은 앞으로 발생할 이벤트들에 의해서 활성화될 수 있는 스테이트들의 정보를 유지한다. 이에 따라, LStack 내에 있는 일부 스테이트들이 이벤트에 따라 활성화된다(기본적인 처리 방법에 대해서는 3.3절에서 자세히 다룬다).

추가적으로, 그림 3의 XTREAM 구조도는 관심 있는 사용자들에게 XML 형태의 결과를 유포(dissemination)하는 과정을 포함하지 않는다. 유포 과정은 질의와 사용자 간의 관계를 관리하여 간단히 처리할 수 있으므로, 본 논문에서는 유포 과정과 관련된 내용을 더 이상 언급하지 않는다.

3.3 XTREAM 질의 처리 알고리즘

3.2절에서 설명한 바와 같이, Query Evaluator는 수정한 SAX 파서가 발생한 이벤트들에 따라 MXT들을 탐색하면서 질의 처리를 수행한다. XML 문서를 파싱하는 동안, startElement 이벤트는 해당 이벤트와 대응하는 스텝들을 가지는 MXT들의 노드들의 집합을 활성화한다. 그리고, endElement 이벤트가 발생하면, 이전에 활성화된 노드들(previously activated nodes)로 되돌아가야 한다. 그러므로, XTREAM은 EStack과 LStack이라 불리는 두 스택들을 사용하여 연속된 SAX 이벤트들을 처리한다. EStack은 현재와 이전에 활성화된 MXT들의 노드들을 관리한다. 만약 Query Evaluator가 start-Element 이벤트가 발생하는 순간마다 MXT들을 탐색하고 대응하는 노드들을 찾는다면, 질의 처리 성능은 저하된다. 다중 트리 탐색에 따른 오버헤드(overhead)를

줄이기 위해서, 앞으로 발생 가능한 이벤트들에 의해 활성화될 노드들의 집합을 LStack의 상단(top)에 유지한다. 이에 따라, LStack 내에 있는 일부 상태들이 이벤트에 따라 활성화되어 EStack의 상단(top)에 놓이게 된다. 추가적으로, LStack은 만족한 프리디케이트 노드들을 해당 부모 노드들에게 알려주는 역할을 수행한다.

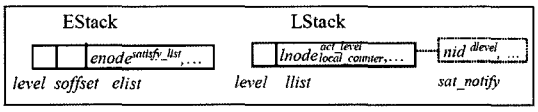


그림 4 EStack과 LStack의 레코드

그림 4는 EStack과 LStack의 레코드(record) 들에 대한 데이타 구조를 보여주며, 다음과 같은 특징을 가진 항목들로 구성된다.

level: EStack과 LStack의 각 레코드의 *level*은 스택에 레코드를 삽입하는 이벤트의 레벨임.

soffset: EStack의 레코드 내의 *soffset*은 레코드를 삽입하는 이벤트(예, startElement 또는 text)의 *offset* 값으로써, Output Generator의 입력으로 사용함.

elist, llist: EStack과 LStack의 레코드들은 *elist*와 *llist*라 불리는 노드 리스트를 가지며, *elist*와 *llist* 내의 노드들은 노드들의 *nid*가 적은 순서대로 정렬함. 또한, 각 리스트는 동일한 노드가 다른 엘리먼트들과 대응할 수 있으므로, MXT들의 노드들 대신 노드들의 실행 시간 이미지(run-time image)들, 다시 말해 다른 상태들로 구성된 실재 노드들을 유지함. (예, 예제 1에서의 노드 6은 그림 1의 예제 XML 문서의 첫 번째 section과 두 번째 section에 대응함.)

enode, lnode: *elist*와 *llist*에 포함되는 실행 시간 이미지들을 각각 *enode*와 *lnode*라고 함.

satisfy_list: 만약 노드가 프리디케이트 노드이거나 자식으로 프리디케이트 노드들(예, 예제 1의 노드 6)을 가진다면, 대응하는 *enode*는 만족된 프리디케이트 노드들을 식별하는 *satisfy_list*를 사용함.

act_level: *lnode*는 *lnode*의 부모에 대응하는 엘리먼트의 레벨을 식별하고자 *act_level*을 사용함. 그 결과, 축의 조건은 대응하는 엘리먼트의 레벨과 *act_level* 간의 차이를 통해 처리함.

local_counter: *position* 값이 0이 아닌 노드(예, 노드 2)에 대한 *lnode*는 엘리먼트의 순서를 관리하기 위해 *local_counter*를 사용하며, 각 *lnode*의 *local_counter* 값은 0으로 초기화함.

sat_notify: 각 *llist*는 프리디케이트 노드가 만족된 이전에 활성화된 노드들을 식별하기 위해서 *sat_notify* 항목을 가지며, *sat_notify*는 프리디케이트 노드가 만족된 레벨을 나타내는 *dlevel*과 함께 만족된 프리디케이트 노드의 *nid*를 유지함.

앞으로의 본 논문에서는 *enode*들과 *lnode*들이 노드들의 실행 시간 이미지들이므로, *enode, lnode*, 그리고 노드라는 단어를 자유롭게 사용한다.

Query Evaluator는 이벤트의 발생에 따라 스택들의 상태를 변화시키는 startElement, endElement, 그리고 text 이벤트 처리기들을 가진다. startElement 이벤트 처리기는 3.3.1절에서 설명하고, endElement 이벤트 처리기는 3.3.2절에서 설명한다. 하지만, text 이벤트 처리기 알고리즘은 startElement와 endElement 이벤트 처리기 알고리즘들에 값 비교를 위한 코드를 추가한 알고리즘과 유사하므로, 본 논문에서는 text 이벤트를 위한 이벤트 처리기 알고리즘은 생략한다.

3.3.1 startElement 이벤트 처리기 알고리즘

그림 5는 startElement 이벤트를 위한 이벤트 처리기 알고리즘을 보여준다.

```
Procedure StartHandler(startElement(level,T,offset))
begin
1.  new_elist := {}
2.  for each l ∈ LStack.top().llist do{
3.    if( level - l.act_level ∈ l.level_gap∧(l.label=T ∨ l.label='*')) {
4.      if(l.position != 0) l.local_counter++
5.      if(l.local_counter = l.position ∨ l.position = 0){
6.        if(new_elist.find(l.nid) = NULL)
7.          { e := new enode(l), new_elist.add(e) }
8.      }
9.    }
10. }
11. if (new_elist!={ } ) {
12.   EStack.push(level,offset,new_elist)
13.   new_llist := {}
14.   for each l ∈ LStack.top().llist do{
15.     if(level - l.act_level < l.level_gap.max) {
16.       l' := new lnode(l), l'.act_level := l.act_level
17.       new_llist.add(l')
18.     }
19.   }
20.   for each e ∈ EStack.top().elist do{
21.     for each child c of MXT's node n where n.nid = e.nid do{
22.       c' := new lnode(c), c.act_level := level, new_llist.add(c')
23.     }
24.   }
25.   LStack.push(level,new_llist)
26. }
end
```

그림 5 startElement 이벤트 처리기 알고리즘

처음에, 모든 MXT들의 루트 노드들을 LStack에 등록한다. 앞에서 설명한 바와 같이, LStack의 상단에는 활성화될 수 있는 노드들에 관한 정보가 있다. 따라서, 이벤트 처리기는 LStack의 상단에 위치한 이벤트에 의해 활성화된 노드들을 선택한다(라인 (1)-(10)). 추가적으로, 새로운 레코드가 EStack에 삽입될 경우, 그에 대응하는 LStack의 레코드 역시 생성된다(라인 (11)-(26)).

그림 5의 알고리즘 시작 부분에서, LStack의 상단(예, LStack.top())에 있는 각 *lnode*에 대한 축과 노드

테스트(NodeTest)에 관한 조건들을 확인한다(라인 (3)). 만약 축과 노드 테스트를 만족하는 *lnode* *l*이 순서 기반 프리디케이트를 가진다면, *l*의 *local_counter*에 1을 더한다(라인 (4)). 그 후, 순서 기반 프리디케이트를 만족하거나 대응하는 노드가 순서 기반 프리디케이트를 가지지 않을 경우, 해당 노드를 활성화한다(라인 (5)-(8)). 조상들이나 자손들과 동일한 태그의 엘리먼트를 가지는 귀납형 엘리먼트(recursively typed element) 들에 따라, 동일한 *nid*를 가지는 *lnode*들이 LStack.top()에 존재할 수 있다. 이러한 경우, 본 알고리즘은 이와 같은 *lnode*들에 대응하는 오직 하나의 *enode*만을 생성한다(라인 (6)-(7)).

만약 LStack.top()에 위치한 일부 노드들이 활성화된다면(라인 (11)), 활성화된 노드들을 위한 새로운 레코드를 이벤트의 *offset* 값과 함께 EStack에 삽입한다(라인 (12)). 그 후, 현재 EStack.top()에 위치한 새롭게 활성화된 노드들을 위한 새로운 LStack 레코드를 생성한다. 이에 따라, LStack.top()에 위치한 일부 *lnode*들은 앞으로 방문할 엘리먼트들과 대응한다. 만약 현재 방문하는 엘리먼트의 레벨과 *l*의 *act_level*간의 차이가 *l*의 *level_gap*의 최대값보다 적으면, *l*은 다음에 발생할 이벤트에 의해 활성화될 것이다(라인 (14)-(19)). 또한, 활성화된 노드들의 자식 노드들은 다음에 발생할 이벤트에 의해 활성화될 수 있다(라인 (20)-(24)). 라인 (14)-(19)와 라인 (20)-(24)에 따라, 만약 현재 방문하는 엘리먼트가 귀납형 엘리먼트라면, 동일한 *nid*를 가지지만 다른 *act_level* 값들을 가지는 *lnode*들이 발생할 수 있다. 마지막으로, 이러한 노드들을 LStack에 삽입한다(라인 (25)).

3.3.2 endElement 이벤트 처리기 알고리즘

endElement 이벤트를 위한 이벤트 처리기 알고리즘은 그림 6에서 보여준다. 본 알고리즘에서 OG는 Output Generator를 의미한다.

만약 EStack.top()의 *level*이 endElement 이벤트의 레벨과 동일하지 않다면, EStack.top()에 위치한 어떠한 노드도 endElement 이벤트를 초래한 엘리먼트와 대응하지 않는다. 따라서, 이벤트 처리기는 아무런 작업도 수행하지 않는다(라인 (1)). 만약 그렇지 않다면, 엘리먼트의 자손 엘리먼트가 발생할 이벤트들이 더 이상 존재하지 않으므로 LStack의 상단에 있는 레코드를 제거한다(라인 (2)). 그 후, 이벤트 처리기는 EStack.top()에 위치한 노드들의 *type*에 따라 이벤트를 처리한다(라인 (3)-(19)).

만약 EStack.top()에 위치한 *enode* *e*의 *type*이 결과 노드의 종류와 동일하다면, Output Generator는 대응하는 XML 부분을 [EStack.top().*soffset,offset*] 값으로 유지한다(라인 (4)-(5)). 이때, 다른 프리디케이트들을 확

```
Procedure EndHandler(endElement(level,T,offset))
begin
1.  if(EStack.top().level != level) return
2.  LStack.pop()
3.  for each e ∈ EStack.top().elist do {
4.    if(e.type = output node)
5.      OG.send(e.qid,EStack.top().soffset,offset)
6.    if(e.type = predicate node ∧ e has no child)
7.      e.satisfy_list[0] := true; //for Predicates such as '[title]'
8.    if(e.satisfy_list = NULL) {
9.      if(e.type = decision node)
10.       OG.output(e.qid,EStack.top().soffset,offset)
11.   } else {
12.     if(all entities of e.satisfy_list are true) {
13.       if(e.type = predicate node)
14.         LStack.top().sat_notify.add(e.nid^EStack.top().level)
15.       if(e.type = decision node)
16.         OG.output(e.qid,EStack.top().soffset,offset)
17.     } else {
18.       if (e.type != predicate node)
19.         OG.discard(e.qid,EStack.top().soffset,offset)
20.     }
21.   }
22. }
23. EStack.pop()
24. for each l ∈ LStack.top().llist
25.       where l.nid is in LStack.top().sat_notify as nid^dlevel do {
26.   if(dlevel -l.act_level ∈ l.level_gap) {
27.     if(l.act_level < LStack.top().level)
28.       LStack.top(-1).sat_notify.add(nid^dlevel)
29.     else if( l is the kth child of node p and p ∈ EStack.top().elist)
30.       p.satisfy_list[k] := true
31.     LStack.top().llist.remove(l)
32.   }
33. }
34. LStack.top.sat_notify.remove_all()
end
```

그림 6 endElement 이벤트 처리기 알고리즘

인할 필요도 존재하므로 결과에 대한 판단은 아직 수행하지 않는다.

만약 '[title]'과 같이 *e*의 *type*이 프리디케이트 노드이고 *e*가 자식 노드를 가지지 않는다면, 해당 프리디케이트의 만족을 표현하고자 *satisfy_list*를 *true*로 설정한다(라인 (6)-(7)). 또한, 만약 *e*가 자식으로 프리디케이트 노드들을 가지지 않고(예, *e.satisfy_list*가 NULL인 경우) *e*가 판단 노드와 동일하다면(예, 예제 1의 노드 2와 5), Output Generator는 질의 결과를 위한 추가 처리 과정이 필요하지 않으므로 사용자들을 위한 XML 결과를 바로 생성한다(라인 (8)-(10)).

*e*가 프리디케이트 노드와 동일하거나 자식으로 프리디케이트 노드들을 가진다고 가정해 보자(라인 (11)-(21)). 만약 자식으로 가진 프리디케이트 노드들의 프리디케이트 조건들이 만족하고 *e* 자신이 프리디케이트 노드라면(예, 예제 1의 노드 7), 이벤트 처리기는 LStack.top()에 EStack.top().*level*과 *e*의 *nid*를 추가함으로써 *e*에 의한 프리디케이트 조건의 만족을 표현한다(라인 (13)-(14)). EStack.top().*level*은 *nid*의 *dlevel*로 사용한다. 이러한 방법을 통해 Query Evaluator는 '[section[title]]'과 같이 중첩된 프리디케이트들을 처리한다. 프리디케이트 노드 *e*가 자식 노드를 가지지 않는 경우는 라인 (13)-(14)에서 만족 여부를 확인한다. 만약 *e*가 판단 노드와 동일하고 자식으로 가지는 모든 프리디케이트 노드들을 만

족한다면, Output Generator 내의 XML 부분들은 질의 결과이다. 만약 프리디케이트 노드가 아닌 *e*가 표현한 스텝의 일부 프리디케이트들을 만족하지 않는다면, 질의 결과로 가능성이 있다고 판단된 질의 결과는 제거한다(라인 (17)-(20)). 하지만, 만약 *e*가 프리디케이트 노드라면, *e*는 앞으로 발생할 이벤트가 만족할 것이다. 따라서, 질의 결과를 위한 판단은 여전히 수행하지 않는다.

EStack의 상단에 있는 레코드를 꺼냄으로써, 이전에 활성화된 레코드를 다시 활성화한다(라인 (23)). 라인 (24)-(33)은 프리디케이트 노드들의 만족 여부를 다시 활성화한 레코드에 반영하기 위해 수행한다. LStack.top()에 위치한 각 *lnode l*의 *nid*가 *sat_notify*의 *nid*일 경우, *l*의 *level_gap* 조건을 확인한다. 여기서, *level_gap*은 *l*의 활성화된 레벨로부터 *l*이 만족된 레벨까지의 거리를 의미한다. 만약 *l*이 *level_gap* 조건을 만족한다면, *l*이 표현한 프리디케이트 노드에 대한 확인이 더 이상 필요하지 않으므로 *l*을 제거한다(라인 (31)). 추가적으로, 만약 *lnode l*이 이전에 활성화된 레코드로부터 발생한다면, 만족한 프리디케이트의 정보를 LStack.top(-1)에 위치한 이전 스텝의 레코드들에게 적용한다(라인 (27)-(28)). 그렇지 않을 경우, *l*의 부모 노드는 EStack.top()에 존재한다. 따라서, 이벤트 처리기는 EStack.top()에 위치한 부모 노드에 프리디케이트 노드의 만족 여부를 표현한다(라인 (29)-(30)). 그 후, *sat_notify* 내의 모든 정보를 제거한다(라인 (34)).

3.4 XTREAM 질의 처리 예제

그림 1에서 보여진 XML 데이타에 대하여 예제 1에 대한 Query Evaluator의 처리 과정은 그림 7에서 보여준다. LStack의 윗첨자는 각 *lnode*의 *act_level*을 의미하고, 일부 *lnode*들의 아랫첨자는 *local_counter*를 의미한다. EStack의 윗첨자 'p'는 프리디케이트 조건의 만족을 나타낸다.

EStack과 LStack의 초기 상태는 SCENE 0과 같다. book 엘리먼트의 startElement 이벤트 발생으로 인해 노드 1과 3을 활성화한다(SCENE 1). publisher 엘리먼트가 LStack.top()에 위치한 어떠한 노드와도 대응하지 않으므로, 스택 레코드들의 상태에는 어떠한 변화도 발생하지 않는다. 첫 번째 author 엘리먼트의 startElement 이벤트가 발생한 경우, 노드 2의 *local_counter*를 증가하고, 노드 2를 활성화한다(SCENE 2). 첫 번째 author 엘리먼트의 endElement 이벤트가 발생한 후에 q1의 순서 프리디케이트 '[1]'를 만족하므로, 첫 번째 author 엘리먼트에 대한 *offset* 쌍인 [5,7]를 생성한다(SCENE 3).

첫 번째 section 엘리먼트를 방문한 경우, 노드 6만 활성화한다. 노드 4의 라벨이 section일지라도, 첫 번째

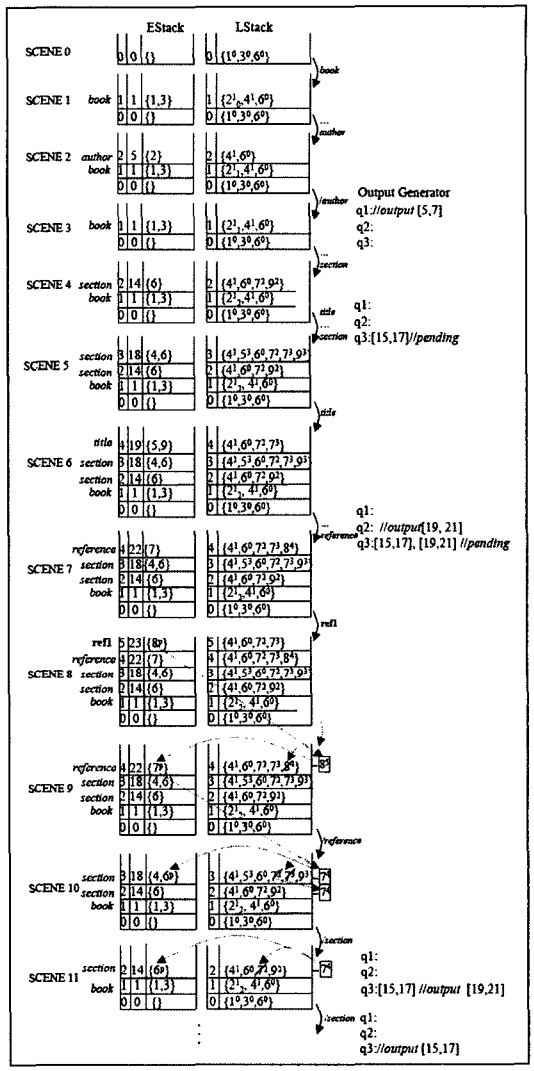


그림 7 Query Evaluator의 처리 과정

section의 레벨(=2)과 *act_level*(=1) 간의 차이가 1이고 노드 2의 *level_gap*이 [2,∞]이므로, 노드 4는 활성화하지 않는다(SCENE 4). 두 번째 title 엘리먼트는 q3의 결과 노드인 노드 9와 일치한다. 따라서, Output Generator는 두 번째 title 엘리먼트에 대한 *offset* 쌍인 [15,17]을 유지한다. 하지만, 프리디케이트 조건의 만족을 판단할 수 없으므로 *offset* 쌍을 생성하지 않는다. Query Evaluator는 이러한 방법으로 이벤트들의 발생을 처리한다.

데이타 값인 'ref1'이 노드 8을 만족할 경우, 프리디케이트 조건의 만족을 'p'로 표현한다(SCENE 8). 그 후, 노드 8의 만족 결과를 부모 노드(예, 노드 7)에게 전달한다(SCENE 9). 또한, 노드 8은 더 이상 확인할 필요

가 없으므로, 노드 8을 LStack.top()에서 제거한다. 노드 7 역시 프리디케이트 노드이므로, 노드 7의 만족 결과를 부모 노드에게 전달한다(SCENE 10). 귀납형 엘리먼트인 section 엘리먼트의 첫 번째 section 엘리먼트는 프리디케이트 '[.//reference=ref1]'를 만족한다. 따라서, 노드 7의 만족 결과를 LStack.top() 아래에 위치한 레코드에 전달한다.

만약 프리디케이트가 '[reference=ref1]'이라면, 자손 노드들 대신 단지 자식 노드만 포함하므로 프리디케이트의 만족 결과는 전달하지 않는다. 이러한 경우, 프리디케이트 노드의 *act_level*이 LStack.top()의 level과 동일하므로(그림 5의 라인 (22) 참고) 만족 결과를 전달하지 않는다(그림 6의 라인 (27)-(28) 참고).

결과적으로, 발생한 endElement 이벤트들에 따라 레코드들을 스택에서 꺼냄으로써, 스택들의 상태는 SCENE 0으로 되돌아간다.

4. 성능 평가

XTREAM의 성능 평가를 위해서, 기존의 스트리밍 XML 데이타 처리기들인 XMLTK[2]와 XSQ[3]의 성능과 XTREAM의 성능을 비교하였다. 하지만, XMLTK와 XSQ는 단일 질의 처리만을 지원한다. 따라서, XTREAM의 다중 질의 처리에 대한 성능 비교를 위해서, 한 번에 하나의 질의만 처리하는 XTREAM의 단일 질의 처리기인 XNAIVE의 성능과 XTREAM의 성능을 비교하였다.

4.1 실험 환경

성능 실험은 메인 메모리 1GB에 MS Windows XP가 탑재된 펜티엄 4급, 2.66 GHz를 사용하였고, XML 데이타들은 로컬 디스크에 저장하였다.

4.1.1 데이타 세트(Data Set)

XMark 데이타 세트과 Shakespeare 데이타 세트를 사용하여 실험을 수행하였다. 각 데이타 세트는 다섯 종류의 XML 문서들을 포함한다. 본 실험에서 사용한 데이타 세트들의 특징은 표 2에서 보여준다. 설명은 각 XML 문서가 어떻게 생성되었는지를 나타낸다.

XML 벤치마킹 프로젝트 일환으로 개발한 XMark [17] 데이타를 사용한다. 크기 조절 인수(scaling factor)인 f의 값을 조절함으로써, XMark 데이타 세트를 위한 5 종류의 XML 문서들을 생성한다.

Shakespeare[18] 데이타는 셰익스피어가 쓴 희곡들로, 실생활에서 사용하는 XML 데이타이다. SH1은 "The Tempest"라 불리는 희곡을 포함한다. SH2는 셰익스피어의 4대 비극인 Hamlet, Macbeth, Othello, 그리고 King Lear를 포함하고, SH3는 12 희극을 포함한다. 셰익스피어의 모든 희곡들은 SH4에 포함한다. 그리고, 큰 XML 데이타에 대한 XTREAM의 성능을 측정하기 위하여 SH4를 10배 확장하여 SH5를 생성한다.

표 2 XML 데이타 세트

데이타 세트	이름	크기(MB)	설명
XMark	XM1	0.03	f = 0.0001
	XM2	0.19	f = 0.001
	XM3	1.18	f = 0.01
	XM4	11.88	f = 0.1
	XM5	118.55	f = 1
Shakespeare	SH1	0.16	1 희곡
	SH2	0.97	4 희곡
	SH3	2.61	12 희곡
	SH4	7.89	37 희곡
	SH5	78.95	37 희곡 X 10

4.1.2 XPath 질의 세트

인위적으로 생성한 XPath 질의들로 실험을 수행하였다. 질의들은 YFilter[5]의 XPath 질의 생성기[4]를 새롭게 수정한 XPath 질의 생성기를 통해 생성하였다.

표 3 예제 XPath 질의

데이타 세트	예제 질의
XMark	/*//bidder[date]//increase
	/site[1]/people/person[watches/watch]/name
	/site/open_auctions[1]/open_auction[quantity]/*
	/site[open_auctions/open_auction//type]/*/*[@id=3]
Shakespeare	/*[PLAYSUBT]/*[3]
	/*[1][TITLE]//PERSONAE/TITLE
	/PLAY/PERSONAE[3][PGROUP/*]/*
	/PLAY[PERSONAE/PERSONA]//TITLE[4]

XPath 질의 생성기가 생성한 XPath 질의에 대한 이해를 돕고자, 새롭게 수정한 XPath 질의 생성기가 생성한 일부 XPath 질의들을 표 3에서 보여준다. XPath 질의 생성기에서 사용한 매개 변수들과 변수의 값들에 관한 정보는 표 4에서 보여준다.

XTREAM의 Query Evaluator의 확장성에 대한 성능을 평가하고자, 표 4에서 설명한 매개 변수들의 값들

표 4 XPath 질의 생성기의 환경 설정

매개 변수	변수 값
질의 최대 깊이	6
와일드카드('*') 생성 확률	20%
descendant 축('//') 생성 확률	20%
값 기반 프리디케이트 수	1

2) http://www.cs.washington.edu/homes/suciu/XMLTK/에서 제공
3) http://www.cs.umd.edu/projects/xsq/에서 제공
4) http://www.cs.berkeley.edu/~diaoyl/yfilter/index.htm에서 제공

을 사용하여 100, 300, 1000, 3000, 그리고, 10000 개의 질의들을 생성하였다. 추가적으로, 기존의 스트리밍 XML 데이타 처리기들인 XMLTK와 XSQ와 XTREAM의 성능 비교를 위하여, 생성한 XPath 질의들 중에서 일부 유형들을 추출하였다. 본 실험에서는 각 유형별로 100개의 XPath 질의들을 실행하여 평균 질의 처리 시간을 측정하였다. 각 유형에 대한 자세한 설명은 표 5에서 보여주며, 표 5의 질의 예제는 대응하는 XPath 질의들을 보여준다.

표 5 XPath 질의 유형 및 예제

질의 유형	질의 예제
유형 1	//PGROUP/PERSONA
유형 2	/PLAY/PERSONAE[1]/TITLE
유형 3	//PERSONAE[TITLE]//PERSONA
유형 4	/PLAY/PERSONAE[PERSONA]/*[5]

본 실험에서는 다음과 같은 이유를 기반으로 4 가지 질의 유형을 선택하였다. 유형 1은 어떠한 프리디케이트들도 가지지 않는 일반적인 경로 표현식들에 대한 질의 처리 성능을, 유형 2는 순서 기반 프리디케이트를 가지는 경로 표현식들에 대한 성능을, 그리고 유형 3은 순서 기반 프리디케이트 외의 프리디케이트들을 가지는 경로 표현식들에 대한 성능을 평가한다. 마지막으로, 각 질의 처리기가 처리할 수 있는 XPath 질의의 범위를 확인하고자, 순서 기반 프리디케이트와 와일드카드를 포함한 프리디케이트들을 가지는 경로 표현식들로 구성된 유형 4를 선택하였다.

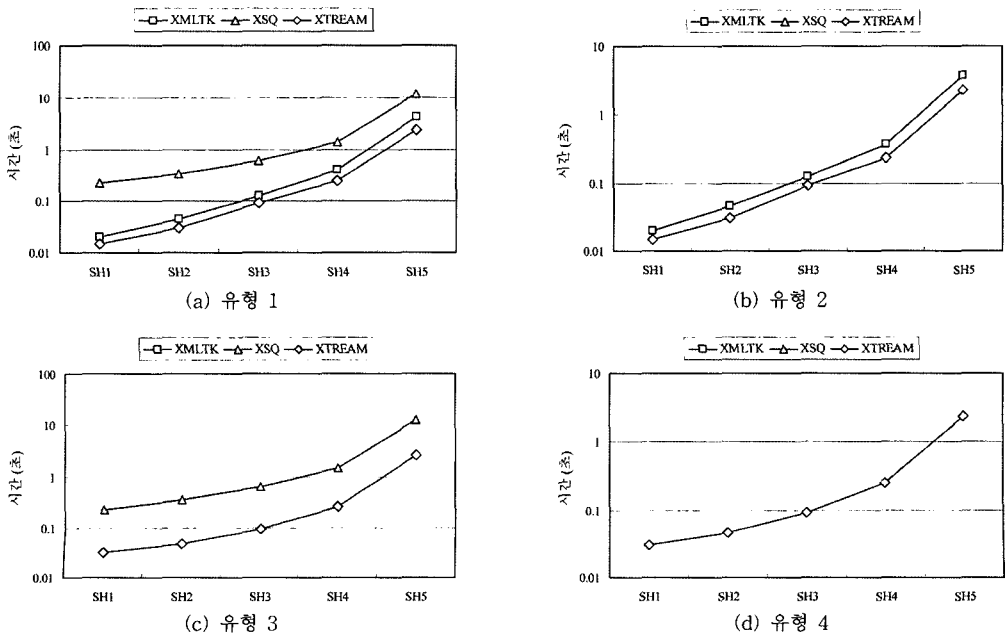
4.2 실험 결과

4.2.1 단일 질의 처리 성능

본 실험에서는 XTREAM의 성능을 비교하기 위해서 기존의 스트리밍 XML 데이타 처리기들을 사용한다. 하지만, 이러한 스트리밍 XML 데이타 처리기들은 한 번에 하나의 질의만 처리한다. 따라서, XTREAM의 성능은 한 번에 하나의 질의만을 처리하는 XNAIVE의 성능과 동일하다.

그림 8은 표 5에서 설명한 질의 유형에 따른 XMLTK, XSQ, 그리고 XTREAM의 단일 질의 처리기에 대한 질의 처리 시간을 로그 비율 기반의 그래프로 보여준다. 표 2에서 설명한 바와 같이, Shakespeare 데이타 세트의 다섯 XML 문서들을 스트리밍 XML 데이타들로 사용하여 단일 질의 처리 성능을 측정하였다. XMark 데이타 세트에 대한 질의 처리 성능은 Shakespeare 데이타 세트에 대한 질의 처리 성능과 유사하므로, 본 논문에서는 XMark 데이타 세트에 관한 결과를 포함하지 않는다.

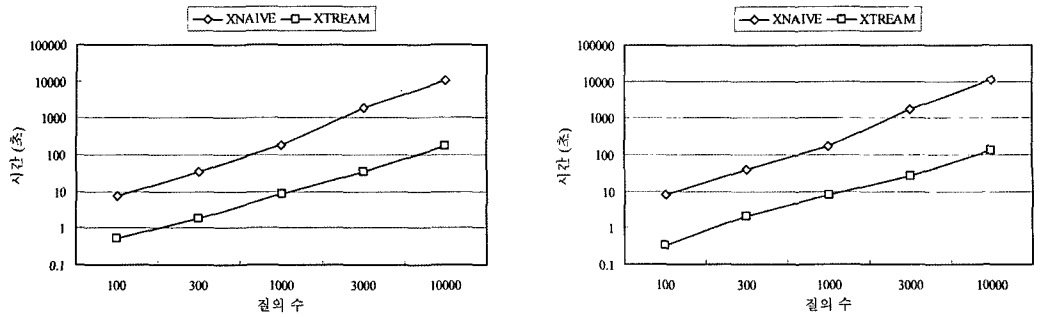
모든 질의 유형에 대해서 단일 질의 처리 기반의 XTREAM이 가장 좋은 질의 처리 성능을 보인다. 더욱이, XTREAM은 다양한 종류의 XPath 질의들을 지원한다. 그림 8(b)는 XSQ가 순서 기반 프리디케이트를 지원하지 않으므로 XMKTK와 XTREAM의 질의 처리



(a) 유형 1 (b) 유형 2

(c) 유형 3 (d) 유형 4

그림 8 단일 질의 처리 시간(로그 비율)



(a) XMark 데이타 세트 (b) Shakespeare 데이타 세트

그림 9 다중 질의 처리 시간(로그 비율)

시간만 포함한다. 이와 유사하게, 그림 8(c)는 XMLTK가 순서 기반 프리디케이트와 애트리뷰트만 지원하므로 XMLTK의 질의 처리 시간은 포함되지 않는다. 마지막으로, 그림 8(d)에서 보여주는 유형 4의 질의들은 XMLTK와 XSQ가 와일드카드를 지원하지 않으므로 XTREAM만 처리한다.

앞에서 언급한 바와 같이 XTREAM은 EStack에서 활성화되는 상태들의 정보를 유지하고, LStack에서 앞으로 발생할 이벤트들에 의해서 활성화될 수 있는 상태들의 정보를 유지한다. 따라서, SAX 파서에서 발생되는 이벤트에 대하여 활성화 될 상태들을 LStack의 상단에 있는 상태들 중에서 고르면 된다. 이에 반하여 XMLTK와 XSQ는 각 이벤트 발생 시 마다 오토마타를 접근하여 활성 가능 상태들을 탐색해야 하는 부담이 존재한다. 그 결과, XTREAM은 다양한 종류의 XPath 질의들을 지원하면서 가장 좋은 질의 처리 성능을 보인다.

평균적으로, 단일 질의 처리에 대한 XTREAM의 질의 처리 시간은 XMLTK의 질의 처리 시간보다 1.5배 빠르고, XSQ의 질의 처리 시간보다 7.5배 빠르다.

4.2.2 다중 질의 처리 성능

본 실험에서는 한 번에 단일 질의를 처리하는 질의 처리기(XNAIVE) 보다 한 번에 다중 질의를 동시에 처리하는 질의 처리기(XTREAM)의 효과적인 질의 처리 성능을 보인다. 그림 9는 다양한 수의 질의들에 대한 XNAIVE와 XTREAM의 질의 처리 성능을 로그 비율 기반의 그래프로 보여준다. 단일 질의 처리 성능에서 설명한 바와 같이, XNAIVE의 질의 처리 성능은 기존의 스트리밍 XML 데이타 처리기들의 성능보다 훨씬 빠르다. 따라서, 본 실험에서는 XNAIVE의 성능과 XTREAM의 성능을 비교한다.

본 실험에서 XML 문서의 크기가 대략 1MB 정도인 XM3와 SH2를 스트리밍 XML 데이타로 사용하였다. 예상한 바와 같이, 한 번에 다중 질의를 동시에 처리하는 XTREAM에 비해, 한 번에 단일 질의만 처리하여 여러 번의 문서 검색을 유발하는 XNAIVE가 모든 경우에 대해서 느린 질의 처리 성능을 보였다. 그 결과, XTREAM은 항상 더 빠른 질의 처리 성능을 보인다.

평균적으로, XTREAM의 질의 처리 시간은 XNAIVE의 질의 처리 시간보다 38.5배 빠르다.

5. 결 론

본 논문에서는 스트리밍 XML 데이타에 대해 다중 질의를 동시에 처리하는 다중 질의 처리기인 XTREAM을 제안한다. 기존의 여과 시스템들은 제한된 종류의 XPath 질의들을 지원한다. 이와 반대로, XTREAM은 스트리밍 데이타의 한 번 읽는 환경에서 트리 기반 표현식, 순서 기반 프리디케이트, 그리고 중첩된 프리디케이트와 같은 다양한 종류의 XPath 질의들을 처리한다.

XTREAM에서, 각 질의는 MXT(Minimal XPath Tree)로 표현하며, EStack(Evaluated Stack)과 LStack(Look-Forward Stack)이라 불리는 두 스택들을 사용한다. EStack은 질의 처리 과정에 대한 현재와 이전의 상태를 유지하고, LStack은 앞으로 방문 가능한 엘리먼트들에 대응하는 MXT 노드들의 집합을 유지한다. 추가적으로, LStack은 프리디케이트 조건의 만족 결과를 전달하는 경로로 사용한다.

XTREAM의 효율성과 확장성을 검증하기 위해, 실제 XML 데이타 세트와 합성 XML 데이타 세트를 사용하여 실험을 수행하였다. 실험 결과, 스트리밍 XML 데이타에 대해 다중 질의 처리를 지원하는 XTREAM이 기존의 스트리밍 XML 데이타 처리기들보다 빠른 질의 처리 성능을 보임을 검증할 수 있었다.

현재, XTREAM은 child 축과 descendant 축만 지원한다. 따라서, 향후 연구로 parent 축과 ancestor 축을 지원하도록 XTREAM을 확장할 예정이다.

참 고 문 헌

[1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and

E. Maler, "Extensible Markup Language(XML) 1.0, W3C Recommendation," http://www.w3.org/TR/REC-XML, 1998.

[2] M. Altinel and M. J. Franklin, "Efficient Filtering of XML Documents for Selective Dissemination of Information," Proc. of 26th International Conference on Very Large Data Bases, pp. 53-64, September 2000.

[3] S. Bose and L. Fegaras, "Data Stream Management for Historical XML Data," Proc. of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 239-250, June 2004.

[4] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi, "Efficient Filtering of XML Documents with XPath Expressions," Proc. of the 18th International Conference on Data Engineering, pages 235-244, February 2002.

[5] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer, "YFilter: Efficient and Scalable Filtering of XML Documents," Proc. of the 18th International Conference on Data Engineering, pp. 341-342, February 2002.

[6] A. K. Gupta and D. Suciu, "Stream Processing of XPath Queries with Predicates," Proc. of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 419-430, June 2003.

[7] B. He, Q. Luo, and B. Choi, "Cache-Conscious Automata for XML Filtering," Proc. of the 21st International Conference on Data Engineering, pp. 878-889, April 2005.

[8] F. Peng and S. S. Chawathe, "XPath Queries on Stream Data," Proc. of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 431-442, June 2003.

[9] J. Clark and S. DeRose, "XML Path Language (XPath) Version 1.0, W3C Recommendation," http://www.w3.org/TR/xpath, November 1999.

[10] B. Ludascher, P. Mukhopadhayn, and Y. Papakonstantinou, "A transducerbased xml query processor," Proc. of 28th International Conference on Very Large Data Bases, pp. 227-238, October 2002.

[11] F. Peng and S. S. Chawathe, "XSQ: A Streaming XPath Engine," Technical Report CS-TR4493, University of Maryland, 2003.

[12] J. E. Hopcraft and J. D. Ullman, "Introduction to Automata Theory, Language, and Computation," Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.

[13] I. Avila-Campillo, T. Green, A. Gupta, M. Onizuka, D. Raven, and D. Suciu, "Xmltk: An xml toolkit for scalable xml stream processing," Proc. of Programming Language Technologies for XML (PLAN-X), October 2002.

[14] F. Bry, F. Coskun, S. Durmaz, T. Furche, D. Olteanu, and M. Spannagel, "The XML Stream Query Processor SPEX," Proc. of the 21st International Conference on Data Engineering, pp. 1120-1121, April 2005.

[15] Y. Chen, S. B. Davidson, and Y. Zheng, "ViteX: a Streaming XPath Processing System," Proc. of the 21st International Conference on Data Engineering, pp. 1118-1119, April 2005.

[16] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava, "Navigational- vs. Index-Based XML Multi-Query Processing," Proc. of the 19th International Conference on Data Engineering, pp. 139-150, February 2003.

[17] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "XMark: A Benchmark for XML Data Management," Proc. of the 28th International Conference on Very Large Data Bases, pp. 974-985, August 2002.

[18] R. Cover, "The XML Cover Pages," http://www.oasis-open.org/cover/xml.html, 2001.

민 준 기

1995년 숭실대학교 전자계산학과(학사) 1997년 한국과학기술원 전산학과(석사) 2002년 한국과학기술원 전산학전공(박사). 2003년~2004년 한국과학기술원 Post-Doc 및 초빙교수. 2004년 한국전자통신연구원 선임연구원. 2005년~현재 한국기술교육대학교 조교수. 관심분야는 XML, 시공간DB, 스트림 데이타, 센서네트워크

박 명 제

정보과학회논문지 : 데이타베이스
제 34 권 제 2 호 참조

정 진 완

정보과학회논문지 : 데이타베이스
제 34 권 제 2 호 참조