NFA 표현을 사용한 문서-중심적 XML의 키워드 기반 필터링 기법

(A Keyword-based Filtering Technique of Document-centric XML using NFA Representation)

이 경 한 † 박 석 ††
(Kyounghan Lee) (Seog Park)

요 약 XPath 명세는 XML 원소 내용을 필터링하기 위한 질의어 작성이 어렵다. 본 논문은 이러한 문제점을 해결하기 위해 SQL의 LIKE 연산자에서 사용되던 특별한 매칭 문자 '%'를 허용한 확장된 XPath 명세와 그것을 표준 질의어로 사용하는 문서-중심적 XML 필터링 기법인 Pfilter를 제안한다. Pfilter는 값-기반 술어(value-based predicate)에서 피연산자의 공통 앞부분 문자를 공유하여 값-기반 술어의 처리 성능을 향상시킨다. 또한 본 논문은 Pfilter와 대표적인 데이타-중심적 XML 필터링 기법인 Yfilter를 값-기반 술어 처리의 확장성과 효율성에 대해 비교하고 Pfilter의 값-기반 술어 삽입, 삭제, 처리 결과를 제공한다. 본 논문에서 제안한 Pfilter는 XML 필터링 시스템에서 XPath의 contains() 함수를 평가(evaluation)하기 위한 핵심 알고리즘으로 사용할 수 있으며, XML 기반의 분산 정보 시스템을 구축하기 위한 기초 연구로 활용될 수 있다.

키워드 : XML, 필터링, 값-기반 술어

Abstract In this paper, we propose an *extended XPath specification* which includes a special matching character '%' used in the LIKE operation of SQL in order to solve the difficulty of writing some queries to filter element contents well, using the previous XPath specification. We also present a novel technique for filtering a collection of document-centric XMLs, called Pfilter, which is able to exploit the extended XPath specification. Owing to sharing the common prefix characters of the operands in value-based predicates, the Pfilter improves the performance in processing those. We show several performance studies, comparing Pfilter with Yfilter in respect to efficiency and scalability as using *multi-query processing time* (MQPT), and reporting the results with respect to inserting, deleting, and processing of value-based predicates. In conclusion, our approach provides a core algorithm for evaluating the *contains*() function of XPath queries in previous XML filtering researches, and a foundation for building XML-based distributed information systems.

Key words : XML, filtering, value-based predicate

1. 서 론

XML(Extensible Markup Language)[1]은 원소 내용(element content)에 여러 가지 방법으로 원소(element)를 표기할 수 있다. 표기 형태에 따라 크게 두 가지로 분류할 수 있는데 그것은 데이타-중심적(data-centric) XML과 문서-중심적(document-centric) XML이다. 데이타-중심적 XML이란, XML 태크(tag)로 표기된 매우 구조적인 데이타이다. 그러나 문서-중심적 XML은 느슨하게 구조화된 데이타이다[2]. 이러한 문서-중심적 XML의 예로 RSS XML 파일을 들 수 있다. CNN.com에서 배포하는 top_stories.xml[1] RSS 파일은 평균 52개의 원소를 가지고 있으며 최대 깊이는 4이다. 또한 평균 원소 내용(element content)의 길이는 약 56개의 문자로 XML 문서의 구조는 매우 간단하지만 한 쌍의 XML 태그에는 비교적 긴 원소 내용을 가지고 있

· 본 연구는 한국과학재단 특정기초연구(R01-2006-000-10609-0) 지원으로 수행되었음
† 정 회 원 : 삼성전자 무선사업부
kyounghan.lee@samsung.com
†† 종신회원 : 서강대학교 컴퓨터학과 교수
spark@sogang.ac.kr
논문접수 : 2006년 4월 5일
심사완료 : 2006년 6월 19일

1) http://rss.cnn.com/rss/edition.rss

다. 현재까지 표준 XML 질의어로 알려진 XPath[3], XQuery[4]는 데이타-중심적 XML에 질의하기는 상당히 효과적이지만 문서-중심적 XML에 질의하기는 불편한 점이 많다. 그 이유는 위와 같은 질의어들은 원소 내용의 정보 검색(information retrieval)을 위한 특별한 매칭 문자를 지원하지 않기 때문이다.

특히 XML 질의 처리와 XML 필터링 기법은 상당히 많은 연구 성과가 이루어진 주제이다. 지금까지 연구된 XML 필터링 기법은 RSS 배포와 같은 문서-중심적 XML에 큰 변경 없이 이용할 수 있지만, 이러한 기법들은 데이타-중심적 XML의 필터링에 초점을 맞췄기 때문에 다음과 같은 요구사항을 갖는다.

첫째, XPath는 값-기반 술어를 수행하기 위해 text() 함수를 사용한다. 그러나 text() 함수는 술어의 피연산자와 XML의 원소 내용 사이의 문자열 동등 비교 연산만 지원하기 때문에 원소 내용이 비교적 짧은 데이타-중심적 XML에서는 값-기반 술어를 작성하기에 충분하나 문서-중심적 XML에서는 값-기반 술어를 만들어 내기 까다롭다. 이러한 문제를 해결하기 위해 SQL문의 LIKE 연산자에서 사용되는 '%'와 같은 특별한 매칭 문자를 XPath 명세에 추가해야 한다.

둘째, 문서-중심적 XML 필터링 시스템에서 가장 중요한 요구사항은 확장성이다. 특히 문서-중심적 XML 필터링 시스템은 XML 데이타가 필터링 시스템에 도착하는 초당 비율과 그 XML 데이타의 크기 그리고 질의 개수의 차원에서 확장성이 있어야 한다. 부언하자면 응용(application)에 따라 XML 데이타 도착 비율은 특정 시간에 따라 상당히 바뀔 수 있고 XML 데이타의 크기 또한 1KB(예, 주식 시세 변화)에서 20KB(예, 뉴스 기사)로 다양하게 변할 수 있다. 그리고 개인화된 뉴스 기사 생성과 주식 시세 검색을 제공하는 단말기 응용에서 질의 개수는 수 백만 개를 넘을 수 있다. 따라서 빠르게 필터링 시스템에 들어오는 변화무쌍한 크기의 XML 데이타에 매칭되는 질의 원소를 매우 큰 질의 집합 속에서 효과적으로 찾아 내는 것이 문서-중심적 XML을 필터링하기 위한 주요한 해결 과제라고 할 수 있다.

셋째, 문서-중심적 XML 필터링 시스템은 동적 환경에서 견고성을 갖추어야 한다. 동적 환경이란, 필터링 시스템을 사용하는 구독자들이 XML 데이타의 관심도에 따라 쉽게 질의를 등록하거나 해지할 수 있음을 의미한다. 따라서 필터링 시스템 내에서는 질의의 삽입/삭제가 빈번하게 이루어지므로, 시스템에 등록된 질의 집합의 크기에 상관없이 XML 데이타의 처리에 최소한의 영향을 미치면서 재빠르게 질의 집합을 연속적으로 갱신할 수 있어야 한다.

넷째, 인터넷-기반 환경에서 문서-중심적 XML 필터링 시스템이 이질적인(heterogeneous) XML 메시지를 받아 질의 수행해야 함은 당연하다. 이러한 특징 때문에 문서-중심적 XML 필터링 시스템은 XML 스키마에서 얻을 수 있는 메타(meta) 정보를 이용한 질의 수행 기법으로 구현될 수 없다. 결국, 문서-중심적 XML 필터링은 전통적인 XML 질의 처리와 다른 요구사항을 가지고 있기 때문에 새로운 질의 처리 알고리즘이 필요하다.

본 논문은 기존의 XML 필터링 기법을 고찰하고 값-기반 술어 수행을 정의한다. 그 후 Aho-Corasick 사전 매칭 알고리즘[5]을 변형하여 값-기반 술어 수행 성능을 높인 새로운 기법을 소개한다. 마지막으로 제안된 기법과 Yfilter[6], 그리고 Yfilter의 값-기반 술어 수행 부분을 원래의 Aho-Corasick 사전 매칭 알고리즘으로 대체한 실험 대조군 기법과의 유지 비용, 확장성과 효율성을 비교한 후 본 연구의 결론을 맺는다.

2. 관련연구

전통적인 RDBMS는 선택 연산을 질의 계획에서 가장 먼저 수행한다. 이러한 질의 계획의 변경은 중간 결과물을 줄여 다음 연산의 입력을 최소화시킴으로써 질의 처리의 효율성을 높이기 위함이다. 그러나 이러한 경험적 방법(heuristic)을 Yfilter의 inline 접근법처럼 XML 필터링 시스템에 적용하면 기대했던 바와 다르게 좋지 못한 성능을 나타낸다. 이러한 문제점을 해결하기 위해 Yanlei Diao는 SP(Selection Postponed) 접근법을 제안했다[6]. SP 접근법에 대해 간단히 설명하면 구조 매칭 동안 비결정적 유한 오토마타(nondeterministic finite automata, NFA)의 승인 상태에 도착할 때까지 값-기반 술어 처리를 지연시킨 후 승인 상태에 도달한 질의에 대해서만 값-기반 술어 처리를 일률적, 순차적으로 진행하는 방법이다. 따라서 SP 접근법은 잠재적으로 세 가지 장점을 갖는다. 첫째, 한 개의 질의에서 서로 다른 원소에 나타나는 술어들에 대해 논리곱(conjunction)을 할 수 있기 때문에 술어의 단축 평가(short-cut evaluation)가 가능하다. 둘째, inline 접근법에서 XML의 중첩 구조를 처리하기 위해 요구되는 NFA 상의 퇴각검색(backtracking)이 불필요하다. 셋째, 질의 평가를 위한 기장정보(bookkeeping)가 필요 없기 때문에 이를 유지하기 위한 메모리 비용이 없어진다.

현재까지 Yfilter 이외의 많은 XML 필터링 기법이 제안되었다. 그러나 대부분의 연구는 상위 레벨의 원소가 많이 중첩되는 XML의 특징 때문에 구조 매칭에 중점을 두었다[7-11]. 이러한 연구는 데이타-중심적 XML에 적용하기에는 매우 효과적이나 주위에서 흔히 접할 수 있는 문서-중심적 XML을 필터링하기에는 부족한 점이 있다. 그렇다고 XML 필터링 시스템에 관한 연구

에서 값-기반 술어를 효과적으로 처리하기 위해 아무런 노력이 없었던 것은 아니다.

XPush[12]와 RDBMS를 이용한 XML 필터링 시스템[13]은 원자적 술어(atomic predicate)를 정의하고 이것을 공유한다. 원자적 술어란, 값-기반 술어에서 논리곱의 요소를 이루는 술어를 말한다. 예를 들면 x_i는 상수 값이고 p_i는 동등 연산자가 아닌 연산자를 가지고 있는 술어들의 논리곱일 때 질의 "$/a/text() > x_i$ AND p_i"에서 $text() > x_i$는 원자적 술어이다. 원자적 술어를 공유하는 방법을 위의 예제 질의를 이용하여 설명하면 다음과 같다. 필터링 시스템은 값-기반 술어를 처리하기 전에 x_i가 가질 수 있는 값의 범위를 n개의 버킷(bucket)으로 분할한다. 따라서 값의 범위는 $[b_0,b_1),[b_1,b_2),\ldots,[b_{n-1},b_n)$로 분할된다. 그 후 값-기반 술어를 처리할 때 x_i가 $[b_k,b_{k+1})$ 버킷에 속한다면 예제 질의를 "$/a/text() > b_k$ AND p_i AND $/a/text() > x_i$"로 재작성(rewriting)한다. 이런 식으로 원자적 술어를 공유하면 술어의 논리곱을 단축 평가할 수 있기 때문에 값-기반 술어 처리의 효율을 높일 수 있다.

숫자값과 문자열을 모두 값-기반 술어의 피연산자로 사용할 수 있는 Yfilter는 원자적 술어를 공유하여 단축 평가를 가능케 하는 방법을 일반화시켰다고 볼 수 있다. 그러나 이러한 방법은 시스템이 얼마나 많은 버킷을 가져야 하는지에 대한 특별한 기준을 정하기 어렵고 문서-중심적 XML에 질의하기 위해, 반드시 필요한 특별한 매칭 문자 '%'를 효과적으로 지원하기 위해서는 또 다른 알고리즘이 필요하다. 또한 원자적 술어를 공유하는 기법은 숫자 값에 대해서만 단축 평가를 이용할 수 있다는 단점이 존재한다.

값-기반 술어를 효과적으로 처리하기 위해 XSQ[14]는 푸시다운 변환기(pushdown transducer)를 이용하여 원자적 술어를 공유한다. 이 기법은 숫자와 문자 모두에 대해 공유가 가능하며 현재까지 제안된 기법 중 값-기반 술어 처리에 가장 효과적이다. 그러나 XSQ가 생성하는 상태의 개수는 NFA를 이용한 Yfilter 보다 $O(2^n)$배로 증가(단, n은 모든 질의의 테스트 노드에서 '*'가 나타나는 횟수)하므로 많은 메모리를 소모하는 문제점이 발생한다. 따라서 많은 질의를 등록해야 하는 XML 필터링 시스템에 적용되기 어렵다. 또한 XSQ는 공통 앞부분 원소를 공유한 구조 매칭을 이용하지 못하는 단점이 존재한다.

3. Pfilter의 값-기반 술어 수행

3.1 개요

문자열을 문자의 유한 순서라 하자. 원소 내용의 스트림 x와 피연산자 y_i $(1 \le i \le k)$는 임의의 문자열이라고 하자(단, k는 필터링 시스템에 등록된 모든 질의에서 유일한 값-기반 술어의 개수). 또한 피연산자 y_i를 인덱스하고 있는 (질의번호, 경로번호, 레벨) 쌍의 집합을 $pair_i$라 하자. 그럼 피연산자 y_i와 (질의번호, 경로번호, 레벨) 쌍의 집합 $pair_i$를 원소로 하는 쌍의 집합 $K = \{(y_i, pair_i) \mid i = 1,2,\ldots,k\}$를 값-기반 술어의 키워드 집합으로 정의할 수 있다. 이러한 설정에서 값-기반 술어 수행의 형식적 정의는 다음과 같다.

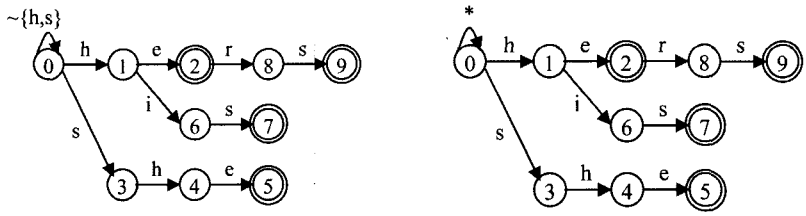
정의 1(값-기반 술어 수행). 원소 내용의 스트림 x를 특별한 매칭 문자 '%'를 고려하여 매칭하는 피연산자 y_j (단, j는 $1 \le j \le k$인 임의의 자연수)가 존재할 때, 그 피연산자 y_j를 포함한 쌍을 원소로 갖는 쌍들의 집합 $M = \{(y_j, pair_j) \mid 1 \le j \le k, j$는 임의의 자연수 $\} \subseteq K$를 찾는다.

3.2 공유된 값-기반 술어 매칭

3.2.1 값-기반 술어의 표현

효율적인 값-기반 술어의 수행을 위해 술어에서 피연산자의 공통 앞부분 문자를 식별하고 그들 사이의 처리를 공유한다. Yfilter에서 술어의 피연산자를 각 술어에 대해 한 개씩 순차적으로 처리했다면 제안하는 기법은 모든 값-기반 술어의 피연산자를 이용하여 비결정적 유한 오토마타(nondeterministic finite automata, NFA) 형태를 갖는 한 개의 유한 상태 기계(finite state machine, FSM)로 결합한다. 이러한 값-기반 술어 NFA는 다음 두 가지 특징을 갖는다. 첫째, 각 술어의 피연산자에 대해 단 한 개의 승인 상태가 존재한다. 둘째, 피연산자의 공통 앞부분 문자는 한번만 나타난다.

일반적으로 문자열을 매칭하기 위한 오토마타는 NFA와 결정적 유한 오토마타(deterministic finite automata, DFA) 형태로 모두 구성될 수 있다. 이 두 가지 형태 중 한가지를 선택하는 것은 일종의 상충관계(trade-off)이다. 즉, NFA 형태를 선택할 경우 오토마타의 빠른 구성이 가능하지만 DFA에 비해 문자열 매칭 시간을 증가하게 된다(그 역도 마찬가지). 그럼 이러한 상충관계를 고려하여 본 논문에서 다루고자 하는 문서-중심적 XML 필터링 환경에 적합한 오토마타를 선택하자. 서두에서도 언급했듯이 기본적으로 XML 필터링 시스템에 등록되는 질의는 일반적인 RDBMS와 달리 상당히 많다. 또한 등록된 질의는 빈번하게 삭제되고 또 다른 질의가 등록될 수 있다. 이러한 XML 필터링 시스템의 특



(a) DFA 형태 (b) NFA 형태

그림 1 키워드 집합 {he, she, his, hers}에 대한 두 오토마타 형태

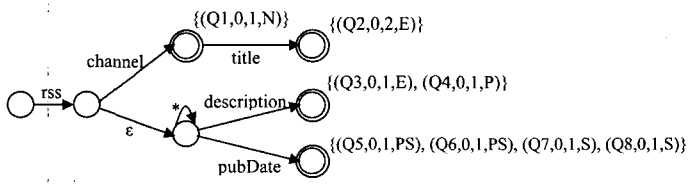
징 때문에 본 논문에서는 시스템의 처리량뿐만 아니라 질의의 삽입, 삭제 시 질의 인덱스의 갱신 비용도 고려해야 한다.

그림 1은 키워드 집합 {he, she, his, hers}에 대한 DFA와 NFA 표현을 보여준다. 본 그림에서 키워드 집합의 DFA 표현은 Aho-Corasick 사전 매칭 트리와 동일하다는 점에 주의하라. (a), (b)에서 각 상태번호에서 매칭 실패 시 상태번호 0으로의 천이는 생략했다. 게다가 (a)는 실패 함수 f의 천이도 생략했다. 이 두 오토마타의 상태 개수는 같고 각 상태의 천이만 약간 다르다. 즉, 사전 매칭을 위한 오토마타는 DFA와 NFA의 차이가 거의 없기 때문에 질의의 삽입, 삭제 시 질의 인덱스 갱신 비용을 고려했을 때, 키워드 집합이 바뀔 때마다 실패 함수를 재계산해야 되는 DFA 형태 보다 실패 함수를 사용할 필요가 없는 NFA가 이점을 얻는다(4절의 [실험 2] 참조). 또한 두 오토마타 형태를 각각 값-기반 술어에 적용했을 때 평가 비용은 두 오토마타 형태의 천이 차이가 거의 없기 때문에 DFA 형태로 얻을 수 있는 이점이 상대적으로 적다(4절의 [실험 3] 참조).

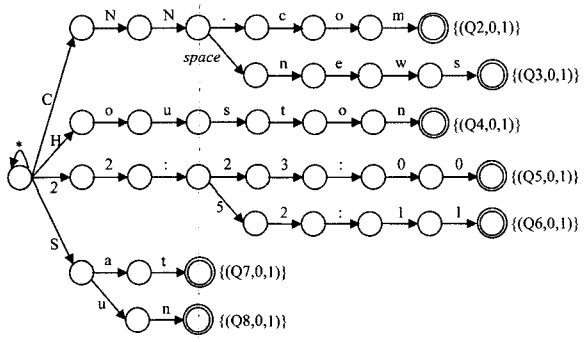
그림 2는 8개의 질의를 표현하고 있는 구조 NFA와 값-기반 술어 NFA의 예를 보여주고 있다. 구조 NFA는 Yfilter의 NFA 구성 방법을 그대로 따랐으므로 설

Q1 = /rss/channel
Q2 = /rss/channel/title[text()="CNN.com"]
Q3 = /rss//description[text()="CNN news"]
Q4 = /rss//description[text()="%Houston"]
Q5 = /rss//pubDate[text()="%22:23:00%"]
Q6 = /rss//pubDate[text()="%22:52:11%"]
Q7 = /rss//pubDate[text()="Sat%"]
Q8 = /rss/pubDate[text()="Sun%"]

(a) Xpath 질의



(b) (a)에 대응하는 구조 NFA



(c) (a)에 대응하는 값-기반 술어 NFA

그림 2 XPath 질의의 NFA-기반 표현

명을 생략한다. 한가지 주의할 점은 Yfilter의 구조 NFA와 달리 각 승인 상태는 (질의번호, 경로번호, 레벨, 매칭 문자 정보) 쌍으로 구성된 부가 정보를 갖는다. 매칭 문자 정보에서 "N"은 술어가 없음, "E"는 술어가 존재하나 '%'가 나타나지 않음, "P"는 술어가 존재하고 접두 '%'가 나타남, "S"는 술어가 존재하고 접미 '%'가 나타남, "PS"는 술어가 존재하고 접두-접미 '%'가 나타남을 의미한다. 그리고 방향성 간선은 천이를 나타내며 값-기반 술어 NFA의 간선 위에 나타나는 문자는 천이를 유발하는 입력을 의미한다. 마지막으로 두 NFA에서 음영 처리된 원은 경로 질의의 각 위치 단계 또는 값-기반 술어에서 피연산자의 문자가 공유됨을 의미한다. 이러한 NFA 형태의 표현에서 주의할 점은 두 NFA 모두 공통 앞부분을 공유하고 있고 다수의 승인 상태를 가지고 있으며 특별한 매칭 문자 '%'는 값-기반 술어 NFA에 표현되지 않는다는 것이다.

각 값-기반 술어의 피연산자는 값-기반 술어 NFA에서 단지 한 개의 승인 상태만 갖는다. 특별한 매칭 문자 '%'를 제거했을 때, 동일한 피연산자를 갖는 술어는 승인 상태를 공유할 수 있다. 이러한 두 NFA는 형식적으로 무어(Moore) 기계로 정의되며 승인 상태 집합에서 (질의번호, 경로번호, 레벨) 쌍 또는 (질의번호, 경로번호, 레벨, 매칭 문자 정보) 쌍들로 사상하는 함수를 무어 기계의 출력 함수로 볼 수 있다.

3.2.2 결합된 NFA 생성

그림 2(c)의 값-기반 술어 NFA는 8개의 질의에 대해 지금부터 설명할 생성 방법을 연속하여 적용한 결과이다. 사실 값-기반 술어 NFA의 구성은 '*'와 '//'을 고려한 구조 NFA 보다 더 간단하다. 그림 3에서 보여주는 값-기반 술어 NFA 조각이라고 부르는 방향성 그래프는 그림 2(a)에 있는 8개의 질의에 대응된다. 단, 질의 Q1은 경로 표현식만 나타나기 때문에 그림 3에는 7개의 값-기반 술어 NFA 조각이 나타나게 된다.

각각의 값-기반 술어 NFA 조각을 NFA_v로 표기하자. NFA_v는 한 개의 값-기반 술어 NFA로 결합할 수 있는데 그 방법은 간단하다. 그림 2(c)에서와 같이 모든 들에 의해 공유되는 초기 상태가 반드시 한 개 존재한다. 새로운 NFA_v를 삽입하기 위해서 결합된 값-기반 술어 NFA는 (1) NFA_v의 승인 상태에 도달되거나 (2) NFA_v에 대응하는 천이가 없을 때까지 반복해서 탐색한다. 첫 번째 경우 마지막 상태가 승인 상태가 되고 (질의번호, 경로번호, 레벨) 쌍을 승인 상태에 연관시킨다. 두 번째 경우 결합된 값-기반 술어 NFA의 마지막으로 도착하는 상태에서 새로운 가지(branch)를 생성한다. 이 가지는 NFA_v의 매칭되지 않는 천이를 가지고 구성한다. 이러한 값-기반 술어 NFA의 삽입/삭제 알고리즘은 그림 4에 기술되어 있으며 이 알고리즘은 이용한 첫 번째 결합 예를 그림 5가 나타내고 있다.

값-기반 술어 NFA의 초기 상태에서 모든 문자에 대해 천이를 유발시키는 '*'는 이 모델을 비결정적으로 만든다. 그림 2(c)의 NFA 모델에서 만약 자기-루프(self-loop)를 가진 상태가 그 다음 상태로 천이되면 다음 입력 문자를 받을 때 자기-루프를 가진 상태와 현재 상태에 대해 각각 천이한다. 이는 다음 문자를 입력할 때 처리할 상태가 1개에서 2개로 늘어남을 의미한다.

지금까지 살펴본 값-기반 술어 NFA의 생성에서 중요한 점은 이러한 과정이 점진적이라는 것이다. 따라서 새로운 피연산자는 쉽게 값-기반 술어 NFA에 추가될

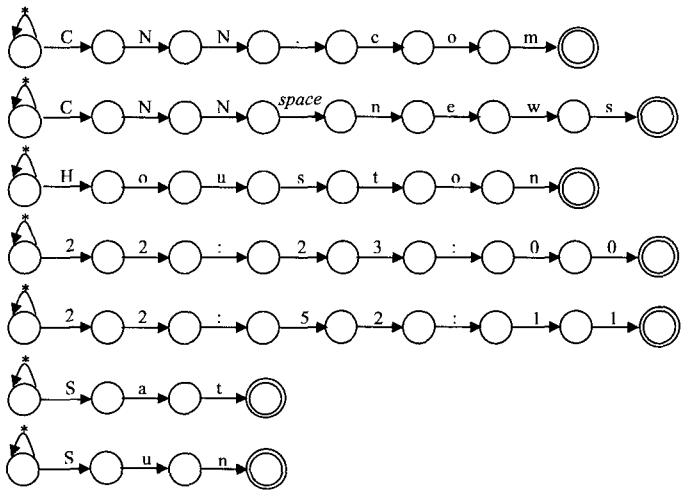


그림 3 값-기반 술어 NFA 조각

```
Input: inserting value-based predicate NFA fragments, NFA_fragments
Output: value-based predicate NFA, NFA_value
procedure INSERT(NFA_fragments)
 1  for each NFA_v ∈ NFA_fragments do
 2      for each state s ∈ NFA_v do
 3          if s = "accept state" then
 4              associate s with (query id, path id, level) pair w.r.t. NFA_v;
 5          end if
 6          if s != "shared with NFA_value" then
 7              NFA_value is branched off using remainder states of NFA_v;
 8              break;
 9          end if
10      end for
11  end for
12  return NFA_value;
```

(a) 삽입 알고리즘

```
Input: deleting value-based predicate NFA fragments, NFA_fragments
Output: value-based predicate NFA, NFA_value
procedure DELETE(NFA_fragments)
 1  for each NFA_v ∈ NFA_fragments do
 2      for each state s ∈ NFA_v do
 3          if s != "shared with NFA_value" then
 4              delete state s form NFA_value;
 5              if s = "accept state" then
 6                  delete (query id, path id, level) pair w.r.t. NFA_v from NFA_value;
 7              end if
 8          end if
 9      end for
10  end for
11  return NFA_value;
```

(b) 삽입 알고리즘

그림 4 값-기반 술어 NFA에 대한 피연산자 삽입/삭제 알고리즘

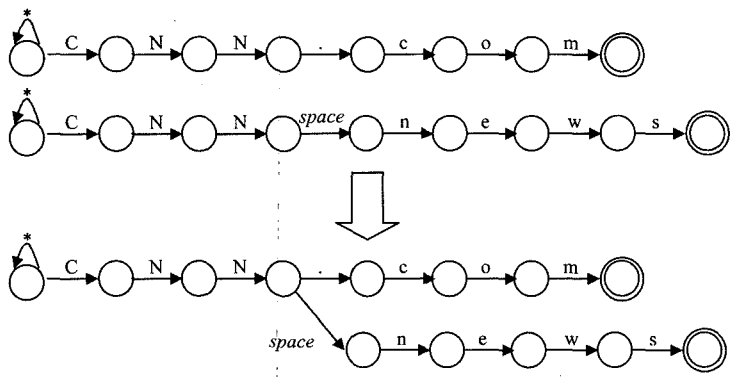


그림 5 NFA 조각의 첫 번째 병합

수 있으므로 값-기반 술어 NFA의 유지가 용이하다는 것이 이 접근법의 장점이다.

3.2.3 NFA 구조의 구현

사실 오토마타는 다양한 자료구조를 이용하여 구현이 가능하다. 그러므로 본 연구에서는 효과적인 값-기반 술어 실행을 위해 값-기반 술어 NFA를 해쉬 테이블로 구현한다. 그 이유는 해쉬 테이블 기반의 오토마타는 NFA 상태의 삽입/삭제 시간을 낮출 수 있기 때문이다. 이러한 접근법을 위해 각 상태는 다음과 같은 자료구조를 갖는다. (1) 상태번호를 저장할 수 있는 변수 (2) 올바른 천이를 저장할 수 있는 작은 해쉬 테이블 (천이 해쉬 테이블) (3) 만약 승인 상태이면 (질의번호, 경로

번호, 레벨) 쌍을 원소로 하는 연결 리스트를 추가적으로 갖는다. 각 상태에 대한 천이 해쉬 테이블은 (입력문자, 다음 상태번호) 쌍을 갖는다. 여기서 입력문자는 해쉬 테이블의 키(Key)이며 천이와 사상된다. 또한 다음 상태번호는 현재 상태번호가 천이될 때 도착하는 상태번호를 나타낸다. 자기-루프 천이를 가지고 있는 초기 상태를 천이 해쉬 테이블에 표현하기 위해 별도의 자료구조는 필요 없다. 그림 6은 그림 2(c)를 해쉬 테이블 기반으로 구현한 모습을 보여주고 있는데 각 해쉬 테이블에 할당된 숫자는 상태번호를 나타내며 굵은 사각형은 승인 상태를 의미한다. 각 승인 상태는 (질의번호, 경로번호, 레벨) 쌍을 가지고 있다.

지금까지 설명한 값-기반 술어의 표현, 오토마타의 생성 방법과 그 구현을 통해 Pfilter와 ACfilter를 정의할 수 있다.

정의 2 (Pfilter) 값-기반 술어의 효과적인 필터링을 위해 Yfilter의 값-기반 술어 처리 부분을 제거한 후 술어 집합을 그림 4의 삽입/삭제 알고리즘을 이용하여 해쉬-기반의 한 개의 NFA 형태로 구현한 문서-중심적 XML 필터링 기법을 Pfilter라고 정의한다.

정의 3 (ACfilter) 특별한 매칭 문자 '%'를 허용하지 못하는 Yfilter와 Pfilter의 합리적인 비교를 위해 Yfilter의 값-기반 술어 처리 부분을 Aho-Corasick 사전 매칭 트리로 대체한 후 연결 리스트-기반으로 구현한 문서-중심적 XML 필터링 기법을 ACfilter라고 정의한다.

ACfilter는 특별한 매칭 문자 '%'를 고려한 값-기반 술어 수행을 위해서 Aho-Corasick 사전 매칭 트리(그림 1(a)를 그대로 이용한다. 그러나 이러한 접근법은 술어의 삽입/삭제 시 매번 실패함수를 재계산하기 때문에 XML 필터링 시스템에 적합하지 못하며 술어를 표현한 각 상태와 천이를 갖는 DFA를 연결 리스트를 사용하여 구현했기 때문에 좋지 못한 성능을 나타낸다(4절의 [실험 2, 3] 참조). 이에 반해 Pfillter는 Aho-Corasick 사전 매칭 트리를 DFA에서 NFA로 바꿔줌으로써 실패함수를 사용할 필요가 없다(그림 1(b)). 그리고 XML 필터링 시스템은 2차 보조기억장치의 도움을 받지 않고 메모리 상에서만 작동하는 프로그램임에 착안하여 Pfilter의 값-기반 술어 NFA를, 일반적으로 메모리상에서 빠른 인덱스 기법으로 알려진 해쉬 테이블을 이용하여 구현한다. 따라서 해쉬 테이블 기반의 NFA는 피연산자의 문자들을 삽입/삭제하기에 용이한 이점을 갖는다.

3.2.4 NFA의 실행

마지막으로 해쉬 테이블을 이용하여 구현된 값-기반 술어 NFA 기계의 실행에 대하여 알아보자. 값-기반 술어 NFA는 하나의 문자를 이벤트로 간주하여 이벤트-구동적인 방법으로 술어를 수행한다. 필터링할 XML 문서가 Pfilter에 도착했을 때 원소 내용은 SAX 파서[15]의 Characters() 메소드에 의해 얻을 수 있다. 이 원소 내용을 첫 번째 문자에서부터 마지막 문자까지 하나씩 읽어가면서 각 문자에 대한 이벤트를 발생시킨다. 발생된 이벤트는 핸들러로 보내지고 값-기반 술어 NFA에서 천이를 유발한다. 참고로 ACfilter는 본 절에서 언급한 NFA 실행 알고리즘을 거의 변형 없이 적용할 수 있다. 따라서 ACfilter의 값-기반 술어 DFA(즉, Aho-Corasick 사전 매칭 트리)를 이용한 실행 알고리즘은 기술하지 않는다.

값-기반 술어 NFA의 운용은 구조 매칭과 달리 전역 스택(global stack)을 사용하지 않는다. 다만 값-기반 술어 NFA에서 여러 상태들이 동시에 활성화되기 때문에 활성 상태번호를 저장하기 위한 목적 상태 연결 리스트와 특별한 매칭 문자 '%'을 처리하기 위해 각 상태번호는 자기-루프 천이 횟수를 저장할 임시 공간이 필요하다. 실행 알고리즘은 각 핸들러와 함께 아래에 자세

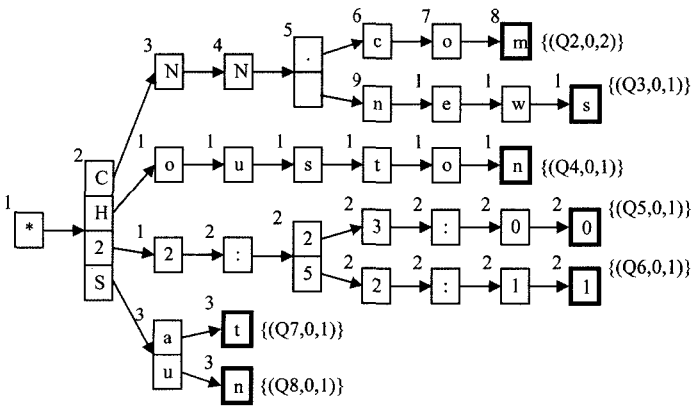


그림 6 값-기반 술어 NFA의 해쉬-기반 구현

히 설명되어 있다.

• 원소 내용 시작 핸들러

원소 내용이 시작될 때 값-기반 술어 NFA의 초기 상태에서 천이를 시작할 준비한다. 이 때 자기-루프 천이 횟수가 0인 초기 상태는 목적 상태 연결 리스트에 저장된다.

• 문자 핸들러

원소 내용으로부터 하나씩 문자를 읽을 때 마다 이 핸들러가 동작한다. 값-기반 술어 NFA 실행은 현재 활성 중인 각 상태에 대해 매칭이 되면 천이를 한다. 보다 자세한 설명을 위해 지금까지 자기-루프를 천이한 횟수를 SL, 피연산자에서 입력되지 않고 남아있는 문자의 개수를 RC라 하자. 그렇다면 각 활성 상태에서 다음 세 가지 절차를 수행한다.

(1) 값-기반 술어 NFA에 입력되는 문자 키를 이용하여 현재 상태 해쉬 테이블을 탐색한다. 만약 그 문자 키에 대응하는 다음 상태번호가 존재하면 그 상태번호를 목적 상태 연결 리스트에 삽입하고 RC를 감소시킨다.

(2) 만약 문자 키에 대응하는 다음 상태번호가 해쉬 테이블에 존재하지 않으면 목적 상태 연결 리스트에서 현재 상태번호를 제거한다. 그러나 현재 상태번호가 초기 상태이면 목적 상태 연결 리스트에서 제거하지 않고 그대로 두고 SL을 증가시킨다.

(3) 천이 후 상태가 승인 상태이면 SL과 RC를 참고하여 다음 4개의 해쉬 집합(hash set) 중 알맞은 곳에 (질의번호, 경로번호, 레벨) 쌍을 삽입한다.

- 만약 $SL = 0 \wedge RC = 0$이면 동등 해쉬 집합(equal hash set, EHS)에 (질의번호, 경로번호, 레벨) 쌍을 삽입한다.
- 만약 $SL > 0 \wedge RC = 0$이면 순수 접두 해쉬 집합(pure prefix hash set, PPHS)에 (질의번호, 경로번호, 레벨) 쌍을 삽입한다.
- 만약 $SL = 0 \wedge RC > 0$이면 순수 접미 해쉬 집합(pure suffix hash set, PSHS)에 (질의번호, 경로번호, 레벨) 쌍을 삽입한다.
- 만약 $SL > 0 \wedge RC > 0$이면 순수 접두-접미 해쉬 집합(pure prefix-suffix hash set, PPSHS)에 (질의번호, 경로번호, 레벨) 쌍을 삽입한다.

• 원소 내용 끝 핸들러

원소 내용의 각 문자가 값-기반 술어 NFA에 모두 입력되면 4개의 해쉬 집합에는 현재 원소 내용에 대해 특별한 매칭 문자 '%'까지 고려하여 매칭하는 (질의번호, 경로번호, 레벨) 쌍을 가지고 있게 된다. 구조 매칭을 만족한 (질의번호, 경로번호, 레벨, 매칭 문자 정보) 쌍은 구조 NFA로부터 알 수 있으므로 다음 절차를 이용하여 값-기반 술어 처리를 완료한다.

(1) 접두, 접미, 접두-접미 '%'가 없는 피연산자(매칭 문자 정보가 "N"인 경우)는 동등 해쉬 집합에서만 구조 매칭을 만족한 (질의번호, 경로번호, 레벨) 쌍이 있는지 확인하고 만약 있다면 그 쌍은 값-기반 술어를 만족한다.

(2) 접두 '%'가 있는 피연산자(매칭 문자 정보가 "P"인 경우)는 동등 해쉬 집합과 순수 접두 해쉬 집합에서 구조 매칭을 만족한 (질의번호, 경로번호, 레벨) 쌍이 있는지 확인하고 만약 있다면 그 쌍은 값-기반 술어를 만족한다.

(3) 접미 '%'가 있는 피연산자(매칭 문자 정보가 "S"인 경우)는 동등 해쉬 집합과 순수 접미 해쉬 집합에서 구조 매칭을 만족한 (질의번호, 경로번호, 레벨) 쌍이 있는지 확인하고 만약 있다면 그 쌍은 값-기반 술어를 만족한다.

(4) 접두-접미 '%'가 있는 피연산자(매칭 문자 정보가 "PS"인 경우)는 모든 해쉬 집합에서 구조 매칭을 만족한 (질의번호, 경로번호, 레벨) 쌍이 있는지 확인하고 만약 있다면 그 쌍은 값-기반 술어를 만족한다.

위와 같은 절차로 값-기반 술어 매칭을 확인한 후 나머지 구조 매칭(예, 중첩 경로 매칭)을 만족하게 되면 그 질의는 현재 XML 문서에 매칭된다. 지금까지 설명한 값-기반 술어 NFA의 실행 알고리즘은 그림 7에 적혀있다.

그림 8은 값-기반 술어 NFA의 실행 예를 보여준다. 그림 8(b)의 각 사각형은 문자 입력에 대한 (활성 상태 번호, 자기-루프를 천이한 횟수) 쌍을 저장하고 있다. 값-기반 술어 NFA의 실행은 Aho-Corasick 사전 매칭 트리를 실행하기 위해 필요한 실패 함수가 사용되지 않고 있음을 본 그림을 통해 확인할 수 있다. 이러한 알고리즘의 개선은 XML 필터링 시스템에서 빈번한 값-기반 술어의 삽입/삭제가 발생할 때 값-기반 술어 NFA의 갱신 비용을 줄여준다.

4. 실험 및 평가

4.1 실험 환경 설정

Pfilter와 ACfilter는 자바를 이용하여 구현했다. Yfilter의 경우 공개된 소스코드[2]가 있기 때문에 본 실험에서는 그것을 수정없이 사용했다. 본 논문에서 제시한 모든 실험은 펜티엄 IV 3.2GHz 프로세서와 메모리 1GB가 장착된 윈도우 서버 2003에서 자바 가상 머신 1.5.0을 이용하여 측정했다. 또한 자바 가상 머신에 할당할

2) http://yfilter.cs.berkeley.edu

Input: the set of (query id, path id, level, matching character info) pair satisfied with the structural matching, $PAIR_{structural}$
Output: the set of (query id, path id, level) pair satisfied with the structural and value-based predicate matching, $PAIR_{total}$

// *TSLL* denotes the target state linked list. And the 1 refers to the initial state number of NFA_{value}.
procedure START-ELEMENT-CONTENT-HANDLER(*element*)
1 $SL \leftarrow 0$; $RC \leftarrow element.contentLength$;
2 $TSLL \leftarrow \varphi$; $PAIR_{total} \leftarrow \varphi$;
3 $TSLL \leftarrow TSLL \cap \{(1, SL)\}$;
// Let *c* denote a current character. Then, let next state s_{next} be a state transited state *s* by the character *c*.
// And let $PAIR_{predicate}$ be a set of (query id, path id, level) pair of an accepting state, s_{next}.
procedure CHARACTER-HANDLER(*c*)
1 **for each** state $s \in TSLL$ **do**
2 **if** $\exists\ s_{next}$ **then**
3 $TSLL \leftarrow TSLL \cap \{(s_{next}, SL)\}$;
4 $RC \leftarrow RC - 1$;
5 **else**
6 **if** state $s \neq 1$ **then** $TSLL \leftarrow TSLL - \{(s, *)\}$;
7 **else** $SL \leftarrow SL + 1$;
8 **end if**
9 **end if**
10 **if** s_{next} = *"accepting state"* **then**
11 **if** $SL = 0$ **and** $RC = 0$ **then** $EHS \leftarrow EHS \cap PAIR_{predicate}$;
12 **elif** $SL > 0$ **and** $RC = 0$ **then** $PPHS \leftarrow PPHS \cap PAIR_{predicate}$;
13 **elif** $SL = 0$ **and** $RC > 0$ **then** $PSHS \leftarrow PSHS \cap PAIR_{predicate}$;
14 **elif** $SL > 0$ **and** $RC > 0$ **then** $PPSHS \leftarrow PPSHS \cap PAIR_{predicate}$;
15 **end if**
16 **end if**
17 **end for**
// Let p_{new} denote a (query id, path id, level) pair projection of *p*.
// And let *matchingcharacterinfo* denote a matching character info projection of *p*.
procedure END-ELEMENT-CONTENT-HANDLER(*element*)
1 **for each** $p \in element.PAIR_{structural}$ **do**
2 **if** *matchingcharacterinfo* = *"N"* **then** $PAIR_{total} \leftarrow PAIR_{total} \cap \{p_{new}\}$;
3 **elif** *matchingcharacterinfo* = *"E"* **then**
4 **if** $\exists\ p_{new}$ in *EHS* **then** $PAIR_{total} \leftarrow PAIR_{total} \cap \{p_{new}\}$;
5 **end if**
6 **elif** *matchingcharacterinfo* = *"P"* **then**
7 **if** $\exists\ p_{new}$ in *EHS* **or** *PPHS* **then** $PAIR_{total} \leftarrow PAIR_{total} \cap \{p_{new}\}$;
8 **end if**
9 **elif** *matchingcharacterinfo* = *"S"* **then**
10 **if** $\exists\ p_{new}$ in *EHS* **or** *PSHS* **then** $PAIR_{total} \leftarrow PAIR_{total} \cap \{p_{new}\}$;
11 **end if**
12 **elif** *matchingcharacterinfo* = *"PS"* **then**
13 **if** $\exists\ p_{new}$ in *EHS*, *PPHS*, *PSHS* **or** *PPSHS* **then** $PAIR_{total} \leftarrow PAIR_{total} \cap \{p_{new}\}$;
14 **end if**
15 **end if**
16 **end for**
17 **return** $PAIR_{total}$;

그림 7 값-기반 술어 NFA의 실행 알고리즘

가상 메모리의 최초, 최대 힙(heap) 크기를 512MB[3]로 설정함으로써 모든 실험 결과에 2차 보조기억장치의 I/O 영향이 실험 결과에 반영되지 못하게 했다. 이는 운영체제의 성능 스냅-인으로 확인할 수 있었다. 마지막으로 자바 가상 머신에서 쓰레기 수집자(garbage collector)의 영향을 피하기 위해 각 실험마다 새로운 프로세스를 생성하여 결과를 측정했음을 밝혀두는 바이다.

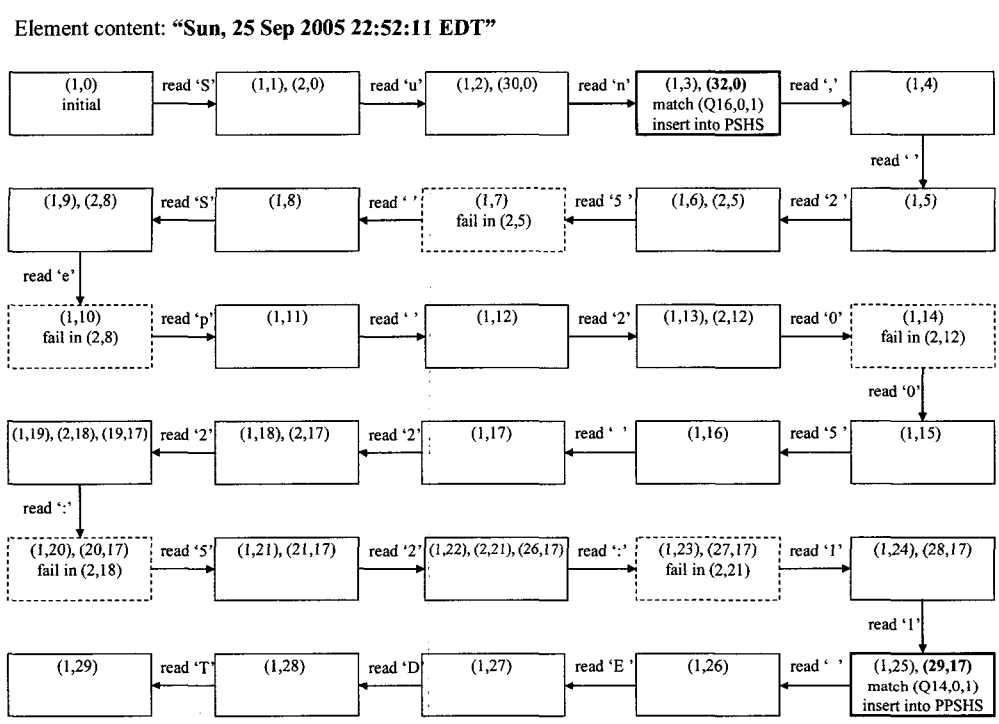
• 실험부하 생성

이미 서론에서 언급했듯이 세 필터링 기법은 XML과

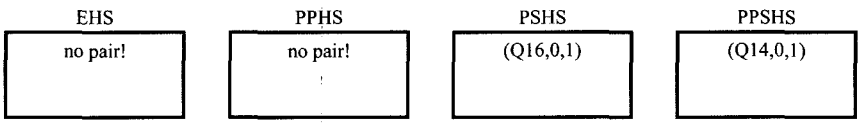
3) 자바 가상 머신을 구동하기 위해 -Xms512과 Xmx512m 옵션을 사용함

<rss><pubDate>Sun, 25 Sep 2005 22:52:11 EDT</pubDate></rss>

(a) XML 조각



(b) 값-기반 술어 매칭 부분



(c) 값-기반 술어 매칭 후 네 가지 해쉬 집합의 상태

Because of (Q13,0,1,**PS**), search EHS, PPHS, PSHS, and PPSHS → no match!
Because of (Q14,0,1,**PS**), search EHS, PPHS, PSHS, and PPSHS → **match!**
Because of (Q15,0,1,**S**), search EHS and PSHS → no match!
Because of (Q16,0,1,**S**), search EHS and PSHS → **match!**

(d) 해쉬 집합을 이용한 질의 매칭

그림 8 값-기반 술어 NFA의 실행 예

질의의 매칭을 위해 DTD 정보가 필요 없다. 그러나 DTD 정보는 실험을 위한 XML 데이타와 값-기반 술어를 생성하기 위해 필요하다. 합성된 XML 데이타에는 의도된 원소 내용을 삽입해야 한다. 그러나 지금까지 제안된 XMark[16], ToXgene[17], XML Generator[18]와 같은 XML 생성기는 이러한 요구사항을 만족시키지 못한다. 따라서 본 실험에서는 간단히 RSS 2.0 DTD에 있는 32개의 원소 중 원소 내용을 가질 수 있는 원소(단말 원소)에 실험 매개변수에 따른 문자열을 원소 내용으로 삽입하여 합성된 200개의 XML 문서를 생성했

다. 문자열을 만들기 위해 먼저 실험 매개변수에 의해 정해진 길이로 중복되지 않은 3000개의 단어를 생성한 후 균등 분포(uniform distribution)에 따라 단어를 연속적으로 선택하여 원하는 크기의 문자열을 만들어 낸다.

Yfilter 질의 생성기를 그대로 이용할 경우 값-기반 술어의 경로 표현식은 실험 매개변수에 따라 생성되지만 피연산자는 실험 매개변수에 따라 올바르게 생성되지 못하는 문제점이 있기 때문에 Yfilter 질의 생성기를 수정했다. 특히 Yfilter 질의 생성기의 수정에서 주목할 점은 앞에서 언급한 3000개의 단어 중 1000개 만이 값-기반 술어의 피연산자가 된다는 것이다. 값-기반 술어의 보다 정교한 실험을 위해 값-기반 술어는 각 피연산자에 나타나는 문자들의 비대칭도(skewness)를 반영하여 생성해야 한다. 이러한 요구사항은 질의와 XML의 매칭을 위해 합성된 XML의 원소 내용에도 적용된다. 이러한 비대칭도를 나타나기 위해서 본 실험에서는 Zipf 분포(Zipf distribution)[19]를 사용한다. 본 실험의 목적은 값-기반 술어에서 피연산자를 수행하기 위한 성능을 평가하는 것이기 때문에 생성할 질의의 경로에 관한 실험 매개변수는 고정시키는 것이 적절하다. 따라서 본 실험에서는 질의의 경로 개수와 경로 당 술어 개수는 1로 질의의 각 위치 단계에서 '*'와 '//'이 나타날 확률을 0.2로 고정했다. 그러나 주의할 사실은 본 실험에서 질의의 경로 개수를 1로 고정했지만 Pfilter와 ACfilter는 중첩 경로 질의의 수행을 지원하고 있다는 것이다. 표 1은 XML과 값-기반 술어를 합성하기 위한 실험 매개변수를 보여준다

표 1의 실험 매개변수 중 P와 Prefix, Suffix, PS에 대해 간단히 설명하면 P는 값-기반 술어의 피연산자에 접두 '%', 접미 '%', 또는 접두-접미 '%'에 관계없이 어느 것이라도 나타날 확률을 의미하는 반면 Prefix (Suffix, PS)는 '%'가 나타난 전체 값-기반 술어의 피연산자에서 그것이 접두 '%'(접미 '%', 접두-접미 '%')가 될 확률이다. 따라서 항상 $\mathbf{Prefix + Suffix + PS = 1}$ **을 만족**한다.

이 후부터 제공되는 실험결과는 200개의 XML을 처리하는데 걸리는 평균시간이다. 각 실험에서 실험 매개변수에 따라 값-기반 술어와 XML을 생성한다. XML 문서는 하나씩 디스크로부터 읽혀지며 XML에 대한 값-기반 술어의 실행 결과는 해당 XML을 매칭하는 값-기반 술어 번호가 된다.

• 측정기준

본 실험의 측정기준은 **다중-질의 처리 시간(multi-query processing time, MQPT)**이며 그 정의는 다음과 같다.

정의 4 (MQPT) 필터링 시스템에 문서-중심적 XML이 입력되는 시점을 t_{start}, 입력된 XML의 SAX 파싱이 완료된 시점을 $t_{parsing}$, SAX 파싱 이벤트를 이용하여 구조 NFA와 값-기반 술어 NFA의 천이를 완료하고 최종 매칭되는 질의를 찾아내는 시점을 t_{end}라 하자. 그럼 다중-질의 처리 시간(MQPT)은 $t_{end} - t_{parsing}$으로 정의할 수 있다.

이러한 다중-질의 처리 시간을 올바로 이해하기 위해 주의할 점은 필터링 시간의 정의인 $t_{end} - t_{start}$와 달리 다중-질의 처리 시간의 정의에는 XML을 파싱하는데 걸리는 시간이 포함되지 않다는 것이다.

4.2 실험 1: Pfilter, ACfilter, Yfilter의 비교

Yfilter는 특별한 매칭 문자 '%'를 포함한 질의를 처리하지 못하기 때문에 실험 매개변수 P, Prefix, Suffix, PS가 변할 때 MQPT를 Pfilter, ACfilter와 비교할 수 없다. 그러므로 [실험 1]에서는 값-기반 술어에서 특별한 매칭 문자 '%'가 나타날 확률을 0으로 고정(P=0)시킨다. 또한 Yfilter는 값-기반 술어의 피연산자 길이와 필터링되는 XML 문서의 원소 내용의 길이가 같을 때 (LO=LC) 만 값-기반 술어와 XML의 매칭이 발생하기 때문에 본 실험에서 두 실험 매개변수는 항상 같게 한다. Yfilter는 값-기반 술어 수행을 위해 일괄적, 순차적 문자열 동등 연산을 사용하기 때문에 실험 매개변수 ZC의 변화에 따른 MQPT를 Pfilter, ACfilter와 비교하는 것은 무의미하다. 그러므로 실험 매개변수 ZC=0으로 고정한다. 그러나 Pfilter와 ACfilter 사이의 비교에서 ZC의 변화는 의미가 있기 때문에 이에 대한 결과는 [실험 3]에서 자세히 다룬다.

그림 9는 값-기반 술어에서 평균 피연산자의 길이와

표 1 질의와 문서 생성을 위한 작업부하 매개변수

매개변수	범 위	설 명
Q	1000~500000	중복되지 않는 질의의 개수
ZC	0~2	피연산자에서 문자의 비대칭도
LO	2~64	모든 값-기반 술어에서 피연산자의 평균 길이
LC	2~1000	모든 XML 문서에서 원소 내용의 평균 길이
P	0~1	모든 값-기반 술어의 피연산자에서 '%'가 발생할 확률
Prefix	0~1	'%'를 가진 피연산자가 접두 '%'일 확률
Suffix	0~1	'%'를 가진 피연산자가 접미 '%'일 확률
PS	0~1	'%'를 가진 피연산자가 접두-접미 '%'일 확률

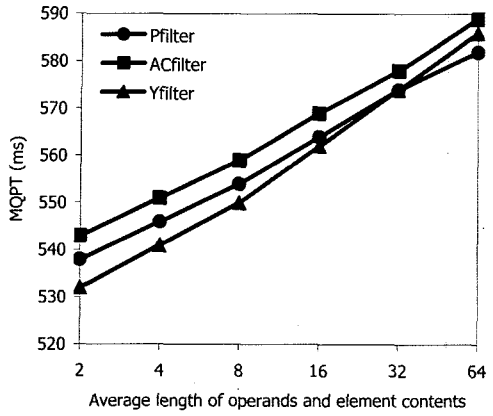


그림 9 피연산자와 원소 내용의 평균 길이가 변할 때 (Q=50000)

XML에서 원소 내용의 길이가 변할 때 MQPT를 측정한 그래프이다. 평균 피연산자의 길이가 32 보다 작을 때는 Yfilter가 더 좋은 MQPT를 나타내지만 평균 피연산자의 길이가 32 보다 크거나 같은 때는 Pfilter가 더 좋은 MQPT를 나타낸다. ACfilter는 실험 결과에서 보여주지 못했지만 평균 피연산자의 길이가 계속 증가하면 Yfilter 보다 더 좋은 MQPT를 나타낼 수 있음을 보이고 있다. 이러한 결과가 나타난 이유는 Yfilter의 경우 순차적인 문자열 동등 연산을 수행하기 때문에 MQPT 증가율이 Pfilter나 ACfilter에 비해 크기 때문이다. 그리고 Pfilter와 ACfilter의 MQPT 차이는 연결 리스트를 사용한 값-기반 술어 DFA 보다 해쉬 테이블을 사용한 값-기반 술어 NFA가 더 효율적이기 때문에 나타난 결과로 볼 수 있다.

사실 값-기반 술어의 피연산자 길이가 32인 질의를 작성하는 것은 현실 세계에서 드물게 발생한다고 볼 수 있다. 그러나 32라는 값은 구현 언어와 필터링 시스템이 구동되는 시스템의 사양에 따라 바뀔 수 있는 값이고 본 결과에서 중요한 점은 분명히 Pfilter와 Yfilter 사이에 특정 시점에서 MQPT가 역전되는 현상이 발생한다는 것이다.

4.3 실험 2: Pfilter와 ACfilter의 유지 비용

본 실험은 Pfilter와 ACfilter의 값-기반 술어 오토마타에서 피연산자의 삽입 비용을 알아본다. 공간적 제약 때문에 삽입 비용과 비슷한 결과를 보여주는 삭제 비용의 결과는 생략했다. 또한 Aho-Corasick 사전 매칭 트리를 그대로 사용한 ACfilter의 값-기반 술어 삽입/삭제 시간은 실패 함수 f의 계산 시간이 포함되었음에 주의하라.

Pfilter와 ACfilter의 유지 비용을 측정하기 위해 합성된 XML을 처리하여 매칭을 찾아낼 필요는 없다. 따라서 이와 관련된 실험 매개변수인 LC의 변화는 본 실험에서 고려하지 않는다. 또한 두 필터링 시스템에서 특별한 매칭 문자 '%'는 실제 값-기반 술어 오토마타에 피연산자를 삽입/삭제하는데 아무런 관련이 없고 단지 값-기반 술어를 수행할 때 어느 해쉬 집합에 (질의번호, 경로번호, 레벨) 쌍을 삽입할지에 관련이 있기 때문에 P=0으로 고정한다. P=0이기 때문에 Prefix, Suffix, PS와 같은 실험 매개변수는 그 값이 무엇을 갖던지 실험 결과에 영향을 미치지 못한다. 그리고 값-기반 술어에서 실험 매개변수 LO의 증가는 두 시스템에서 전반적인 삽입/삭제 시간의 증가를 가져오므로 8로 고정시켰다. 모든 합성된 값-기반 술어에서 경로 개수를 1로 고정했기 때문에 실험 매개변수 Q는 피연산자의 개수와 동일하다. 따라서 [실험 2]의 모든 결과는 4000개의 질의 즉, 4000개의 피연산자를 삽입/삭제하는데 걸리는 시간을 의미한다.

그림 10은 필터링 시스템에 등록된 값-기반 술어의 개수가 변할 때 4000개의 값-기반 술어를 삽입하는 시간을 보여준다. Pfilter에 등록된 질의의 개수가 증가할수록 각 상태 해쉬 테이블에서 문자 키가 공유될 확률이 증가하기 때문에 해쉬 테이블에 (문자 키, 다음 상태 번호) 쌍을 삽입하는 횟수가 줄어든다. 결국 이러한 현상은 삽입 시간을 낮춘다. ACfilter는 (문자 키, 다음 상태번호) 쌍을 삽입할 때 연결 리스트에서 문자 키의 중복을 막기 위해 각 상태 연결 리스트에 삽입할 문자 키가 있는지 확인하기 위해 연결 리스트의 탐색이 필요하다. 따라서 질의의 개수가 많아질수록 피연산자의 개수도 많아지고 값-기반 술어 DFA에서 각 상태 연결 리스트의 길이가 증가하기 때문에 탐색시간이 늘어나므로 삽입 시간이 점점 증가한다.

그림 11은 각 피연산자에서 문자들의 비대칭도가 증가할 때 4000개의 질의를 삽입하는 시간을 보여준다.

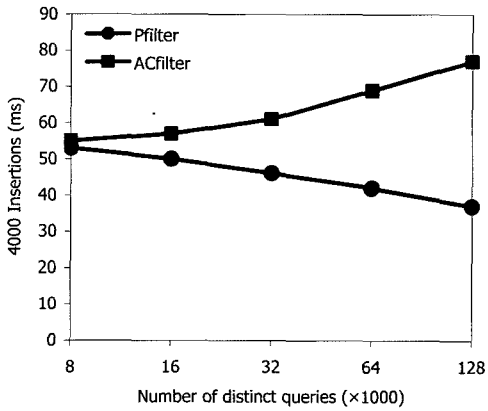


그림 10 4000개의 질의를 삽입하는 비용(varying Q, ZC=0)

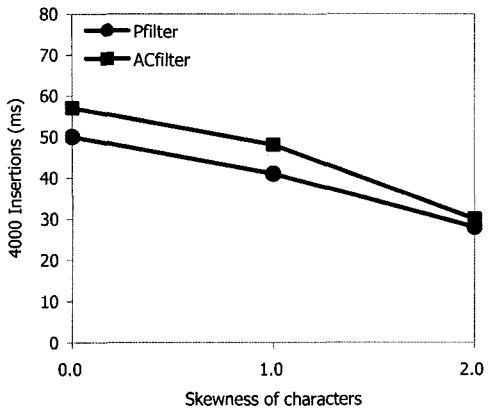


그림 11 4000개의 질의를 삽입하는 비용 (Q=16000, varying ZC)

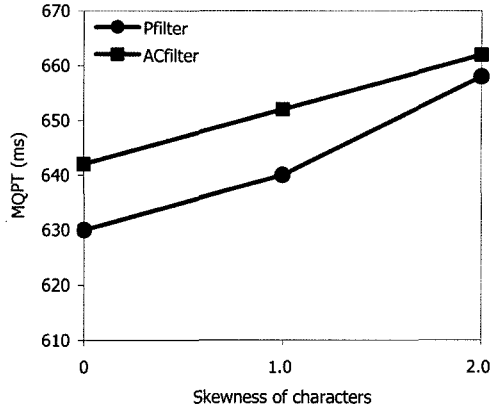


그림 12 문자의 비대칭도가 변할 때 (LO=8, LC=100, P=0.7, Prefix=Suffix=PS=1/3)

Pfilter와 ACfilter 모두 ZC가 증가할수록 삽입시간이 줄어드는 양상을 보인다. ZC가 증가할수록 값-기반 술어 오토마타에서 각 상태가 공유될 확률은 증가한다. 따라서 많은 상태가 공유될수록 목적 상태 해쉬 테이블과 목적 상태 연결 리스트에 각 (문자 키, 다음 상태번호) 쌍을 삽입할 필요가 없어진다. 이는 삽입시간의 감소를 야기한다. 여기서 주목할 점은 ACfilter의 경우 ZC가 증가할수록 삽입시간이 줄어드는 현상이 가속화 된다는 것인데 그 이유는 연결 리스트를 사용한 값-기반 술어 DFA는 각 상태에서 다음 상태 번호를 적게 가질수록 연결 리스트 탐색 시간이 해쉬 테이블을 사용한 값-기반 술어 NFA에서 탐색 시간보다 더 적게 걸리기 때문이다.

4.4 실험 3: Pfilter와 ACfilter의 효율성과 확장성

Yfilter의 경우 값-기반 술어에서 피연산자에 특별한 매칭 문자 '%'를 사용할 수 없기 때문에 본 논문에서 제안한 Pfilter와 효율성과 확장성을 적절하게 비교할 수 없다. 따라서 Yfilter와 Aho-Corasick 사전 매칭 트리를 결합한 ACfilter와 Pfilter를 중점적으로 비교한다. 또한 값-기반 술어의 피연산자 수행에 초점을 맞춰 비교하기 위해 XML 구조 매칭과 관련이 깊은 실험 매개변수 Q=50000으로 경로의 개수는 1로 고정시켰다. 특별한 매칭 문자 '%'가 존재하면 Yfilter에서와 달리 LO≤LC일 때 값-기반 술어와 XML의 매칭이 발생할 수 있으므로 기본값으로 LO=8, LC=100로 설정하였고 LO와 LC가 독립변수가 될 때 적절히 변화시켰다.

그림 12는 피연산자에서 문자들의 비대칭도가 변할 때 두 필터링 시스템의 MQPT를 보여주고 있다. ZC가 증가할수록 값-기반 술어 오토마타의 전체 크기 (상태 해쉬 테이블의 개수)가 줄어든다. Pfilter는 값-기반 술어 NFA의 크기가 줄어들더라도 다음 상태를 얻기 위한 해쉬 테이블 접근 시간은 일정하기 때문에 여기서 얻을 수 있는 이점이 없다. 그러나 ACfilter는 다음 상태로 천이되는 상태의 개수가 줄어들게 되면 연결 리스트의 원소 개수가 줄어들기 때문에 다음 상태를 찾기 위한 탐색 비용이 줄어들게 된다. ZC가 증가할수록 Pfilter와 ACfilter의 MQPT 차이는 줄어들게 된다. 그러나 ZC가 증가할수록 두 필터링 기법의 MQPT는 전반적으로 증가하는 양상을 보인다. 그 이유는 ZC가 증가할수록 매칭되는 값-기반 술어의 개수가 증가하면서 (질의번호, 경로번호, 레벨) 쌍을 4개의 해쉬 집합에 삽입하는 횟수가 늘어나기 때문이다.

그림 13은 평균 피연산자의 길이가 증가할 때 두 필터링 시스템의 MQPT를 보여준다. LO가 증가하게 되면 Pfilter와 ACfilter에서 상태 개수가 증가한다. 특히 두 기법에서 값-기반 술어 오토마타의 깊이는 LO와 상관관계가 깊다. 그러나 두 기법 모두 본 실험에서는 피

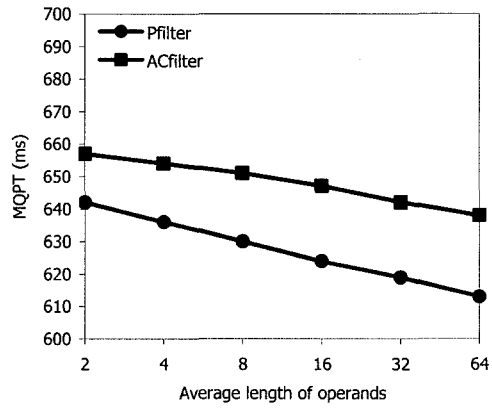


그림 13 피연산자의 평균 길이가 변할 때 (ZC=0, LC=100, P=0.7, Prefix=Suffix=PS=1/3)

연산자를 매칭하기 위해 천이를 테스트하는 횟수는 LC=100으로 같기 때문에 LO가 짧을 때는 값-기반 술어와 XML이 매칭되는 횟수가 늘어나게 된다. 따라서 4개의 해쉬 집합에 (질의번호, 경로번호, 레벨) 쌍을 삽입하는 횟수가 늘어나고 이는 MQPT를 높인다. 그러나 LO가 늘어나면 반대 현상을 나타낸다.

그림 14는 합성된 XML의 평균 원소 내용 길이가 증가할 때 두 기법의 MQPT를 보여준다. LC가 증가하면 값-기반 술어 오토마타의 입력이 그 만큼 많아지고 이는 천이 테스트의 횟수를 증가시키기 때문에 MQPT가 증가하는 현상은 당연하다. 해쉬 테이블을 사용한 Pfilter는 각 상태 해쉬 테이블의 접근 시간이 일정하기 때문에 연결 리스트를 사용한 ACfilter 보다 LC가 증가할수록 좋은 MQPT를 나타낸다. 또한 본 실험에서 확인할 수 있듯이 LC는 다른 실험 매개변수보다 Pfilter 기법의 MQPT에 가장 큰 영향을 미치는 매개변수임을 알 수 있다.

지금까지 Yfilter, Pfilter, ACfilter의 각 실험 매개변수의 변화에 따른 MQPT를 알아보고 특별한 매칭 문자 '%'을 수행할 수 있게 Yfilter를 변형한 ACfilter를 이용하여 Pfilter의 특징을 확인했다. Pfilter는 값-기반 술어를 효과적으로 처리하기 위해 공통 앞 부분 문자를 공유하기 때문에 이러한 접근법은 실세계의 값-기반 술어 매칭에서 여러 이점을 얻을 수 있다. 그 이유는 대부분 기사의 검색에 이용되는 값-기반 술어의 경우 특정 시간에 많이 검색되는 단어가 발생하는 특징을 가지고 있는데 그런 단어가 필터링 시스템에 많이 등록될수록 값-기반 술어에서 공통 앞 부분 문자 공유가 발생할 확률이 높아지기 때문이다. 물론 Yfilter와 Pfilter의 비교에서 LO와 LC가 작을 때는 값-기반 술어의 처리가 단순한 Yfilter가 우수한 성능을 나타내지만 LO와 LC가 점

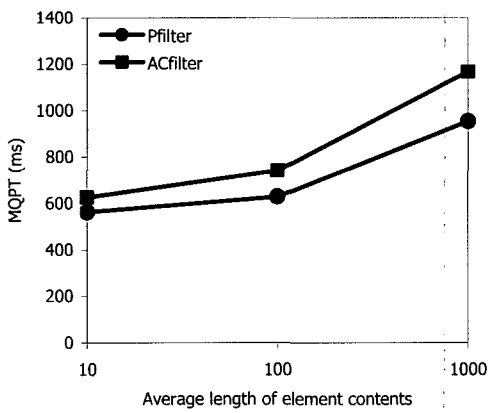


그림 14 원소 내용의 평균 길이가 변할 때 (ZC=0, LO=8, P=0.7, Prefix=Suffix=PS=1/3)

점 커짐에 따라 Pfilter의 성능이 우수해 짐을 알 수 있다. Pfilter와 ACfilter를 비교했을 때 삽입/삭제 비용 면과 효율성과 확장성 면에서 Pfilter의 성능이 우월함을 알 수 있다. 따라서 문서-중심적 XML을 필터링하기 위한 효과적인 기법은 Pfilter라고 할 수 있다.

5. 결 론

본 논문은 그 동안 많은 연구가 진행되었던 XML 필터링 시스템의 구조적 매칭을 확장하여 문서-중심적인 XML에 적합한 Pfilter를 제안했다. 이 기법은 다음과 같은 세가지 주요 특징을 갖는다.

첫째, Pfilter는 많은 양의 값-기반 술어를 처리할 수 있는 필터링 엔진으로 XML의 구조 매칭 뿐만 아니라 값-기반 술어의 피연산자 매칭을 위하여 공통 앞부분 공유된 NFA 기반 접근법을 사용한다. 본 연구에서 제안한 기법은 우리가 알고 있는 한 현재까지 제안된 XML 필터링 기법 중 값-기반 술어의 피연산자를 가장 효과적으로 처리할 수 있는 시스템이다.

둘째, 문서-중심적 XML에 효과적인 질의를 위해 제안된 Pfilter는 기존 XPath 명세에 추가적인 문법을 지원한다. 여기서 추가적인 문법이란, 기존 XPath의 text() 함수에서 피연산자 문자열에 SQL 문의 LIKE 연산자에서 사용되던 특별한 매칭 문자 '%'를 허용한 문법이다. 이렇게 확장된 문법은 기존 XPath 명세에 있는 conatins() 함수와 구별되는 데 그 차이점은 contains()의 경우 그 결과로 불리언 값을 반환하지만 특별한 매칭 문자 '%'를 사용한 값-기반 술어의 반환 값은 경로 질의에 해당하는 원소 내용이라는 점이다. 원래 XML 필터링 시스템의 출력이 불리언 값이라는 점을 감안하면 contains() 함수의 사용으로 요구사항을 만족시킬 수 있으나 값-기반 술어에 매칭되는 XML 조각으로 필터링 시스템의 출력을 확장하면 본 연구에서 제안한 확장된 XPath 명세가 필요하다. 추가적으로 본 논문에서 제안한 특별한 매칭 문자 '%'의 처리 기법은 contains() 함수의 구현에 별 다른 변경 없이 적용할 수 있다.

셋째, Pfilter의 개발 과정에서 구조 매칭과 값-기반 술어 매칭을 구분하여 구현했으므로 그 동안 많이 연구된 구조 매칭 기법을 선택적으로 사용할 수 있다. 이는 대부분의 구조 매칭 기법이 서로 다른 특징을 가지고 있기 때문에 상황에 맞는 적절한 구조 매칭 기법으로 대체할 수 있음을 의미한다.

Pfilter는 RSS XML, 웹 서비스 중 옥션 데이타[4], DBLP XML 데이타[5], 셰익스피어의 희곡 XML[6]과 같

4) http://www.cs.washington.edu/research/xmldatasets/data/auctions/yahoo.xml
5) http://dblp.uni-trier.de/xml/

은 문서-중심적 XML을 위한 필터링 기법이다. 이러한 XML의 특징은 일반적인 문서 데이타와 구조화된 데이타의 중간 형태로 특정 원소 내용이 상당히 긴 특징을 가지고 있다. 또한 그러한 원소 내용은 의미적으로 하위 원소로 나누기 곤란한 경우가 대부분이다. 이러한 XML을 필터링하기 위해 사용자는 XML 구조 매칭 뿐만 아니라 원소 내용이 상대적으로 긴 부분을 키워드를 사용하여 검색하기를 희망한다. 그러나 지금까지의 필터링 기법들은 문서-중심적 XML의 사용자 요구사항을 만족시키지 못했다. 이에 반해 Pfilter는 다량의 값-기반 술어가 필터링 시스템에 등록될 때 각 술어의 피연산자를 이용하여 공통 문자 앞부분 공유된 NFA를 구성함으로써 문서-중심적 XML 필터링에 효과적이다. 이러한 본 연구의 노력은 필터링 시스템에 키워드 기반 검색 기능을 추가시키며 사용자에게 보다 구체적인 필터링이 가능하도록 도와줄 것으로 예상된다.

참 고 문 헌

[1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 Second Edition W3C Recommendation. Technical Report REC-xml-200010006, *World Wide Web Consortium*.

[2] J. Kamps, M. Marx, M. de Rijke, B. Sigurbjörnsson. Best-match Query form Document-centric XML. *In Proceedings of the International Workshop on the Web and Databases*, Pages 55-60, 2004.

[3] J. Clark, and S. DeRose. XML Path Language (XPath) Version 1.0 W3C Recommendation. Technical Report REC-xpath-19991116, *World Wide Web Consortium*.

[4] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language W3C Working Draft. Technical Report WD-xquery-20050404, *World Wide Web Consortium*.

[5] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, Volume 18, Issue 6, Pages 333-340, 1975.

[6] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *The ACM Transactions on Database Systems*, Volume 28, Issue 4, Pages 467-516, 2003.

[7] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata and Stream Indexes. *The ACM Transactions on Databases Systems*, Volume 29, Issue 4, Pages 752-788, 2004.

[8] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. Index-based XML Multi-query Processing. *In Proceedings of the IEEE International Conference on Data Engineering*, Pages 139-150, 2003.

[9] C. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. *In Proceedings of the IEEE International Conference on Data Engineering*, Pages 235, 2002.

[10] V. Josifovski, M. Fontoura, and A. Barta. Querying XML Streams. *The International Journal on Very Large Data Bases*, Volume 14, Issue 2, Pages 197-210, 2005.

[11] J. Kwon, P. Rao, B. Moon, and S. Lee. FiST: Scalable XML Document Filtering by Sequencing Twig Patterns. *In Proceedings of the International Conference on Very Large Data Bases*, Pages 294-315, 2005.

[12] A. K. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, Pages 419-430, 2003.

[13] F. Tian, B. Reinwald, H. Pirahesh, T. Mayr, and J. Myllymaki. Implementing A Scalable XML Publish/Subscribe System Using Relational Database Systems. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, Pages 479-490, 2004.

[14] F. Peng, and S. S. Chawathe. XSQ: A Streaming XPath Engine. *The ACM Transactions on Databases Systems*, Volume 30, Issue 2, Pages 577-623, 2005.

[15] D. Megginson. SAX: A Free API for Event-based XML Parsing. Available at http://www.saxproject.org, 2005.

[16] A. R. Schmidt, F. Waas, M. L. Kersten, I. Manolescu, M. J. Carey, and R. Busse. XMark: A Benchmark for XML Data Management. *In Proceedings of the International Conference on Very Large Data Bases*, Pages 974-985, 2002.

[17] D. Barbosa, A. Mendelzon, J. Keenleyside and K. Lyons. ToXgene: a template-based data generator for XML. *In Proceedings of the International Workshop on the Web and Databases*, Pages 49-54, 2002.

[18] A. L. Diaz and D. Lovell. XML Generator. Available at http://www.alphaworks.ibm.com/tech/xmlgenerator, 2005.

[19] C. D. Manning and H. Schütze. Foundations of Statistical Natural Language Processing. *The MIT Press*, 1999.

6) http://www.ibiblio.org/xml/examples/shakespeare

이 경 한

2001년 서강대학교 컴퓨터학과(공학사)
2006년 서강대학교 컴퓨터학과(공학석사)
2006년 2월~현재 삼성전자 무선사업부 연구원. 관심분야는 스트리밍 XML 질의 처리, XML 인덱싱, 개인 영역 통신망(PAN) 및 임베디드 소프트웨어 개발 기법임.

박 석

정보과학회논문지 : 데이타베이스
제 33 권 제 1 호 참조