프리픽스 매취 조인을 이용한 XML 문서에 대한 분기 경로 질의 처리

(Branching Path Query Processing for XML Documents using the Prefix Match Join)

박 영 호 † 한 욱 신 †† 황 규 영 †††
(Young-Ho Park) (Wook-Shin Han) (Kyu-Young Whang)

요 약 본 논문에서는 정보 검색(Information Retrieval, IR) 기술과 새로운 인스턴스 조인 기술을 이용하여 방대하고도 이질적인 XML 문서들에 대한 부분 매취 질의(Partial Match Query)를 처리하는 새로운 방법으로, XIR-Branching을 제안한다. 부분 매취 질의는 경로 표현식에 조상-후손 관계성(descendent-or-self axis) "//"를 가지는 질의로 정의되며, **선형 경로 표현식(Linear Path Expression, LPE)**과 **분기 경로 표현식(Branching Path Expression, BPE)**으로 구분된다. 일반적 형식의 부분 매취 질의는 분기하는 경로들을 만드는 분기 조건들을 가진다. XIR-Branching의 목적은 이질적인 스키마들을 가진 방대한 문서들에 주어지는 부분 매취 질의를 효과적으로 지원하는 것이다. XIR-Branching은 관계형 테이블을 사용하는 전통적인 스키마-레벨 방법들(XRel, XParent, XIR-Linear[21])에 그 기초를 두고, 역 인덱스(inverted index) 기술과 새롭게 소개하는 인스턴스-레벨 조인 기술인 프리픽스 매취 조인(Prefix Match Join)을 사용하여 질의 처리 효율성과 확장성을 향상시킨다. 전자는 LPE를 처리하기 위한 기술로 XIR-Linear[21]에서 사용한 방법이다. 후자는 BPE를 처리하기 위한 기술로 본 논문에서 새롭게 제안하는 기술이며, 전통적인 방법에서 사용하는 포함 관계 조인(containment join) 보다 효과적인 방법으로 결과 노드(result node)를 찾는다. 기존 연구인 XIR-Linear는 역 인덱스를 사용하여 LPE 처리에 우수한 성능을 보이고 있지만, BPE 처리 방법을 다루지 않았다. 그러나, 더욱 구체적이고 일반적인 질의를 위해서는 BPE도 처리할 수 있어야 한다. 본 논문에서는 BPE까지 다룰 수 있는 새로운 방법으로 기존의 XIR-Linear를 확장한 XIR-Branching을 제안한다. 제안하는 방법은 스키마-레벨 방법으로 질의 대상 후보 집합을 크게 줄인 후, 인스턴스-레벨 조인 방법인 프리픽스 매취 조인으로 최종 결과 집합을 효과적으로 구하는 방법이다. XIR-Branching의 우수성을 보이기 위해 기존 BPE 처리 방법인 XRel, XParent와 비교 분석을 수행한다. 마지막으로, 성능 평가를 통하여 XIR-Branching이 기존 방법들에 비해 수십에서 수백배 효과적이고 확장성 또한 뛰어남을 보인다.

키워드 : XML, 역 인덱스, 프리픽스 매취 조인, 부분 매취 질의, 분기 경로 표현식, XIR-Branching, XIR-Linear

Abstract We propose XIR-Branching, a novel method for processing partial match queries on heterogeneous XML documents using information retrieval(IR) techniques and novel instance join techniques. A partial match query is defined as the one having the descendent-or-self axis "//" in its path expression. In its general form, a partial match query has branch predicates forming branching paths. The objective of XIR-Branching is to efficiently support this type of queries for large-scale documents of heterogeneous schemas. XIR-Branching has its basis on the conventional schema-level methods using relational tables(e.g., XRel, XParent, XIR-Linear[21]) and significantly improves their efficiency and scalability using two techniques: an inverted index technique and a novel prefix match join. The former supports linear path expressions as the method used in XIR-Linear[21]. The latter supports branching path expressions, and allows for finding the result nodes more efficiently than

· 본 연구는 첨단정보기술연구센터(AITrc)를 통한 한국과학재단 및 BK21의 지원을 받았음
† 비 회 원 : 한국과학기술원 전산학과
yhpark@mozart.kaist.ac.kr
†† 종신회원 : 경북대학교 컴퓨터공학과 교수
wshan@kpu.ac.kr
††† 종신회원 : 한국과학기술원 전산학과 교수
kywhang@cs.kaist.ac.kr
논문접수 : 2004년 8월 18일
심사완료 : 2005년 4월 27일

containment joins used in the conventional methods. XIR-Linear shows the efficiency for linear path expressions, but does not handle branching path expressions. However, we have to handle branching path expressions for querying more in detail and general. The paper presents a novel method for handling branching path expressions. XIR-Branching reduces a candidate set for a query as a schema-level method and then, efficiently finds a final result set by using a novel prefix match join as an instance-level method. We compare the efficiency and scalability of XIR-Branching with those of XRel and XParent using XML documents crawled from the Internet. The results show that XIR-Branching is more efficient than both XRel and XParent by several orders of magnitude for linear path expressions, and by several factors for branching path expressions.

Key words : XML, inverted index, prefix match join, partial match queries, branching path expressions, XIR-Branching, XIR-Linear

1. 서 론

최근, XML 문서[1]들에 대한 질의 처리를 위해 의미 있는 연구들이 진행되어 왔다. 그러나, 우리가 알기로 대부분의 XML 질의 처리 연구는 정해진 스키마를 가진 한정된 수의 문서들만을 고려하였으므로, 인터넷 검색 엔진[2,3]과 같은 이질적인 스키마(heterogeneous schema)를 다루는 대규모 응용에 적합하지 않다. 그러므로, 이러한 응용을 위한 새로운 방법이 필요하다는 것을 인식하고, 새로운 방법의 제안에 초점을 맞추고자 한다.

XPath[4]에서의 부분 매취 질의는 XML 문서들의 스키마가 매우 이질적인 상황에서 스키마 정보를 부분적으로만 알고 있을때 질의 대상 XML 문서들을 찾는데 특히 유용할 수 있다. 여기서, **부분 매취 질의(partial match query)**란, 경로 표현식에 조상-후손 관계성(*descendent-or-self axis*) "//"를 가지는 질의로 정의된다. **전체 매취 질의(full match query)**[5]는 부분 매취 질의의 특별한 경우로 여겨질 수 있다. 예를 들어, 그림 1과 같이 네 개의 웹 싸이트에 저장된 **논문(paper)**에 관한 XML 문서들을 생각해 보자. 주목할 사항은 이들 웹 싸이트들에 존재하는 스키마들이 이질적이라는 것이다. 이질적인 XML 문서들 가운데 저자(author)의 이름(name)을 찾고자 한다면, 사용자는 부분 매취 질의 *//author//name*을 사용해야 한다.

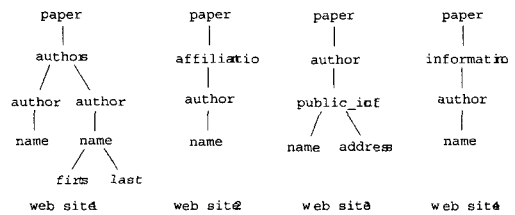


그림 1 네개의 웹 싸이트에 존재하는 논문(paper)에 관한 XML 문서들

부분 매취 질의는 **선형 경로 표현식**(*Linear Path Expression, LPE*)과 **분기 경로 표현식**(*Branching Path Expression, BPE*)으로 구분될 수 있다. LPE는 레이블들 간에 부모-자식 관계나 조상-후손 관계를 가진 레이블들의 연속(sequence)으로 구성된 경로 표현식으로 정의될 수 있다. BPE는 그 LPE 내에 있는 하나 이상의 레이블에 분기하는 조건(branching condition)이 있는 경로 표현식으로 정의된다(LPE와 BPE에 대한 보다 정형적인 정의는 2.2절에서 소개한다.)

부분 매취 질의를 처리하기 위한 기존 방법들은 질의 처리에 사용하는 정보의 레벨에 따라 스키마-레벨 방법과 인스턴스-레벨 방법의 두 가지 타입으로 구분될 수 있다. 전자는 질의 처리 시 먼저 스키마-레벨의 정보를 사용하여 질의를 처리하고, 이후 인스턴스-레벨 정보를 사용하는 방법들이고, 후자는 질의 처리 시 인스턴스-레벨 정보만을 사용하여 질의를 처리하는 방법들이다. 스키마-레벨 방법의 예로는 XRel[6], XParent[7,8], Index Fabric[9] 등이 있다. 인스턴스-레벨 방법의 예로는 대부분이 Numbering Scheme을 사용하는 구조적 조인(Structural Join) 방법들로써 Element-Attribute Join Scheme[10], Multi-Predicate Merge Join[11], 스택을 이용한 Structural Join[12], Holistic Twig Join[13]과 그의 변형들[14,15], XQuery/IR[16], Keyword Search on XML-QL[17], XRANK[18] 등이 있으며, 두 가지 인스턴스-레벨 질의 처리 모델인 트리 항해 모드와 역 리스트 필터링 모드를 혼용하는 Mixed Mode 방법[19]이 있다. 이 방법들 가운데, 첫번째 타입의 방법들은 LPE와 BPE에 모두 사용할 수 있으나, 이질적인 스키마를 가진 대규모 문서들에 사용하기 적합하게 디자인되어 있지 않는 방법이거나[6-8], LPE들을 제한적으로만 지원하고 BPE에 대한 처리 방법이 명확하지 않은 방법이다[20]. 두번째 타입의 방법들은 LPE와 BPE를 모두 지원하지만, 효율성이 떨어져 대규모 데이타베이스 환경에서 최선의 선택이 될 수 없다. 이들 두 분류의 방법들 중에서 인스턴스-레벨 방법에 비해 스키마-레벨

방법이 대규모 XML 문서들에 대한 처리에 더욱 적합하다. 왜냐하면, 이것은 스키마-레벨에서 문서 인스턴스들을 **거르는**(filter out) 능력을 가지고 있기 때문이다. 그러므로, 본 논문에서는 스키마-레벨 방법을 우리 방법의 기본으로 적용하였다. 보다 자세한 내용들은 3장에서 볼 수 있다.

본 논문은 특히, XRel[6], XParent[7,8]와 같이 **관계형 테이블(*relational table*)**들을 사용하는 스키마-레벨 방법들에 기반한다. 그 이유는 다음 두 가지 이다. 첫째, 이 방법들은 관계형 DBMS에서 잘 정립되어 있는 기술을 XML 질의 처리에 활용할 수 있다. 두번째, 이 방법들은 SQL 또는 그의 확장 버전을 활용하여 XML 문서들에 질의할 수 있다. 질의 처리시 그들은 XML 문서의 스키마 정보와 인스턴스 정보를 관계형 테이블에 저장하고, 다음 두 단계에서 부분 매취 질의를 수행한다. 첫번째 단계에서 질의의 경로 표현식을 매취하는 스키마를 가진 문서들을 먼저 찾고, 두번째 단계에서 경로 표현식에 명시된 선택 조건들이 있다면, 그들을 모두 만족하는 문서들을 찾는다. 그러나, 기존 방법인 XRel과 XParent의 질의 처리 효율성은 6장에 있는 실험에서 볼 수 있는 바와 같이 대규모 응용에 대해서 너무 제한적이다.

본 논문에서 제안하는 방법은 *XIR-Branching*이라 부른다. 이 방법의 목적은 이 두 단계의 효율성을 모두 향상시키는 것이다. XIR-Branching은 주어진 질의에 대해 스키마 레벨 방법을 사용하여 구조를 먼저 찾은 뒤, 인스턴스 레벨 방법을 사용하여 결과 노드를 찾는 방법이다. 구체적으로, 첫번째 단계인 스키마-레벨에서는 기존 연구인 *XIR-Linear*[21]의 역 인덱스(inverted index)[22] 기술을 그대로 사용한다. 이때, 역 인덱스 기술은 정보 검색(Information retrieval, IR) 분야에서 전통적으로 사용해 온 기술로서 스키마 정보를 구성하는 단 몇 개의 키워드들 즉, 부분적인 스키마 정보만을 가지고 대규모 문서들을 찾는 응용에 성공적으로 사용되어 왔다. *XIR-Linear*[21]는 부분 매취 질의 시 이러한 역 인덱스를 사용하여 이질적인 대규모 XML 문서들을 효과적으로 찾을 수 있는 장점을 가지나, 두번째 단계인 인스턴스 레벨 문제에 초점을 맞추고 있지 않다. 그러므로, 본 논문에서는 이를 위하여 **프리픽스 매취 조인**(*prefix match join*)라고 부르는 새로운 조인 기술을 소개하여 대량의 인스턴스 정보를 찾을 수 있는 방법을 제안한다.

BPE는 LPE의 포함 집합(superset)이므로, XIR-Branching은 기능적으로 기존의 XIR-Linear를 확장한 것이다. 즉, XIR-Branching은 XIR-Linear를 포함하는 방법으로 기본적인 질의 형태인 LPE 뿐만 아니라 구체적인 질의를 위한 BPE 까지도 효과적으로 처리할 수 있다. 본 논문에서는 먼저 XIR-Linear의 LPE 처리 방법과 그 처리 특징을 소개한 후, XIR-Linear 처리 방법을 기반으로 하는 XIR-Branching을 소개한다. 본 논문에서는 XIR-Branching의 성능을 XRel, XParent와 비교 분석하고, 인터넷에서 크로울하여 모은 실제 XML 문서 집합들을 사용하여 실험함으로써 분석 내용을 입증한다. 실험 결과로서 XIR-Branching이 XRel이나 XParent에 비교하여 LPE 처리 능력이 수십배 우수하며, BPE 처리 능력은 수배 우수하다는 것을 보인다. 또한, 실험 결과는 질의된 XML 문서들의 수가 증가하는 환경에서 XIR-Branching이 좋은 확장성을 가지고 있다는 것을 보인다.

본 논문은 이질적인 대규모 XML 문서들에 대한 BPE 처리를 위해 다음과 같은 새로운 공헌을 제시한다.

- XIR-Branching은 보다 구체적인 질의 형태인 BPE를 효과적으로 처리하기 위하여 프리픽스 매취 조인이라 부르는 새로운 인스턴스-레벨 조인을 제안한다. 프리픽스 매취 조인은 분기 조건들을 만족하는 인스턴스 노드들을 찾기 위한 조인의 수를 최소화함으로써 성능을 의미있게 향상시킨다.
- XIR-Branching은 XIR-Linear의 LPE 처리 기능과 유기적으로 결합하는 BPE 처리 알고리즘을 제안하였다. BPE 처리 알고리즘은 역 인덱스를 이용해서 먼저 스키마 정보를 찾고, 이후 프리픽스 매취 조인을 이용해서 인스턴스 정보를 찾는 방법으로 역 인덱스와 프리픽스 매취 조인의 자연스러운 결합을 통하여 LPE 뿐만아니라 BPE 처리도 우수하게 만든다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문에서 사용하는 XML 문서 모델과, 질의 모델을 소개한다. 3장에서는 XIR-Linear를 포함한 기존 질의 처리 방법들에 대하여 논의한다. 4장에서는 XIR-Branching의 저장 구조를 제안하고, 5장은 4장에서 제안한 저장 구조를 기반으로 하는 질의 처리 알고리즘을 제안한다. 6장에서는 기존 방법인 XRel과 XParent와 비교한 XIR-Branching의 성능 평가 결과를 제시한다. 마지막으로 7장에서는 본 논문의 결론을 내린다.

2. 연구 배경

2.1 XML 문서 모델

본 논문에서 사용하는 XML 문서 모델은 Bruno 등[13]에서 제안한 모델을 기반으로 하고 있다. 이 모델에서, XML 문서는 루트가 있고, 순서가 있으며, 레이블이 기록된 트리(rooted, ordered, labeled tree)로 표현된다. 이 트리의 노드(node)는 엘리먼트(element), 애트리뷰트(attribute), 값(value) 중 하나를 나타낸다. 또한, 이 트

리의 에지(edge)는 엘리먼트-서브엘리먼트 관계성(element-subelement relationship), 엘리먼트-애트리뷰트 관계성(element-attribute relationship), 엘리먼트-값 관계성(element-value relationship), 애트리뷰트-값 관계성(attribute-value relationship) 중 하나를 나타낸다. 엘리먼트와 애트리뷰트 노드는 XML 문서의 구조를 결정하는 요소로서, 각 노드에게는 레이블(label, 즉, 이름)과 고유 식별자(unique identifier)를 부여한다. 그림 2는 XML 문서의 XML 트리 예를 보인다. 이 그림에서 두 개의 애트리뷰트 값(attribute value) "R"과 "T"를 나타내는 노드 번호 15와 27을 제외한, 모든 단말 노드(leaf node)들은 엘리먼트 값(element value)들을 나타내고, 애트리뷰트 @category를 나타내는 노드 번호 14와 26을 제외한, 단말 노드가 아닌 모든 노드들은 엘리먼트를 나타낸다. 이때, 애트리뷰트는 그 레이블 앞에 프리픽스 '@'를 사용함으로서 엘리먼트와 구분한다.

본 논문에서 사용하는 모델은 노드를 값(value)이 아닌 엘리먼트나 애트리뷰트를 표현하도록 참고 문헌[13]의 모델을 수정하고, 다음 정의 1과 같이 레이블 경로의 개념을 가지도록 하였다.

정의 1. XML 트리 내의 **레이블 경로**(*label path*)는 그 트리의 루트 노드로 부터 특정 노드 p까지의 노드 레이블 $l_1, l_2, ..., l_p (p \geq 1)$들의 연속(sequence)으로 정의되고, $l_1, l_2, ..., l_p$로 표현된다[21]. □

정의 2. XML 트리 내의 **노드 경로**(*node path*)는 그 트리 내의 루트 노드로 부터 특정 노드 p 노드 식별자 $n_1, n_2, ..., n_p (p \geq 1)$ 들의 연속(sequence)으로 정의되고, $n_1, n_2, ..., n_p$로 표현된다. □

레이블 경로는 XML 문서 구조를 표현하며, 이를 **스키마-레벨** 정보(schema-level information)라 하고, 이에 반하여, 노드 경로는 XML 문서 인스턴스를 표현하며, 이를 **인스턴스-레벨** 정보(instance-level information)라 한다. 본 논문에서는, 레이블 경로는 경로 표현식과 "매칭(match)된다", 노드 경로는 레이블 경로에 "속한다", 노드 경로는 경로 표현식으로 부터 "구해진다"고 말한다. 예를 들어, 그림 2에서 ***issue.editor.first***는 경로 표현식 ***//editor//first***과 매칭하는 레이블 경로이고, 1.2.3, 1.7.8은 레이블 경로에 "속"하는 노드 경로들이다. 이때, 같은 구조를 가지는 하나 이상의 인스턴스들이 있을 수 있기 때문에, 같은 레이블 경로에 속하는 하나 이상의 노드 경로들이 있을 수 있다.

2.2 XML 질의 모델

우리의 질의 언어는 트리 패턴 질의(tree pattern query, TPQ) 클래스[23]에 속한다. 이 질의 언어는 두 종류의 경로 표현식을 지원한다. 그들은 선형 경로 표현식(linear path expression, LPE)과 분기 경로 표현식(branching path expression, BPE)이다.

정의 3에서 정의된 바와 같이, LPE는 '/' 나 '//'로 연결된 레이블(label)들의 시퀀스로서 표현된다.

정의 3. 선형 경로 표현식(*linear path expression*)은 $l_0\ o_1\ l_1\ o_2\ l_2\ ...\ o_n\ l_n$로서 정의된다. 이때, $l_i (i=1,2,...,n)$은 그 경로 내에 존재하는 i 번째 레이블이고, $o_j (j=1,2,...,n)$는 '/' 나 '//' 중 하나이며, 이들 각각은 l_{j-1}와 l_j 간의 부모-자식 관계성(parent-child relationship)이나 조상-후손 관계성(ancestor-descendant relationship)을 나타낸다. 여기서, l_0는 모든 XML 문서들의 집합인 XML 트리들을 모두 하나로 연결하는 가상의 루트 노드이다. □

정의 5에서 정의된 바와 같이, BPE는 어떤 레이블들 $l_i (i \in \{1,2,...,n\})$에 대해서, '분기 조건 표현식' $[C_i]$가

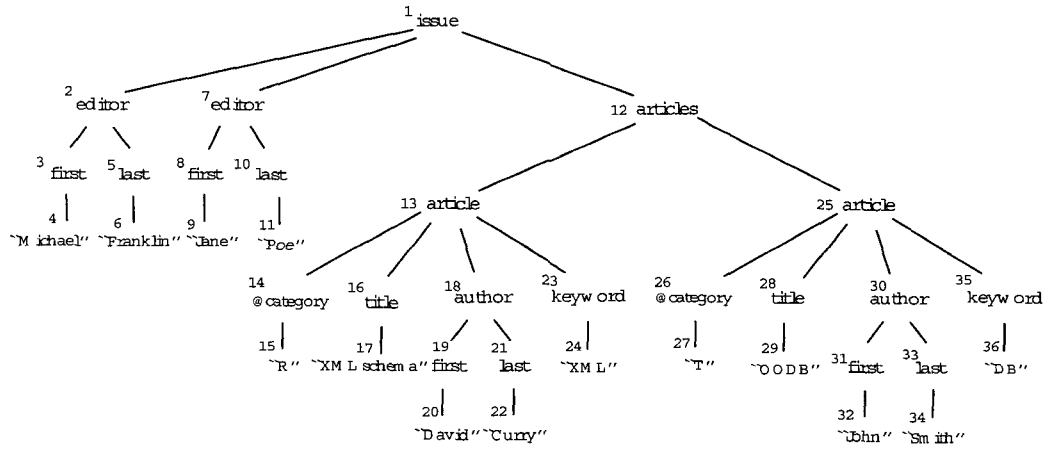


그림 1 XML 문서의 XML 트리 예

추가된 LPE로서 표현된다[4]. 다른 문헌[24,25]의 연구와 같이, 본 논문에서는 정의 4의 정의대로 분기 조건 표현식에 대한 단순 선택 조건(simple selection predicate) 만을 다룬다.

정의 4. 분기 조건 표현식(*branch predicate expression*) C_i는 각기 불리언 표현식(Boolean expression)으로 정의된다. 이들 불리언 항(Boolean term)들은 단일 레이블 l 이거나, 조건(predicate) $l\ \theta\ v$ 중 하나이다. 여기서, v는 상수이고, θ는 비교 연산자($\theta \in \{=, \neq, >, \geq, <, \leq\}$)이다. 단일 레이블 l은 그 레이블을 가진 엘리먼트나 애트리뷰트의 존재 유무를 명시하고, $l\ \theta\ v$는 선택 조건(selection condition)을 만족하는 레이블들을 명시한다. □

분기 경로 표현식(Branching Path Expression)은 선형 경로 표현식(Linear Path Expression) 내의 임의의 레이블에 조건을 부여할 수 있는 형태로 정의할 수 있다. 분기 패스 질의(Branching Path Query)에 관한 연구[26]에서는 XML 문서를 그래프로 모델하고 그래프에 대한 분기 경로 표현식을 정의하였다. 그러나, XML 문서는 트리로 표현하는 것이 자연스럽기 때문에, 본 연구에서는 다른 연구[15,20]에서와 같이 XML 문서를 트리로 표현한다. 본 연구에서는 기존 연구[26]와 유사한 내용으로 분기 경로 표현식을 정의할 수 있으며, 기존 연구[26]에서 XML 문서를 트리로 모델하고 싸이클 표현을 제거하면, 정의 5와 같이 새롭게 정의할 수 있다.

정의 5. 분기 경로 표현식(*branching path expression*) 은 은 $l_0\ o_1\ l_1[C_1]\ o_2\ l_2[C_2]\ \ldots\ o_n\ l_n[C_n]$ 로서 정의된다. 이때,(1) $l_0\ o_1\ l_1\ o_2\ l_2\ \ldots\ o_n\ l_n$ 는 정의 3에서 정의된 LPE이고,(2) LPE에 존재하는 레이블에 명시될 수 있는$[C_1]$ ...$[C_n]$는 해당 레이블에 속한 노드 인스턴스에 대한 필터링 조건을 명시한 것이다. 이때, $[C_k](k = 1,2, \ldots, n)$는 정의 4에서 정의된 **분기 조건 표현식**이다. 이들 중(전체가 아닌) 일부가 생략될 수 있다. □

다음 질의는 "*Curry*"라는 값을 가진 *last* 엘리먼트를 자식으로 가진 *author* 엘리먼트가 후손이 되는 *issue* 엘리먼트들 중, 후손으로 적어도 하나의 *keyword* 서브엘리먼트를 포함하고 있는 *article* 엘리먼트들을 선택한 후, 이들 중 자식인 *title* 엘리먼트들을 찾는 BPE 예이다.

Q1:: //issue[//author/last = "Curry"]//article
[/keyword]/title

Q1는 LPE //issue//author/last 내의 레이블인 *last*에 대해서 선택 조건을 가지며, LPE //issue//article/keyword 내의 레이블인 *keyword*에 대해 존재여부 확인조건(existential condition)을 가지는 BPE이다.

2.3 질의 패턴

본 논문은 정의 6에서 정의된 **질의 패턴**(*query pattern*)을 가지고 경로 표현식을 모델한다. 본 논문에서는 사용된 개념들을 정형적으로 표현하기 위하여 Holistic Twig Join[13]에서 처음 사용된 트위그 패턴(twig pattern) 정의를 약간 확장하였다. 이 정의는 정의 5에서 정의한 BPE에 기반한다.

정의 6. 정의 5의 정의 중, l_0가 생략된 경로 표현식 $l_0\ o_1\ l_1[C_1]\ o_2\ l_2[C_2]\ \ldots\ o_n\ l_n[C_n]$의 경우, 그 경로 표현식을 이진 트리(binary tree)와 그 트리의 루트 노드에 연결된 **매달린 에지**(*dangling edge*) 로 구성되어 있는 **질의 패턴**(*query pattern*) 으로 표현한다. 이 **질의 패턴**은 다음의 성질들을 가진다.

- 에지는 경로 표현식 내의 $o_j(j \in 1,2, \ldots, n)$를 나타낸다. 에지는 만약, 경로 표현식 내의 o_j가 '/'일 경우는 단일 실선으로, '//'일 경우는 이중 실선으로 표현된다. 질의 패턴의 루트 노드에 연결된 매달린 에지는 o_1을 나타낸다.
- 노드는 경로 표현식 내의 레이블 $l_k(k \in 1,2, \ldots, n)$를 나타낸다. 루트 노드는 경로 표현식 내의 레이블 l_1을 나타낸다.
- 경로 표현식 내의 레이블 $o_k(k \in 1,2, \ldots, n-1)$를 나타내는 노드의 좌측 자식 노드는 l_{k+1}를 나타낸다.
- 경로 표현식 내의 레이블 $l_k(k \in 1,2, \ldots, n)$를 나타내는 노드의 우측 서브 트리는 분기 조건 표현식 $C_k \equiv o_{k1}l_{k1}o_{k2}l_{k2} \ldots o_{kp}l_{kp}(p \geq 1)$를 나타내는 질의 패턴이다.
- 만약, 노드의 레이블이 노드에 대한 선택 조건("$\theta\ v$")을 가진다면, 그 노드는 "$\theta\ v$"라는 윗 첨자를 가진다. □

본 논문에서는 질의 패턴과 관련하여 다음과 같은 용어를 사용한다.

- 질의 패턴의 노드들 중 하나는 질의 결과로서 검색된다. 이 노드는 정의 3에서 정의된 LPE나 정의 5에서 정의된 BPE 내의 레이블 l_n에 대응된다. 이 노드를 **결과 노드**(*result node*)라 부르고, 결과 노드를 다른 노드들과 구분하기 위하여 회색톤으로 표현한다.
- 질의 패턴 내에 있는 노드들 중 일부 노드는 우측 서브 트리를 가진다. 우측 서브 트리를 가지는 노드는 **분기하는 노드**(*branching node*)라 부른다. 정의 5에서 볼 수 있는 표현식 중 $l_k[C_k]$에서, 레이블 l_k를 분기하는 노드라고 한다. 이러한 분기하는 노드는 레이블 l_k에 속하는 모든 노드들 중에서 조건$[C_k]$를 만족하는 노드를 구하기 위해 사용된 표현이다.

그림 3은 2.2절의 질의 Q1의 질의 패턴을 보이고 있다. *title* 노드는 결과 노드이고, *issue*와 *article* 노드는 분기하는 노드들이다.

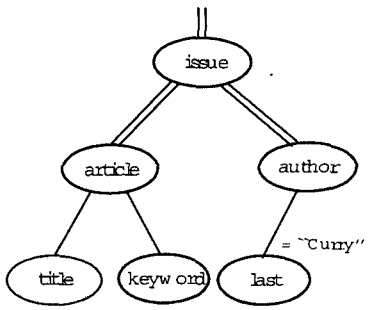


그림 3 질의 Q1의 질의 패턴

선형 질의 패턴(*linear query pattern*)은 질의 패턴의 특별한 경우로서 다음과 같이 정의한다.

정의 7. 선형 경로 표현식의 질의 패턴은 **선형 질의 패턴**(*linear query pattern*)이라 부른다. 정의 6에서 정의된 질의 패턴과 비교되는 것은, 선형 질의 패턴의 경우, 분기하는 노드(branching node)가 없다는 것이다. □

추가적으로, 본 논문에서는 질의 패턴에서 사용하는 **루트 노드, 단말 노드, 결과 노드, 분기하는 노드** 라는 용어와 경로 표현식에서 사용하는 **루트 레이블, 단말 레이블, 결과 레이블, 분기하는 레이블** 이라는 용어를 상호 혼용하여 사용한다. 예를 들어, 그림 3에서, *issue*는 질의 Q1의 루트 레이블이다; *title, keyword, last*는 단말 레이블들이다; *title*은 결과 레이블이다; 마지막으로, *issue*와 *article*는 분기하는 레이블들이다.

3. 관련 연구

서론에서 언급한 바와 같이, 경로 표현식을 처리하는 방법은 크게 스키마-레벨 방법과 인스턴스-레벨 방법으로 나누어 진다. 스키마-레벨 방법은 경로 표현식과 매취하는 노드들을 찾기 위해서 레이블 경로와 같이 구조적인 정보를 사용한다[6-8,20]. 반면, 인스턴스-레벨 방법은 노드의 시작(start)과 끝(end) 위치와 같이 노드 구분 정보 만을 사용한다[12-15]. 본 절에서는 인스턴스-레벨 방법을 간단히 설명한 후, 스키마-레벨 방법에 초점을 두어 설명한다.

3.1 인스턴스-레벨 방법

인스턴스-레벨 방법에는 세가지 다른 방법들이 소개되고 있다. 첫째는 XML 트리 항해 방법이다[27-29]. 이것은 경로 표현식을 상태 머신(state machine)[1]으로 변경하고, 상태 머신을 따라 XML 트리를 항해하는 방법으로 경로 표현식을 처리한다. 둘째는 XML 트리 내의 각 노드들을 위하여 저장된 노드 인스턴스 정보를 사용하는 방법이다[10-15,30]. 이것은 경로 표현식을(구조적) 조인 질의로 변경한 후, 노드 인스턴스 정보를 사용하여 질의를 처리한다. 세번째는 정보 검색(information retrieval, IR) 기술을 사용하는 방법으로, 특히 XML 문서들에 대해서 구축한 역 인덱스를 사용한다[16-19]. 이 절의 나머지 부분에서는 XML 트리를 항해하는 방법인 첫번째 방법보다 더욱 효과적인 두번째와 세번째 방법에 대해서 추가로 설명한다.

두번째 방법을 사용하는 기존 연구들로는 Multi-Predicate Merge Join[11], Structural Join[12], Holistic Twig Join[13], Holistic Twig Join의 변형[13-15] 등이 있다. 이 방법들은 오직 인스턴스-레벨 정보만을 가지고 부분 매취 질의를 처리할 수 있다는 장점을 가진다. 그러나, 이 방법은 노드 인스턴스 정보를 비교하는 질의 처리를 해야 하므로, 스키마 정보를 매우 효과적으로 활용하여 노드 인스턴스들을 필터링할 수 있는 스키마 레벨 방법들에 비해 질의 처리 비용이 커지는 경향이 있다. 이들 중 Holistic Twig Join[13]의 대표적 변형인 인덱스를 가진 Holistic Twig Join[15]은 인스턴스-레벨 정보인 엘리먼트 집합들에 대해 인덱스들을 구축하여 트위그 질의(twig query)를 처리하는 인스턴스-레벨 질의 처리 방법이다. 트위그 형태의 질의 처리라는 목적은 제안하는 XIR-Branching과 같으나 질의 처리가 이루어지는 타겟 시스템이 다르고, 질의 처리 방법 및 사용 정보의 레벨 또한 다르다. 먼저, 타겟 시스템이 다르다는 것은 XIR-Branching이 SQL을 확장하여 관계형 데이타베이스에서 사용할 수 있는 질의 처리 방법에 초점을 맞추고 있으나, 기존 연구[15]에서는 전용(native) XML 저장 장치를 타겟 시스템으로 하는 질의 처리 방법에 초점을 맞추고 있다는 것이다. 두 번째로, 질의 처리 방법 및 사용 정보의 레벨이 다르다는 것은 XIR-Branching이 일차적으로 스키마-레벨 정보를 가지고 스키마-레벨 필터링을 거친 후, 이차적으로 축약된 최소의 정보를 대상으로 인스턴스-레벨 질의 처리를 수행하는 방법[2]을 사용하지만, 기존 연구[15]에서는 인스턴스-레벨 정보만을 사용한다는 점이 다르다. 우리가 알기로, 현재까지의 연구 중에 스키마-레벨 정보를 사용하는 방법과 인스턴스-레벨 정보만을 사용하는 방법을 비교한 연구가 아직 제안되지 않았고 두 분야 간에 질의 처리 효율에 관한 정확한 비교를 위해서는 공정한 척도(metric)가 필요하므로 이에 활용한 비교 연구는 또 다른 연구로 남긴다.

1) 경로 표현식 내의 레이블들의 연속을 유한 상태 오토마타(finite state automata) 내의 상태들의 연속으로써 표현한 것.

2) 인덱스를 가진 Holistic Twig Join[19]의 근본을 이루는 구조적 조인(Structural Join) 방법과의 성능 비교 설명은 5.1절 마지막 부분 참조.

세번째 방법을 사용하는 기존 연구들로는 XQuery/IR [16], Keyword Search on XML-QL[17], XRANK[18], Mixed Mode[19] 등이 있다. 역 인덱스들은 인스턴스-레벨 정보에 생성된다. 즉, XQuery/IR[16]과 XRANK [18]는 XML 문서 내에 존재하는 값(value)들에 생성되고, XML-QL 키워드 검색[17]과 Mixed Mode[19]는 값들 뿐만 아니라 노드(즉, element와 attribute)들에 생성된다. 주목할 사항은, 같은 역 인덱스를 사용한다 할지라도 이 방법들은 본 논문에서 제안하는 방법과 근본적으로 다르다. 제안하는 방법은 스키마-레벨 정보인 *label paths*에 대하여 역 인덱스를 생성한다.

3.2 스키마-레벨 방법

스키마-레벨 방법은 레이블 경로들이 어디에 어떻게 저장되는가에 따라, 특별한 목적의 인덱스를 사용하는 방법[20,31]과 관계형 테이블을 사용하는 방법[6-8]으로 구분된다. 전자의 경우, 레이블 경로들은 질의 내에서 그 경로가 사용되는 시점에 동적으로 저장되고 후자의 경우, 문서 내의 모든 레이블 경로들이 질의 처리 이전에 관계형 DBMS 테이블에 저장된다는 차이가 있다.

Index Fabric[20]은 특별한 목적의 인덱스를 사용하는 스키마-레벨 방법 중에 대표적인 방법이다. Index Fabric은 질의에서 빈번하게 발생하는 질의들 내에서 발견되는 레이블 경로들과 값들을 인덱싱하기 위해서 패트리시아 트라이(Patricia trie)를 사용한다. 그러나, Index Fabric은 부분 매칭 질의를 지원하지 않는다. 또한, 이질적인 환경에서의 탐색에 매우 효과적인, BPE를 지원하도록 디자인되지 않았다. 따라서, 이 방법은 대규모 이질 환경에서 사용하기 어렵다는 단점이 있다. 그러므로, 이 절에서는 관계형 테이블을 이용하는 스키마-레벨 방법에 주된 초점을 맞춘다.

XRel[6]과 XParent[7,8]는 XIR-Branching 방법의 기초를 제공하는, 관계형 테이블을 사용하는 스키마-레벨 방법들 가운데 대표적인 두 가지 방법으로, 본 절에서 각 방법을 설명한다. 이때, **엘리먼트**나 **애트리뷰트**는 **노드**(*node*)라는 용어를 사용한다. 이러한 표현은 2.1절의 XML 문서 모델과 2.3절의 XML 질의 패턴에서 **노드**로 표현한 것과 동일한 의미를 가진다.

3.2.1 XRel

XRel에서, XML 트리의 구조 정보는 다음과 같이 네 개의 테이블에 저장된다.

Path(label_path_id, label_path)

Element(document_id, label_path_id, start_position, end_position, sibling_order)

Text(document_id, label_path_id, start_position, end_position, value)

Attribute(document_id, label_path_id, start_position, end_position, value)

Path 테이블은 문서들 내에 있는 모든 레이블 경로들과 그들의 식별자들을 저장한다. **Element** 테이블은 노드들, 즉, 엘리먼트들에 대한 정보를 저장한다. 여기서, 각 노드에 대한 정보는 그 노드를 포함하고 있는 문서의 식별자, 그 노드에서 종료되는 레이블 경로의 식별자, 한 문서 안에 있는 그 노드의 시작과 끝 위치에 대한 옵셋(offset)들, 노드의 형제(sibling)들 가운데 해당 노드의 순번(order) 등으로 구성된다. 시작 위치와 끝 위치의 조합은 노드 식별자를 대신하여 노드를 고유하게 구분하기 위한 정보로 사용된다. **Text** 테이블은 노드의 값들에 대한 정보를 저장한다. 여기서, *value* 컬럼은 텍스트 값들을 저장한다. **Attribute** 테이블은 *value* 컬럼에 텍스트 값 대신 애트리뷰트 값이 저장된다는 것 이외에 **Text** 테이블과 같다. 이들 두 테이블의 정보는 구현에 따라 하나의 테이블에 저장될 수도 있다.

XRel은 LPE와 BPE를 처리하기 위하여 두 가지 기술을 사용한다. 그것은 **스트링 매칭**(*string match*) 와 **포함 관계 조인**(*containment join*) 이다. 전자는 스키마-레벨 방법에 속하고 LPE를 처리하기 위하여 사용된다. 후자는 인스턴스-레벨에 속하고 BPE를 처리하기 위하여 사용된다.

LPE의 경우, XRel은 **Path** 테이블에서 질의의 경로 표현식에 매칭되는 레이블 경로들을 먼저 찾는다. 이때, 매칭 시키는 과정(matching)은 SQL의 스트링 매칭 연산자인 **LIKE** 연산자를 사용한다. 레이블 경로에는 B+-tree 인덱스와 같은 인덱스를 사용할 수 없다. 이는 저장된 레이블 경로에 대해 부분 매칭를 위한 다양한 질의 형태가 주어지기 때문이다. 그러므로, LPE 질의 처리를 위해 **Path** 테이블 내에 있는 레이블 경로들을 모두 읽어야 한다. 결과 노드들을 얻기 위해, XRel은 **Element** 테이블 내의 *label_path_id* 컬럼을 경유하여, 레이블 경로들과 매칭하는 노드들을 조인하여 결과 노드들의 집합을 찾으며, 이것을 **노드 집합**(*node set*) 이라 부른다. XIR-Branching에서는 이 노드 집합이 **노드 경로 집합**(*node path set*) 으로 표현된다.

BPE의 경우, 본 논문에서는 2.3절에서 정의된 질의 패턴을 이용한다. XRel은 먼저, 하나의 BPE를 여러 개의 LPE들로 분할한다. 각 LPE는 루트 노드로 부터 분기하는 노드까지의 LPE와 루트 노드로 부터 단말 노드까지의 LPE로 구성된다. 예를 들어, BPE $/l_1[./l_2/l_3=v_2]/l_4$는 다음 세개의 LPE $/l_1$, $/l_1/l_2/l_3$, $/l_1/l_4$로 분해된다. 이때, 각 LPE에 대해서, XRel은 위에서 설명한 것과 같은 방법으로 LPE로 부터 얻은 노드들의 집합을 찾는다. 이것은 **노드 집합**(*node set*)이라 부른다. 그리고, 선택 조건을 만족하는 노드들로 그 노드 집합을 줄

인다. 이후, 분기하는 노드(예, $/l_1$)에서 끝나는 어떤 LPE로 부터 얻은 노드 집합과 단말 노드들(예를 들어, $/l_1/l_2/l_3$, $/l_1/l_4$)에서 끝나는 LPE들로부터 얻은 노드 집합을 비교한다. 그리고, 후자의 LPE들로 부터 얻은 노드들 가운데, 전자의 노드 집합 내에 있는 노드들의 후손이 되는 노드들만을 취한다. 이것은 노드들의 시작 위치와 종료 위치를 비교하는 θ-join으로 구현된 **포함 관계 조인**(*containment join*)을 사용하여 완료된다.

예제 1. 선형 다음과 같은 BPE를 생각해 보자.

Q2:: //article[./keyword=``XML"]//author[./last =``Curry"]/first

그림 4는 BPE Q2의 질의 패턴을 보여준다. 여기서는, 다음과 같이 다섯개의 LPE들이 생성된다. (1) *//article*, (2) *//article/keyword*, (3) *//article// author*, (4) *//article//author/last*, and (5) *//article// author /first*.

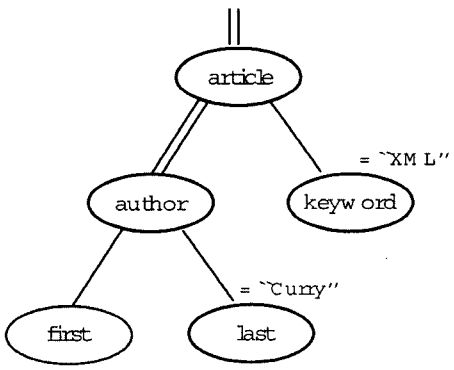


그림 4 BPE Q2의 질의 패턴

그림 5는 이 LPE Q2를 처리하기 위해 생성된 SQL 문이다. 그림 5의 XRel SQL은 먼저, 스트링 매취를 통해서, LPE의 **Path** 튜플들의 집합을 찾는다. **LIKE** 절에서, '#' 심볼은 레이블 구분자이고, '%' 심볼은 한개 이상의 문자들을 매취 시키기 위한 와일드 카드이다. 이때, **Path** 튜플 집합에서, 그 집합 안에 있는 경로의 결과 레이블과 매취하는 엘리먼트의 집합을 찾기 위해, **Element** 테이블이나 **Text** 테이블과 조인을 수행한다. 이때, 어떤 분기 노드(예를 들어, *article*, *author*)들에 해당되는 엘리먼트들의 집합과 그 분리하는 노드의 후손 노드(예를 들어, *author*와 *keyword*, *first*와 *last*)들에 해당되는 엘리먼트들의 집합들 간의 포함 관계 조인을 실시한다.[3] 두 개의 분기하는 노드가 있기 때문에 총 네 번의 포함 관계 조인이 수행된다. 각 포함 관계 조인은 두 개의 θ-join들과 한 개의 equi-join으로 구성되어 수행된다. □

```
SELECT  DISTINCT e5.document_id, e5.start_position, e5.end_position
FROM    Path p1, Path p2, Path p3, Path p4, Path p5,
        Element e1, Element e3, Element e5, Text t2, Text t4
WHERE   p1.label_path LIKE ``#%/article#''
AND     p2.label_path LIKE ``#%/article#/keyword#''
AND     p3.label_path LIKE ``#%/article#%/author#''
AND     p4.label_path LIKE ``#%/article#%/author#/last#''
AND     p5.label_path LIKE ``#%/article#%/author#/first#''
AND     e1.label_path_id = p1.label_path_id
AND     t2.label_path_id = p2.label_path_id
AND     e3.label_path_id = p3.label_path_id
AND     t4.label_path_id = p4.label_path_id
AND     e5.label_path_id = p5.label_path_id
AND     e1.start_position < t2.start_position
AND     e1.end_position  > t2.end_position
AND     e1.document_id = t2.document_id
AND     t2.value = ``XML''
AND     e1.start_position < e3.start_position
AND     e1.end_position  > e3.end_position
AND     e1.document_id = e3.document_id
AND     e3.start_position < t4.start_position
AND     e3.end_position  > t4.end_position
AND     e3.document_id = t4.document_id
AND     t4.value = ``Curry''
AND     e3.start_position < e5.start_position
AND     e3.end_position  > e5.end_position
AND     e3.document_id = e5.document_id;
```

그림 5 BPE Q2를 위한 XRel SQL 문

3.2.2 XParent

XParent[7,8]는 XRel과 유사하지만, θ-join들을 사용하지 않고 equi-join들을 사용하는 포함 관계 조인 연산자를 구현할 수 있도록 하기 위하여 다른 테이블 스키마를 사용한다. XParent의 스키마는 다음과 같다.

LabelPath(label_path_id, length, label_path)
Element(document_id, label_path_id, node_id, sibling_order)
Data(document_id, label_path_id, node_id, sibling_order, value)
Ancestor(node_id, ancestor_node_id, offset_to_ancestor)
DataPath(parent_node_id, child_node_id)

LabelPath 테이블은 XRel의 **Path** 테이블과 같다. **Element**와 **Data** 테이블은 영역 정보(즉, *start_position*, *end_position*) 대신, 노드 식별자를 사용한다는 점을 제외하면 XRel의 **Element**와 **Text** 테이블과 같다. Jiang 등[7,8]의 연구에서는 **Ancestor** 테이블은 노드들 간의 조상-후손 관계성과 그들 간의 레벨 수를 유지하는데 사용하며, 이와 선택적으로 사용할 수 있는 **DataPath** 테이블(소위, **Parent** 테이블)은 노드들 간의 부모-자식 관계성을 유지하는데 사용한다. 그러나, 부분 매취 질의에 대해서는 **DataPath** 테이블의 사용이 적합하지 않으므로 **Ancestor** 테이블을 사용한다[7,8].

질의 처리시 XParent는 XRel과 같은 방법으로 LPE를 처리한다. 그러나, BPE의 경우에 XParent는 질의

3) 이 처리 순서는 선택 조건(selection condition)을 먼저 처리하기 위하여 질의 최적화 과정 동안에 변경될 수 있으며 특히, *value* 컬럼에 대해 인덱스가 있을 때 변경될 수 있다.

패턴의 루트 노드로 부터 각 단말 노드까지의 LPE들만을 만들기 때문에 XRel 보다 더 작은 수의 LPE들이 만들어 진다. 이때, XRel과 같은 방법으로 노드 집합을 검색한 후, 결과 노드를 포함하고 있는 LPE로 부터 얻은 노드 집합은 다른 LPE들로 부터 얻은 노드 집합과 조인하는 과정에서 줄어든다. 이때, 조인은 **Ancestor** 테이블을 통한 한 번의 equi-join으로써 수행함으로 이를 통하여 공통의 조상이 되는 노드의 노드 식별자들을 찾는다.

예제 2. 예제 1 안에 있는 BPE Q2를 생각해 보자. 생성되는 LPE는 다음의 세가지이다. (1) *//article/keyword*, (2) *//article//author/last*, (3) *//article//author/first*. 그림 6은 결과가 되는 SQL 문을 보여준다. 그 LPE들로 부터 얻어진 노드 집합들은 공통의 조상 노드 경로를 공유하는 노드들을 찾기 위하여 **Ancestor** 테이블과 조인된다. 그 밖의 다른 SQL 문은 XRel의 문과 매우 유사하다. □

```
SELECT   DISTINCT e3.document_id, e3.node_id
FROM     LabelPath lp1, LabelPath lp2, LabelPath lp3,
         Data d1, Data d2, Element e3,
         Ancestor an1, Ancestor an2, Ancestor an3
WHERE    lp1.label_path LIKE ``.%/article./keyword.''
AND      lp2.label_path LIKE ``.%/article.%/author./last.''
AND      lp3.label_path LIKE ``.%/article.%/author./first.''
AND      d1.label_path_id = lp1.label_path_id
AND      d2.label_path_id = lp2.label_path_id
AND      e3.label_path_id = lp3.label_path_id
AND      d1.value = ``XML''
AND      d2.value = ``Curry''
AND      d1.node_id = an1.node_id
AND      d2.node_id = an2.node_id
AND      e3.node_id = an3.node_id
AND      an1.offset_to_ancestor = 1
AND      an2.offset_to_ancestor = 1
AND      an1.ancestor_node_id = an2.ancestor_node_id
AND      an2.ancestor_node_id = an3.ancestor_node_id;
```

그림 6 BPE Q2를 위한 XParent SQL 문

3.2.2 XIR-Linear

XIR-Linear[21]는 LPE 만을 처리하기 위한 방법으로 다음과 같이 XML 문서에 대한 정보를 저장하기 위하여 세 개의 테이블과 역 인덱스를 사용한다.

```
LabelPath(label_path_id, label_path)
Element(document_id, label_path_id, node_id,
        sibling_order)
Data(document_id, label_path_id, node_id, sibling_order,
     value)
Inverted Index on label_path of the table LabelPath.
```

그림 7은 그림 2의 XML 트리에 대한 스키마-레벨 정보의 저장 예를 보여준다.

```
SELECT   DISTINCT e3.document_id, e3.node_id
FROM     LabelPath lp1, LabelPath lp2, LabelPath lp3,
         Data d1, Data d2, Element e3,
         Ancestor an1, Ancestor an2, Ancestor an3
WHERE    lp1.label_path LIKE ``.%/article./keyword.''
AND      lp2.label_path LIKE ``.%/article.%/author./last.''
AND      lp3.label_path LIKE ``.%/article.%/author./first.''
AND      d1.label_path_id = lp1.label_path_id
AND      d2.label_path_id = lp2.label_path_id
AND      e3.label_path_id = lp3.label_path_id
AND      d1.value = ``XML''
AND      d2.value = ``Curry''
AND      d1.node_id = an1.node_id
AND      d2.node_id = an2.node_id
AND      e3.node_id = an3.node_id
AND      an1.offset_to_ancestor = 1
AND      an2.offset_to_ancestor = 1
AND      an1.ancestor_node_id = an2.ancestor_node_id
AND      an2.ancestor_node_id = an3.ancestor_node_id;
```

그림 7 LabelPath 테이블과 역 인덱스(Inverted Index)의 예

LabelPath 테이블은 XML 문서들에서 발생하는 중복이 배제된 모든 레이블 경로들과 그들의 경로 식별자(path identifier, pid)를 저장한다. 프리픽스(prefix)로써 '$'와 '&'가 덧 붙여진 레이블들은 각 레이블 경로의 첫 레이블과 마지막 레이블이라는 것을 표시한다. 이러한 덧붙임은 경로 표현식의 루트 레이블과, 결과 레이블에 매칭시키기 위해 필요한 것이다.

LabelPath 역 인덱스 **Inverted Index**는 **LabelPath** 테이블의 *labelpath* 필드에 대해서 구축된다. 이때, XIR-Linear는 각 레이블 경로를 텍스트 문서로 간주하고, 이들 레이블 경로 안에 있는 각 레이블을 키워드로 간주한다. 전통적인 역 인덱스[22]와 같이, **LabelPath** 역 인덱스 구조는 키워드(즉, 레이블)와 포스팅 리스트의 쌍들로 구성된다. 포스팅 리스트 내의 각 포스팅은 *pid*, *occurrence_count*, *offsets*, *label_path_length*와 같은 필드를 가진다. 여기서, *pid*는 레이블이 존재하는 레이블 경로의 식별자이고, *occurrence_count*는 레이블 경로 내에서 그 레이블이 나타난 개수이며, *offsets*는 레이블 경로가 시작하는 위치로 부터 해당 레이블들의 위치까지 존재하는 레이블 수(label들의 개수)들의 집합(즉, 옵셋의 집합. 레이블 하나를 1로 간주)이다. *label_path_length*는 레이블 경로 내에 존재하는 레이블들의 개수이다. 예를 들어, 레이블 경로 *$chapter.chapter.section.section.section.paragraph.&paragraph*에서, *section*의 *occurrence_count*, *section*의 *offsets*, *label_path_length*는 각각 3, {3, 4, 5}, 7이다.

XIR-Linear는 XML 문서들 내에 있는 모든 노드들을 고유하게 식별하기 위하여 인스턴스-레벨 정보를 유지한다. XIR-Linear에서는 설명의 편의를 위하여 XParent의 인스턴스-레벨 저장 구조인 **Element**와 **Data** 테이블 구조를 XIR-Linear의 인스턴스-레벨 저장 구조로 사용한다[21].

XIR-Linear의 질의 처리 방법은 다음과 같다. 그림 8은 LPE를 처리하는 알고리즘이다. 먼저, **LabelPath** 역 인덱스를 사용하여 **LabelPath** 테이블 내에 매취하는 레이블 경로를 찾고, 이로서 얻어진 레이블 경로들의 집합과 **Element** 테이블 간에 **pid** 컬럼을 이용하여 equi-join을 수행한다. 이때, 매취하는 노드들이 질의의 결과가 된다. 이 LPE 처리 과정을 자세히 설명하면 다음과 같다. 먼저, 주어진 입력 LPE ***P***를 IR Expression ***E***로 변환한다. 이 변환은 향후 설명할 **Rule 1**을 따른다. 이후, ***E***에 속한 각 키워드(keyword)를 가지고 그림 7(b)의 LabelPath Inverted Index를 검색하여 그 키워드에 대한 포스팅들을 구한다. 이때, **Rule 1**을 통해 변환된 ***E***의 근접 연산자를 이들 포스팅들 간에 적용하여 그 연산 결과를 만족하는 ***pid***들을 구한다. 이 결과 ***pid***들은 그림 7(a)의 ***pid***들이 된다. 마지막으로, 이 ***pid***를 가지고, 그림 10에서 설명하는 **NodePath** 테이블과 조인하여 최종적으로 노드 경로를 구할 수 있다. XIR-Linear 방법은 기존 방법의 LPE 처리 부분을 역 인덱스 검색으로 대치한 방법으로, 비용이 큰 스트링 매취를 사용하지 않는다는 장점이 있다. 그러나, XIR-Linear는 관계형 데이타베이스 시스템에 정보 검색 모듈을 추가하여야 한다. 이는 *off-the-shelf DBMS*를 사용하는 XRel나 XParent에 비해 단점이 될 수 있으나, 좋은 질의 처리 성능을 위해서 정보 검색 모듈을 추가하여 사용할 수 있다.

Algorithm XIR_LPE_evaluation
Input: LPE *P*, LabelPath inverted index, NodePath table
Output: set of node paths *NPset* matching the LPE
begin
1. Convert the input LPE *P* to an IR expression *E* using the syntactic mapping rule LPE-to-IRExp (Rule 1).
2. Find the set of pids (*pidSet*) using the LabelPath inverted index given the IR expression *E*.
3. Find the set of node paths (*NPset*) through an equi-join between *pidSet* and the NodePath table.
4. Return *NPset*.
end

그림 8 XIR-Linear LPE 처리 알고리즘

LPE의 처리를 정형적으로 표현하면 식 (1)과 같다.

$$\Pi\ nodepath(\ \sigma_{MATCH(labelpath,\ LPE)}\ \text{LabelPath} \bowtie_{pid=pid} \text{Element}) \quad (1)$$

식 (1)의 선택문 $\sigma_{MATCH(labelpath,\ LPE)}$ **LabelPath**는 labelpath 컬럼에 대한 텍스트 검색으로서 구현되기 때문에 먼저, LPE를 키워드 기반의 텍스트 검색 조건으로 변환해야 한다. 본 논문에서는 이 검색 조건을 **정보 검색 표현식**(*information retrieval expression, IRExp*)이라 한다. 다음 ***Rule 1***은 이러한 변환을 완성하는 방법을 명시하고 있다.

Rule 1 LPE $o_1\ l_1\ o_2\ l_2\ \ldots\ o_p\ l_p$(여기서 $i=1,2,\ldots,p$)에 대한 $o_i \in \{\ '/',\ '//'\ \}$는 다음 rule을 사용하여 IRExp로 변환된다.

$$o_1 l_1 \Longrightarrow \begin{cases} l_1 & \text{if } o_1 = \text{`//'} \\ \$l_1\ near(1)\ l_1 & \text{if } o_1 = \text{`/'} \end{cases}$$

$$l_i o_{i+1} l_{i+1} \Longrightarrow \begin{cases} l_i\ near(\infty)\ l_{i+1} & \text{if } o_{i+1} = \text{`//'} \\ l_i\ near(1)\ l_{i+1} & \text{if } o_{i+1} = \text{`/'} \end{cases} \quad \text{for } i = 1, 2, \ldots, p-1$$

$$l_p \Longrightarrow l_p\ near(1)\ \&l_p$$

여기서, *near(w)*는 최대 *w*개의 키워드들만큼 떨어진 두 개의 대상 키워드(operand keyword)들을 가진 문서들을 검색하기 위해 사용되는 **근접 연산자**(*proximity operator*)로 정보 검색의 근접 연산 개념[22]을 구현한 연산자이다[25,32]. □

이때, l_1과 l_p는 각각 LPE를 표현하는 선형 질의 패턴의 루트 노드(즉, 첫 노드)와 결과 노드(즉, 마지막 노드)가 됨에 주목해야 한다. 예를 들어, LPE *//article//author/last*는 IRExp *article near(∞) author near(1) last near(1) &last*으로 변환된다. 또한, LPE */issue/articles//author*는 IRExp *$issue near(1) issue near(1) articles near(∞) author near(1) &author*로 변환된다. 이때, *$issue*는 *issue*가 그 문서의 루트 노드임을 나타내기 위해 사용된다.

예제 3. LPE *//article//author/first* 를 생각해 보자. XIR-Linear는 이 LPE에 LPE-to-IRExp 규칙을 적용하여, IRExp인 *article near(∞) author near(1) first near(1) &first*로 변환한다. 그리고, 그림 7(b)의 **LabelPath 역 인덱스**에 대한 검색을 통해 *pidSet* {10}를 반환하고, 이 집합을 **Element** 테이블과 조인하여 *NPset* {19, 31}을 반환한다. 이러한 질의 처리 결과는 그림 2를 통해 검증될 수 있다. 그림 9는 LPE *//article//author/first*를 처리하기 위해 생성된 XIR-Linear SQL 문을 보인다. WHERE 절에 있는 MATCH는 LabelPath 역 인덱스를 사용하게 하는 연산자이다. MAXINT 심볼은 시스템에 정의된 최대 정수 값이다. □

```
SELECT  DISTINCT n1.docid, n1.nodepath
FROM    LabelPath p1, NodePath n1
WHERE   p1.pid = n1.pid
AND     MATCH(p1.labelpath, `article' NEAR(MAXINT) `author'
                            NEAR(1) `first' NEAR(1) `&first');
```

그림 9 LPE //article//author/first를 위한 SQL 문

XIR-Linear의 LPE 처리 알고리즘은 ***Rule 1***을 통해 주어진 LPE를 **IRExp**로 변환하고, 이를 통해 그림 7(b)에서 보인 역 인덱스를 검색함으로써(XML 구조 정보인) 레이블 경로를 빠르게 찾을 수 있는 방법으로 매우 우수한 선형 질의 처리 성능을 얻을 수 있어 스키마-레벨 정보를 빠르게 찾을 수 있는 장점을 가진다. 본 논문에서는 XIR-Linear의 장점인 스키마-레벨 정보 처리 방법을 기반으로 하면서 분기 경로 표현식(BPE)을 빠르게 처리할 수 있는 방법으로 XIR-Branching을 제안한다.

4. XML 저장 구조

본 장에서는 XIR-Branching에서 사용하는 XML 문서의 저장 구조를 제안한다. XML 문서는 스키마-레벨과 인스턴스-레벨 정보로 분리되어, 질의 처리에 필요한 정보의 형태로 관계형 데이타베이스 테이블에 저장된다. XIR-Branching은 **스키마-레벨 정보**를 저장하는 테이블로 XIR-Linear에서 제안한 **LabelPath** 테이블과 **LabelPath** 역 인덱스(inverted index)를 변형 없이 사용한다. 그러나, 노드 인스턴스의 저장 구조는 XIR-Linear와 다르다. XIR-Branching은 노드 인스턴스 저장 구조로서 XIR-Linear에서 사용하는 **Element**와 **Data** 테이블 구조를 제거하고, 새로운 저장 구조인 **NodePath** 테이블을 사용한다. 이 테이블에는 XML 트리 내에 있는 모든 **인스턴스-레벨 정보**를 레이블 경로와 매취되는 노드의 경로 형식으로 *nodepath* 컬럼에 저장한다. 다음은 XIR-Branching이 XML 문서 구조에 관한 정보를 저장하기 위하여 사용하는 두 테이블과 역 인덱스이다.

LabelPath(pid, label_path)
NodePath(pid, docid, nodepath, value)
Inverted index: LabelPath 테이블의 *label_path*에 대한 역 인덱스.

XIR-Branching 방법은 스키마-레벨 정보 저장 시 XIR-Linear 방법을 사용하므로, 스키마-레벨 정보의 저장 예는 3.2.3절의 그림 7에서 볼 수 있다.

인스턴스-레벨 정보는 XML 문서들 내에 있는 모든 노드들을 고유하게 식별할 필요가 있으며, 이러한 정보는 **NodePath** 테이블에 저장된다. 그림 10은 그림 2의 XML 트리에 대한 인스턴스-레벨 정보의 저장 예를 보여준다.

그림 10의 **NodePath** 테이블은 XML 문서의 모든 노드 경로들을 *nodepath* 컬럼에 저장한다. 만약 노드 경로의 단말 노드가 값(value)을 가지고 있다면, 그 값은 *value* 컬럼에 저장한다. *pid* 컬럼은 동일한 레이블

pid	docid	nodepath	value
1	1	1	Null
2	1	1.2	Null
3	1	1.2.3	Michael
4	1	1.2.5	Flanklin
2	1	1.7	Null
3	1	1.7.8	Jane
4	1	1.7.10	Poe
5	1	1.12	Null
6	1	1.12.13	Null
7	1	1.12.13.14	R
8	1	1.12.13.16	XML schema
9	1	1.12.13.18	Null
10	1	1.12.13.18.19	David
11	1	1.12.13.18.21	Curry
12.	1.	1.12.13.23	XML
6	1	1.12.25	Null
...	...	...	...
12	1	1.12.25.35	DB

그림 10 NodePath 테이블의 예

경로에 속하는 모든 노드 경로들을 찾을 때, **LabelPath** 테이블과 조인하기 위해 사용될 레이블 경로 식별자들을 저장한다. *docid* 컬럼은 XML 문서 식별자들을 저장한다.

5. XIR 질의 처리 알고리즘

본 절은 XIR-Branching 저장 구조를 기반으로 BPE를 처리하기 위한 알고리즘을 제안하고, 성능과 관련된 항목들에 초점을 맞추면서 XRel, XParent, XIR-Branching의 BPE 처리 방법을 분석적으로 비교한다. 이때, BPE는 LPE의 포함 집합이므로, XIR-Branching은 XIR-Linear[21]의 LPE 처리 알고리즘을 기반으로 한다.

5.1 BPE 처리 알고리즘

그림 11은 BPE를 처리하기 위한 알고리즘을 보인다. 이 알고리즘은 XParent와 같은 방법으로 BPE를 LPE들로 분할한다. 즉, 이러한 BPE의 분할은 루트 노드로부터 각 단말 노드까지를 각각의 LPE로 하여 분할하는 것이다. 이후, 노드 경로 집합들(*NPset*들) 중 하나의 집합을 얻기 위해 이들 각 LPE를 연산한다. 이 연산은 식 (1)에서와 같은 방법으로 완료되지만, 분기 조건 표현식을 처리하기 위하여 식 (2)와 같이 약간의 수정을 거친다.

$$\Pi\ nodepath(\ \sigma_{MATCH(labelpath,\ LPE)}\ \textbf{LabelPath} \bowtie_{pid=pid} \sigma_{value\ \theta\ v}\ \textbf{NodePath}) \quad (2)$$

여기서, "***value θ v***"는 처리될 LPE 내에 포함된(정

의 4에서 정의된) 분기 조건 표현식의 단말 레이블에 대한 선택 조건(selection condition)이다.

```
Algorithm XIR_BPE_evaluation
Input: BPE P, LabelPath inverted index, NodePath table
Output: set of node paths result_NPset matching the BPE
begin
   { Assume the BPE P has r leaf labels l0, l1, l2, ..., lr-1 and that l0 is
     the result label. }
1.  Set P0 as the LPE from the root to l0.
2.  Obtain result_NPset by evaluating the LPE P0.
3.  for each of the remaining leaf nodes li (i=1,2, ..., r-1)
4.  begin
5.      Set Pi as the LPE from the root to li.
6.      Obtain NPseti by evaluating the LPE Pi.
7.      Reduce result_NPset through a prefix match join
        with NPseti.
8.  end.
9   return result_NPset.
end.
```

그림 11 XIR-Branching BPE 처리 알고리즘

LPE들 중 하나 만이(2.3절에서 정의된) 결과 노드를 포함한다. 그러한 LPE(그림 11의 p_0)는 **결과 LPE** (*result LPE*) 라 한다. 이때, XIR-Branching은 그림 11 내의 다른 LPE들 각각을 가지고 **프리픽스 매칭 조인**(*prefix match join*) 을 수행하여, 결과 LPE로 부터 얻은 *result_NPset*을 줄인다. 정의 8은 프리픽스 매칭 조인의 정형적인 정의를 보인다.

정의 8. 애트리뷰트 A_1A_2 ... A_c들을 공유하는 스키마 $R(A_1A_2$ ... A_cB_1 ... $B_m)$과 $S(A_1A_2$ ... A_cC_1 ... $C_k)$을 가지는 두 릴레이션들에 대해서, R과 S 간의 **프리픽스 매칭 조인**(*prefix match join*)은 R ▷ S로 표현되고,다음과 같이 정의된다.

$$R \triangleright S = \sigma_{R.A1=S.A1 \text{ and } \ldots \text{ and } R.Ac=S.Ac}(R \times S) \quad \square$$

정의 8에서, 그 릴레이션 스키마는 레이블 경로를 말하고, 릴레이션 인스턴스는 노드 경로 집합을 말한다. 이 정의에 따라, 주어진 두 LPE들로부터 얻는 두 개의 *NPset*들 간의 프리픽스 매칭 조인은 다음과 같이 수행된다.(1) 양쪽 LPE들을 매칭하는 정의 8의 가장 긴 공통의 프리픽스 레이블 서브-경로 l_1l_2 ... l_c(= A_1A_2 ... A_c)를 찾는다.(2) 그 레이블 서브-경로 l_1l_2 ... l_c에 속하는 공통의 프리픽스 노드 서브-경로들 {n_1n_2 ... n_c}의 집합을 찾는다. 그리고, (3) 그 집합 안의 각 노드 서브-경로 n_1n_2 ... n_c에 대해서, 공통의 서브-경로를 가진 모든 노드 경로들을 선택한다.

예제 4. 예제 1에서 사용된 BPE Q2를 생각해 보자. 생성되는 세 개의 LPE들은 (1) *//article/keyword*, (2) *//article//author/last*, (3) *//article//author/first*이다. 이들 가운데, LPE(3)은 결과 LPE이다. XIR-Branching은 이들 LPE들과 분기 조건 표현식들로 부터 다음 세 개의 노드 경로 집합들을 검색한다. 첫째는 LPE(1)과 *keyword* = *"XML"*로부터 $NPset_1$ {1.12.13.23}를 검색한다. 둘째는, LPE(2)와 *last*="*Curry*"로 부터 $NPset_2$ {1.12.13.18.21}를 검색한다. 마지막으로, LPE (3)로 부터 $NPset_3$ {1.12.13.18.19, 1.12.25.30.31}를 검색한다. 이때, $NPset_2$와 함께 진행되는 프리픽스 매칭 조인은 공통의 프리픽스 레이블 서브-경로 */issue/ articles/article/ author*에 기반하는 {1.12.13.18.19}으로 $NPset_3$를 줄인다. 그리고, $NPset_1$과의 추가적인 조인은 공통의 프리픽스 레이블 서브-경로 */issue/articles/ article*에 기반하는 {1.12.13.18.19}가 될 $NPset_3$를 유지한다.

그림 12는 BPE Q2에 대해 만들어진 SQL 문을 보인다. 그것은 각 튜플 씩 프리픽스 매칭을 수행함으로, *XIR_BPE_evaluation* 알고리즘에서 보여준 노드 집합들의 프리픽스 매칭 조인을 구현한다. *Prefix_matching* 함수는 처음 두 입력 인수들(예, *p1.nodepath*와 *p2.nodepath*)로서 제공된 두 노드 경로들 간에 *getCommonPrefixLength* 함수로 부터 반환된 길이 만큼의 프리픽스를 비교함으로써 프리픽스 매칭를 수행한다. *getCommonPrefixLength* 함수는 두 개의 LPE와, 그들을 매칭하는 두 레이블 경로들을 입력으로 받아 다음과 같은 과정을 거쳐 프리픽스의 길이를 계산한다. 그 과정은 (1) 두 개의 입력 LPE가 가진 가장 긴 공통의 서브-표현식을 밝힌다. 예를 들어, 그림 7에서 *//article/keyword*와 *//article/author/last*의 공통은 *//article*이다. (2) 두 개의 입력 레이블 경로, 즉, *p1.labelpath*와 *p2.labelpath* 각각에 대해 공통의 프리픽스 서브-표현식을 매칭하는 프리픽스 레이블들의 수를 센다. 예를 들어, 그림 7에서, 레이블 경로인 $issue.*issue.articles.article*.keyword.&keyword는 3으로 센다. 이때, 레이블 경로에 부수적으로 추가된 *$issue*는 배제된다. 그리고, (3) 만약, 그 두 숫자가 같으면, 그 *count*를 반환하고, 그렇지 않으면, -1을 반환한다. □

```
Algorithm XIR_BPE_evaluation
Input: BPE P, LabelPath inverted index, NodePath table
Output: set of node paths result_NPset matching the BPE
begin
   { Assume the BPE P has r leaf labels l0, l1, l2, ..., lr-1 and that l0 is
     the result label. }
1.  Set P0 as the LPE from the root to l0.
2.  Obtain result_NPset by evaluating the LPE P0.
3.  for each of the remaining leaf nodes li (i=1,2, ..., r-1)
4.  begin
5.      Set Pi as the LPE from the root to li.
6.      Obtain NPseti by evaluating the LPE Pi.
7.      Reduce result_NPset through a prefix match join
        with NPseti.
8.  end.
9   return result_NPset.
end.
```

그림 12 XIR-Branching BPE Q2를 위한 SQL 문

그림 13은 그림 11에서 소개된 XIR-Branching BPE 처리 알고리즘에 대한 구체적인 과정을 기술한 것이다.

그림 13의 각 스텝을 설명하면 다음과 같다. 라인 3~4는 BPE를 LPE들로 분할하는 과정으로 그림 11의 라인 1과 5를 표현한 것이다. 라인 5~9는 각 LPE를 통하여 경로 식별자(그림 7(a)의 *pid*)들을 알아내기 위한 과정이다. 이때, 라인 6의 *GenerateIRExpr*은 3장에서 소개된 ***Rule 1***을 구현한 함수이며 라인 7~8은 LPE가 변환된 *IRExpr*을 가지고 그림 7(b)에서 소개된 **Label-Path Inverted Index**를 검색하여 *IRExpr*을 만족하는 경로 식별자들을 모두 구하는 과정이다. 이 과정은 그림 12에서 소개된 SQL *WHERE*절의 *MATCH* 함수에 대응된다. 라인 10~11은 라인 5~9에서 구해진 각 경로 식별자들을 가지고, 각각 **NodePath** 테이블과 조인하여 해당 경로 식별자에 대한 노드 경로들을 구하는 과정이다. 이 과정은 그림 11의 라인 2와 6을 표현한 것이며, 그림 12의 SQL *WHERE*절의 첫 세줄에 대응된다. 라인 12는 결과 노드 경로 집합을 초기 설정한 것이다. 결과 노드 경로 집합은 라인 13~16의 프리픽스 매취 조인을 통하여 조건에 부합하는 노드 경로만 남게 되므로 최종 결과 집합은 라인 12의 초기 결과 집합에 비해 같거나 축소되게 된다. 그림 13의 라인 13~16은 결과 집합에 존재하는 노드 경로들의 집합과 나머지 경로 집합들 간에 실시되는 프리픽스 매취 조인 과정이다. 이 과정은 그림 11의 라인 7을 표현한 것이며, 이것은 그림 12의 SQL *WHERE*절에 있는 마지막 두 AND 조건에 대응된다. 마지막으로 라인 17에서 최종 노드 경로 집합을 얻는다.

예제 5. 그림 14는 BPE *//issue//editor*[*./last="Kim"*]*/first*를 처리하는 과정에서 프리픽스 매취 조인의 사용 예를 보인다. 그림은 다음과 같이 구성되어 있다. 그림 14(a)는 질의 대상 XML 문서를 트리로 표현한 것이다. 이때, "*" 기호는 질의 결과가 되는 엘리먼트를 표현한 것이다. 그림 14(b)는 주어진 BPE에 대한 질의 패턴이다. 이때, *//issue//editor*까지는 분할될 LPE들 공통의 구조적 조상이 된다. 그림 14(c)는 주어진 BPE를 두 개의 LPE들로 분할한 것으로, 생성되는 두 개의 LPE들은 (1) *//issue/editor/first*, (2) *//issue//editor/last="Kim"*이다. 이들 가운데, LPE(1)은 결과 LPE이다. 이 과정은 LPE 질의 처리가 수행되는 단계

```
XIRQueryProcessing(Twig pattern p)
{
1.   IRExpression e;
2.   Result = {};
3.   decompose p into k linear twig patterns p0, p1, p2, ... , pk-1
4.            by traversing the twig pattern p in the postorder;
5.   for (i = 0; i < k; i++) {
6.          e = GenerateIRExp(pi);
7.          find a set of label paths, LabelPathSeti,
8.               by executing the IR expression e on LabelPathInvertedIndex;
9.   }
10.  for (i = 0; i < k; i++)
11.       NodePathSeti = JOIN (LabelPathSeti, NodePathTable);
12.  Result = NodePathSet0;
13.  for (i = 1; i < k; i++)
14.       Result = PrefixMatchJoin (Result, NodePathSeti,
15.                 Maximum_Common_Prefix_Length(Result.labelPath, NodePathSeti.labelPath,
16.                                              Common_linear_twig_pattern));
17.  return Result;
}
```

그림 13 XIR-Branching 질의 처리

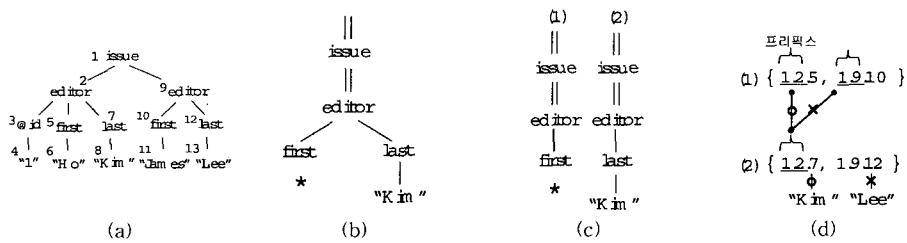


그림 14 프리픽스 매취 조인 처리 예

로, 각 LPE에 대한 노드 경로 집합들을 구한다. 마지막으로 그림 14(d)는 두 집합 씩 쌍을 지어 프리픽스 매취 조인을 실시하는 단계로 과정(c)에서 구한 노드 경로 집합들에 대해 루트로 부터 분기하는 노드(branching node)까지의 공통의 인스턴스 노드 조상(common instance node prefix)을 가지고 주어진 조건을 만족하는 노드 경로들을 구한다. 그림 14(d)는 (c)의 LPE들에 부합되는 노드 경로들의 집합들을 프리픽스 매취 조인하여 결과 노드 경로 집합인(1) 중에서 "1.2.5"만을 결과로 얻게 되어 노드 경로의 마지막에 있는 노드인 "5"가 결과 노드임을 보인 것이다. 이때, "1.2", "1.9" 등은(b)의 구조적 조상에 대응되는 인스턴스 조상들이다. □

프리픽스 매취 조인은 기본적으로 스트링 매취를 사용한다. XRel과 XParent에서는 스트링 매취 연산을 스키마-레벨 정보를 찾기 위해 사용하였으나, 본 연구에서는 인스턴스-레벨 정보를 찾기 위해서 스트링 매취를 사용한다. 인스턴스-레벨 정보 처리 시 사용하는 스트링 매취는 XRel이나 XParent에서 사용한 스트링 매취와는 그 역할이 다르고 그 역할에 따른 처리 비용 역시 다르다. 먼저, 프리픽스 매취 조인에서 사용하는 스트링 매취는 노드 경로의 구조적 조인(structural join) 역할을 대신할 수 있는 방안이다. 즉, 스트링 매취가 구조적 조인 비용에 비해 싸다는 것이다. XIR-Branching은 **노드 경로**라는 형태로 인스턴스-레벨 정보를 저장한다. 이 경우 값 비싼 포함 관계 연산을 실시하는 구조적 조인 대신 간단한 스트링 매취로 인스턴스-레벨 질의 처리가 가능하다. 이것은 노드 경로가 가지는 특성 때문이다. 노드 경로는 XML 트리의 루트 노드로부터 임의의 노드까지 노드 식별자들의 시퀀스로 구성되어 있다. 이것은 노드 경로가 한 노드의 조상 노드 식별자들의 시퀀스를 경로 내에 유지하고 있어서 노드 경로의 구조 자체만 가지고도 이미 구조적 조인(structural join)이 완료된 형태를 갖추고 있으므로 조상을 의미하는 프리픽스만을 비교해도 동일 조상임을 밝혀낼 수 있기 때문이다. 즉, 노드 경로의 구조는 미리 조인된(pre-joined) 형태를 지니고 있기 때문에 구조적 조인에서와 같이 정렬(혹은 인덱싱)을 수행한 후, 동적으로 노드들 간의 조상-후손 관계를 밝히는 값 비싼 포함 관계 연산을 수행할 필요가 없다. 이는 질의 처리 이전인 XML 문서의 파싱 단계에서 포함 관계 연산이 내포된 정적인 노드 경로의 형태로 **NodePath** 테이블에 저장해 놓았기 때문이다. 그러므로 프리픽스 매취 조인은 간단한 스트링 매취를 가지고 값비싼 구조적 조인 비용을 대신할 수 있는 것이다.

요약하면, XIR-Branching은 스키마-레벨 정보 처리 시 스트링 매취 대신 정보 검색 방법을 사용하여 스키마-레벨 질의 처리 성능을 향상시키고, 인스턴스-레벨 정보 처리 시 구조적 조인 대신 새로운 프리픽스 매취 조인을 사용하여 인스턴스-레벨 질의 처리 성능을 향상시켰다. XIR-Branching은 스키마-레벨에서 일차적으로 필터링된 인스턴스-레벨 정보만이 이차적인 조인에 참여하므로 그 조인 대상이 매우 작아지게 되므로 매우 효과적으로 질의를 처리할 수 있다. 그러나, 프리픽스 매취 조인 자체의 성능을 더욱 향상시키기 위하여 스트링 매취 대신 새로운 자료 구조를 활용한 인스턴스-레벨 정보 처리 방안을 향후 연구로 진행하고 있다.

5.2 XRel 및 XParent와의 BPE 질의 처리 방법 비교

세 방법 XRel, XParent, XIR-Branching은 유사한 골격을 가지고 있으나, 구현 방법이 상이하여, 성능에 큰 차이가 있다. 그림 15는 XRel, XParent와 제안된 XIR-Branching의 구현 방법과 성능을 비교한 표이다.

성능 관련 요소들	XRel	XParent	XIR-Branching
레이블 경로 매치 방법	string match	string match	inverted index search
분기 조건들을 가진 BPE에 대한 LPE들의 개수	*2n* or *2n+1*	*n+1*	*n+1*
엘리먼트 식별 방법	interval	node id	node id sequence
공통의 조상들을 가진 인스턴스를 찾는 조인 방법	containment join	containment join	prefix match join
공통의 조상들을 가진 인스턴스를 찾는 조인의 수	*(2n-1 또는 2n)* × (2 *Θ-joins* + *1 equi-join)*	*(n+1)* + *n equi-joins*	*n prefix match joins*
길이 *p*의 node path를 저장하기 위해 필요한 튜플의 수	*1* (Element 나 Text 테이블)	*1* (Element 나 Data 테이블) *p-1* (Ancestor 테이블)	*1* (NodePath 테이블)

그림 15 XParent, XIR-Branching 간의 비교

BPE 질의 처리의 경우 세가지 방법은 공히 다음과 같은 세가지 과정을 거친다. 첫째, 주어진 BPE를 LPE들로 분할한다.[4] 둘째, 분할된 각 LPE들을 처리하여 각 LPE에 속하는 노드 인스턴스들을 구한다. 세째, 이들 인스턴스들의 그룹 간 조인을 통하여 결과 노드 인스턴스들을 구한다. 그러나, XIR-Branching은 XRel, XParent와 비교해 볼때, 첫번째의 나누는 방법이 XRel과 다르고, 두번째의 LPE를 처리하는 방법이 XRel, XParent와 다르며, 세번의 조인 방법 역시 XRel, XParent와 다르다. 기존 연구인 XIR-Linear[21]는 세가지 과정 중, 두번째 단계에서 LPE를 값 비싼 스트링 매취(string match)로 처리하지 않고, 역 인덱스(inverted index)를 사용하여 처리하는 방법을 소개하고 있으므로, 본 논문에서는 BPE와 관련이 있는 첫번째와 세번째 단계에 대해서만 분석한다.

첫번째 단계를 통한 성능 향상의 근거를 살펴 보면, XIR-Branching은 주어진 BPE에 대하여 XRel의 경우보다 상대적으로 작은 수의 LPE들($n+1$개)을 만들어 낸다. 구체적으로, BPE $/l_1[C_1]/l_2[C_2]/ \ldots /l_k[C_k]$를 생각해 보자(예, 정의 5). 3.2.1절에서 언급한 것처럼, XRel은 질의 패턴의 루트 노드로 부터 단말 노드까지의 각 경로를 하나의 LPE로 만들고, 추가적으로, 루트 노드로 부터 분기하는 노드까지의 각 경로에 대해 LPE를 만든다. 그러므로, 만약, 질의 패턴에 $n(\leq k)$ 개의 분기하는 노드들이 있다면, BPE는 C_k가 생략된 경우에 $2n+1$개의 LPE를, 그렇지 않다면, $2n$개의 LPE를 만든다. 반면, 3.2.2절과 5.1절에서 언급한 것처럼, XParent와 XIR-Branching은 분기하는 노드까지의 LPE를 고려할 필요가 없다. 그러므로, 그들은 단지 $n+1$개의 LPE만을 만든다.

세번째 단계를 통한 성능 향상의 근거를 살펴 보면, XIR-Branching은 질의 결과로서 반환되는 최종 결과 노드 집합(노드 경로 집합)을 얻기 위해 수행되는 조인에 잇점이 있다. 특히, 이러한 이점은 수행되는 조인들의 수와 조인된 노드 집합들의 카디널리티(cardinality)들 때문이다. 각 방법에 대한 조인의 수는 다음과 같이 결정된다. XIR-Branching의 경우, $n+1$개의 LPE들로 부터 얻어진 $n+1$개의 노드 경로들의 집합들 사이에 n개의 프리픽스 매취 조인들이 수행된다. XRel의 경우, $2n$(혹은, $2n+1$)개의 LPE들로 부터 얻어진 $2n$(혹은, $2n+1$)개의 노드 집합들 사이에 $2n-1$(혹은, $2n$)개의 포함 관계 조인들이 수행된다. XRel에서의 한개의 포함 관계 조인은 두 개의 θ-*join*과 한개의 *equi-join*을 포함하기 때문에, 실질적인 조인들은 $4n-2$개의 θ-*join*과 $2n-1$개의 *equi-join*(혹은, $4n$ θ-*join*과 $2n$개의 *equi-join*)을 포함하게 된다. XParent의 경우, 먼저 $n+1$개의 LPE들의 하나로 부터 얻은 노드 집합과 **Ancestor** 테이블간에 각각 $n+1$개의 equi-join을 수행한다. 이것은 조상 노드 집합을 찾기 위함이고, 이것은 $n+1$ 조상 노드 집합들 사이에 n개의 포함 관계 연산을 통해 밝혀진다. 여기서, 포함 관계 조인은 **Ancestor** 테이블 내에서 *self*-equi-join으로써 구현된다.

XIR-Branching이나 XRel에서 조인된 노드 집합들의 카디널리티들은 XParent의 카디널리티보다 작다. 특히, 길이가 p인 *nodepath*를 저장하기 위한 XIR-Branching의 **NodePath** 테이블에 존재하는 단 한개의 튜플은 $p-1$개의 조상-후손 쌍들을 저장하고 있는 XParent의 **Ancestor** 테이블에 존재하는 $p-1$개의 튜플들과 동일한 정보의 양을 가진다. 반면, XRel에서는 **Element**나 **Text** 둘 중 한 테이블에서 노드 경로의 마지막 노드만을 저장하면 되고, 그 노드들의 구간 정보를 가지고 결합하면 된다. 그러므로, 조인될 노드 집합들의 카디널리티는 XIR-Branching과 같다. 이때, XParent와 XRel을 비교해 보면, 조인된 노드 집합들의 카디널리티들은 XParent가 많은 반면, 수행된 조인의 수는 XRel이 많다는 것에 주목해야 한다. 따라서, 어느 것이 조인 비용에 더 영향을 주느냐에 따라, 그들의 상대적인 성능이 변할 수 있다.

6. 성능 평가

본 장은 XIR-Branching의 질의 처리 성능을 XRel 및 XParent와 두 종류의 실험을 통하여 비교한다. 첫번째는 데이타베이스 내에 있는 서로 다른 레이블 경로의 수를 기준으로 이들 방법의 **효율성**(*efficiency*)를 비교한다. 두번째는 매우 큰 데이타베이스를 수용할 능력이 있는지에 대해 살펴보기 위해 **확장성**(*scalability*)을 비교한다. 실험 결과는 XIR-Branching이 XRel이나 XParent에 비해 보다 큰 효율성과 확장성을 가지고 있음을 보인다.

6.1 실험 설정

6.1.1 데이타베이스

본 실험에서는 두 개의 웹 크로울러(web crawler) Teleport Pro Version 1.29.1959[33]와 ReGet Deluxe 3.3 Beta(build 173)[34]을 사용하여 인터넷으로 부터 10008개의 XML 문서를 수집하였다. XML 문서들을 수집하기 위해 먼저, 기본 주소(base URL)들을 가지고 수집을 시작하고, 이어서 그 기본 주소들로 부터 도달할 수 있는 모든 XML 문서들을 수집하는 방법을 사용하였다. 그 기본 주소들은 여러 나라에 있는 주요 대학들,

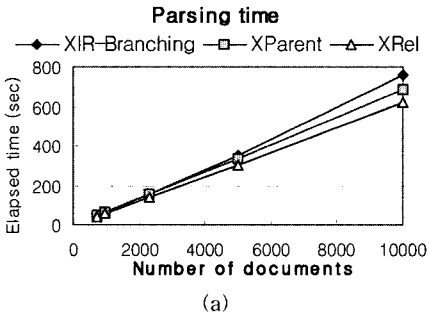
4) XIR-Linear[21]에서는 기 분할된 하나의 LPE만이 처리 대상이기 때문에 이러한 분할 과정이 없다.

회사들, 출판사들의 웹 싸이트를 포함한다. 수집된 XML 문서의 91%에 해당하는 문서들은 그 크기가 4 Kbyte 이하이다.

수집된 XML 문서들은 크기에 따라 서로 구별되는 다섯 개의 데이타 화일 집합으로 나뉘어 저장되었다. 각 집합은 서로 다른 레이블 경로(distinct label path)들을 기준으로 대략 5000, 10000, 20000, 40000, 80000개를 포함하고 있다. 이들 중 마지막 집합은 1460000개의 노드를 가지며, 레이블 경로 개수에 대한 노드 경로 개수의 비율은 대략 1:18로 실험 범위 내에서 전반적으로 유사한 경향을 보인다. 더 큰 집합은 더 작은 집합이 가진 레이블 경로들을 모두 가지고 있다. 즉, 더 큰 집합은 더 작은 집합의 수퍼셋(superset)이다. 각 집합은 XRel, XParent, XIR-Braching에 의해 사용되는 각각의 테이블들을 포함하고 있는 서로 다른 세개의 데이타베이스들로 로딩된다. 그러므로, 생성되는 데이타베이스의 총 개수는 15개 이다.

그림 16은 세 방법의 인덱스 구축 시간을 포함하는 문서 파싱 시간과 로딩 시간을 보인다. 측정 결과를 통해 XIR-Branching이 XRel과 XParent에 비해 로딩 시간 측면에서는 빠르지만 파싱 시간은 더 늦음을 알 수 있다. 이것은 XRel이나 Xparent와 달리 XIR-Branching의 파싱은 루트 노드로 부터 단말 노드들까지의 노드 경로들을 생성하기 위하여 스택을 이용하여 조상 노드 정보를 관리하는 비용이 포함되는 반면, 파싱된 결과들은 XRel이나 XParent에 비해 크기가 작고(6.1.2절 참고), 더 작은 테이블들로 로딩되기 때문이다. 그림 16은 세가지 방법의 데이타베이스 구축 비용이 서로 유사한 수준임을 보이고 있다.

본 논문에서는 XRel과 XParent의 원 논문[6,7]에서 사용한 것과 같은 데이타베이스 스키마와 인덱스들을 사용하였다. 구체적으로 살펴보면, XRel의 경우 **Element, Text, Attribute** 테이블들에 존재하는 *label_path_id, start_position, end_position* 컬럼들 각각에 군집 복합 B+-tree 인덱스를 사용하였다. 또한, 같은 세 테이블의 *document_id* 컬럼에 추가적으로 B+-tree 인덱스를 사용하였고 **Text/Attribute** 테이블의 *'value'* 컬럼에는 B+-tree 인덱스를 사용하였다. XParent의 경우 **LabelPath, Element, Data** 테이블에 존재하는 *label_path_id* 컬럼들 각각에 B+-tree 인덱스를 사용하였다. 또한, 추가적으로 **Element, Data, Ancestor** 테이블들의 *node_id* 컬럼들에 B+-tree 인덱스를 사용하였고, **Ancestor** 테이블의 *ancestor_node_id* 컬럼에도 B+-tree 인덱스를 사용하였다. XRel과 마찬가지로 **Data** 테이블의 *'value'* 컬럼에 B+-tree 인덱스를 사용하였다. 3.2.2절에서 언급한 바와 같이, **DataPath** 테이블은 사용하지 않는다. XIR-Branching은 XRel이나 XParent에서와 같이 각 테이블의 *pid*와 *docid* 컬럼에 B+-tree 인덱스를 구축한 **LabelPath** 테이블로 데이타 화일들을 로드했고, **Labelpath** 테이블의 *labelpath* 컬럼에 역 인덱스를 구축했다.



(a)

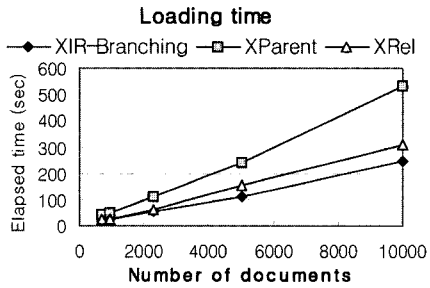


그림 16 XParent, XIR-Branching의 데이타베이스 구축 비용

6.1.2 데이타베이스 크기

그림 17은 세 가지 방법들의 최종 데이타베이스 크기를 보인다. 그림 17에서 보면, XIR-Branching의 데이타베이스 크기는 XRel이나 XParent의 크기 보다 작은 것을 볼 수 있다. 이것은 XIR-Branching이 레이블 경로와 노드 경로 정보를 더 작은 테이블들에 저장하고 더욱 간략하게 유지하기 때문이다. XIR-Branching은 같은 양의 정보를 저장하기 위한 튜플 수가 작다. 예를 들어, XRel이 동일한 노드 정보를 저장하기 위해 **Element, Text, Attribute** 테이블의 결합된 형태에서 요구되었던 튜플의 수나, XParent가 **Element, Data** 테이블의 결합된 형태에서 요구되었던 튜플의 수보다, XIR-Branching은 더 작은 튜플 수로 동일한 노드 정보를 유지한다. 이것은 **NodePath** 테이블의 *nodepath* 컬럼은 노드 식별자의 연속인 자료 형태를 활용하여, 로딩 단계에서 이미 조인된 형태로서 저장하고 있기 때문이다. 더욱이, 애트리뷰트나 텍스트의 값을 분리해서 저장하는 XRel이나 XParent와 달리, XIR-Branching은 같은 튜플 안에 *nodepath* 튜플로서 저장한다. 또한, 그

표 1 질의들

그룹	Label	Tree Pattern Query
그룹 1 (LPEs)	LPE1	//issue//author/first
	LPE2	//issue//article//author/first
	LPE3	//movie//actor//first
	LPE4	//movie/cast//actor//first
그룹 2 (BPEs without selection condition)	BPE1	//issue[.//keyword]//author[./last]/first
	BPE2	//issue[.//keyword]//article[./summary]//author[./last]/first
	BPE3	//movie[./director]//actor[./award]//first
	BPE4	//movie[./director]/cast[./actress]//actor[./award]//first
그룹 3 (BPEs with selection condition)	BPS1	//issue[.//keyword='XML']//article[./summary]//author[./last]/first
	BPS2	//issue[.//keyword='XML']//article[./summary]//author[./last='Smith']/first
	BPS3	//movie[./director/last='Mendes']//actor[./award]//first
	BPS4	//movie[./director/last='Mendes']//actor[./award='Oscar']//first

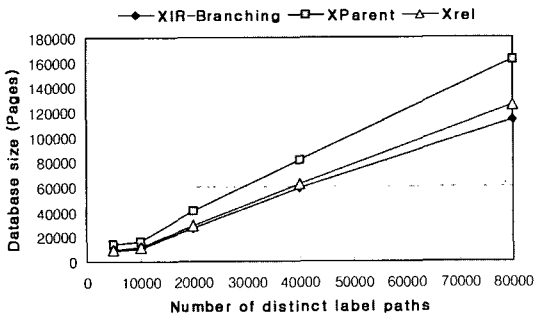


그림 17 XRel, XParent, XIR-Branching의 데이타베이스 크기

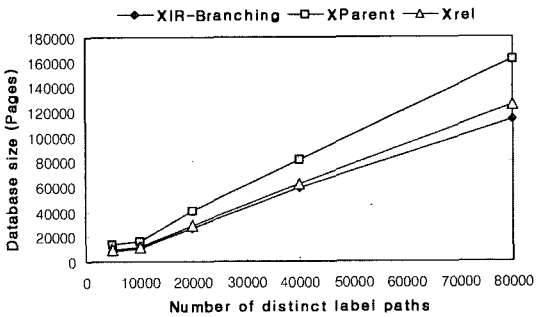


그림 18 레이블 경로가 증가함에 따른 레이블 빈도수의 변화

림 17은 XParent의 저장 공간이 가장 크다는 것을 보여준다. XParent의 저장 공간 문제는 조상-후손 관계성을 가진 노드들의 각 쌍마다 하나의 튜플을 저장하는 **Ancestor** 테이블의 높은 카디널리티 때문이다.

6.1.3 질의

표 1은 트리 패턴 질의들의 세 가지 그룹을 보인다. 그것은 (1) LPE 그룹, (2) 분기 조건 표현식이 선택 조건(selection condition)을 포함하지 않은 BPE 그룹, 그리고, (3) 분기 조건 표현식이 선택 조건을 포함하는 BPE 그룹이다. 각 그룹은 두 가지 집합의 질의들을 가진다. 하나는 *issue* 문서들에 관한 것이고, 나머지는 *movie* 문서들에 관한 것이다. 전자는 후자보다 훨씬 많은 문서 인스턴스들을 가진다. 표 1에서 소개한 질의에 대한 질의 선택률(Query Selectivity)은 *issue* 관련 질의에 대해 LPE는 10^{-4}~10^{-5}, BPE는 10^{-5}~10^{-6}, BPS는 10^{-6}~10^{-7}이며, *movie* 관련 질의에 대해 LPE는 10^{-5}~10^{-6}, BPE는 10^{-6}~10^{-7}, BPS는 10^{-6}~10^{-7}인 데이타와 질의 특성을 가진다.

크로울된 XML 문서에 대한 성격은 문서가 가진 레이블들의 분포를 통해 알 수 있다. 그림 18은 구분된 레이블 경로의 개수에 따라 계수된 레이블 빈도수, 즉 포스팅 개수를 보인다. 전체 레이블들 중 *article, issue, author, first* 등과 같은 레이블들은 크로울된 XML 문서들 내에서 가장 높은 빈도수를 가지는 레이블들을 차례대로 나열한 것이다. 레이블들 *article, issue, author, first*의 레이블 빈도수는 80000개의 레이블 경로를 기준으로 각각 5288, 3690, 2624, 1286번 나타난다. 문서 내의 나머지 레이블들의 빈도수는 *first* 레이블 보다 매우 작다. 이들 중 질의에 사용된 레이블은 *issue*와 *movie* 이며, 비례적으로 전자가 후자 보다 더 많은 문서 인스턴스를 가진다는 사실을 알 수 있다.

6.1.4 수행 환경

실험은 512 Mbyte RAM을 가진 SUN Ultra~60 워크스테이션 상에서 문서 검색 엔진(text search engine)이 필요로 하는 연산들을 지원하는 오디세우스 객체관계형 데이타베이스 관리 시스템[35][5]을 사용하여 실험을 수행하였다. 운영 체제의 예상치 못한 버퍼링 효과를 제거하기 위하여 로우 디스크 장치(raw disk device)를 사용하였다. 또한, 질의 수행 결과가 바로 이전에 수행

5) 본 시스템은 한국과학기술원(KAIST) 첨단정보기술연구센터에서 개발되었다. 본 시스템의 아키텍처는 텍스트 검색 엔진이 오디세우스 DBMS 엔진과 밀결합된 구조를 가지며, 검색을 위해 요구되는 연산들을 지원한다.

된 결과로 인해 영향 받지 않게 하기 위하여, 각 질의 수행 이후에 DBMS 버퍼를 초기화하였다. 오디세우스 DBMS에서 사용하는 정보 검색 엔진[35]은 DBMS 엔진과 밀결합된 구조를 가지고 있어 별도의 버퍼를 유지하지 않으므로 질의 처리 시 정보 검색 엔진이 사용하는 버퍼로 인한 효과는 없다. 사용된 비용 측정 기준은 수행 시간(elapsed time)과 디스크 입출력 횟수(the number of disk I/O's)이다.

6.2 실험 결과

6.2.1 실험 1: 효율성

크로울러들은 인터넷으로부터 **임의의** 문서들을 수집하기 때문에, 크로울러에 의해 새로운 문서들이 더해질 때마다 새로운 레이블 경로들이 추가된다. 그림 19는 수집된 XML 문서들로부터 구분되는 레이블 경로들을 추출한 결과이다.

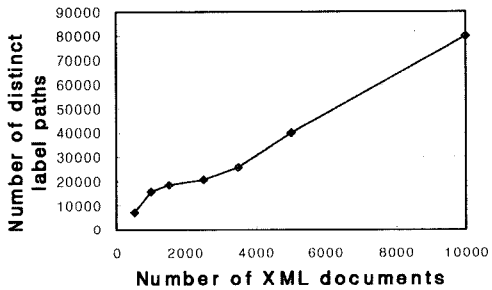
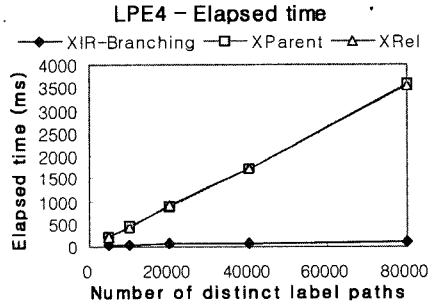


그림 19 XML 문서의 수가 증가함에 따라 변화하는 서로 다른 레이블 경로의 수



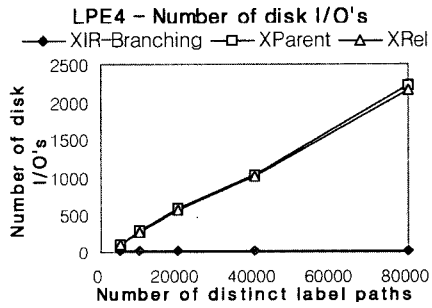
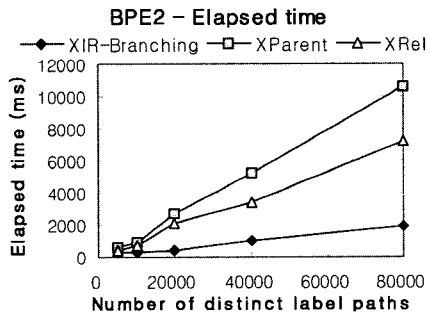


그림 20 LPE4에 대한 XRel, XParent, XIR-Branching 질의 처리 비용(버퍼 200 페이지)

그림 20은 서로 다른 레이블 경로 개수의 증가에 따른, Table 1 내의 LPE4 질의 처리 비용을 세 가지 방법에 대해 보인 것이다. 그림 21과 22는 BPE2와 BPS1에 대한 질의 처리 비용을 보인 것이다. 버퍼 크기는 불충분한 버퍼 크기로 인해 발생하는 추가적인 디스크 입출 횟수를 제거하기 위해, 4Kbyte 크기의 200 페이지를 할당하여 충분한 공간을 제공하였다. 지면의 제약으로 인해, 다른 질의들에 대한 그림은 생략하기로 한다. 그들의 비용들 역시 전형적인 커브로 유사한 경향을 보인다.

그림 20에서 부터 그림 22까지는 XRel과 XParent[6] 보다, XIR Branching이 보다 효과적임을 볼 수 있다. 성능 차이는 LPE의 경우, 수십(백)배 이상(several orders of magnitude)로 부터, BPE의 경우, 수배 이상(several factors)까지 변화한다. 특히, LPE 비용은



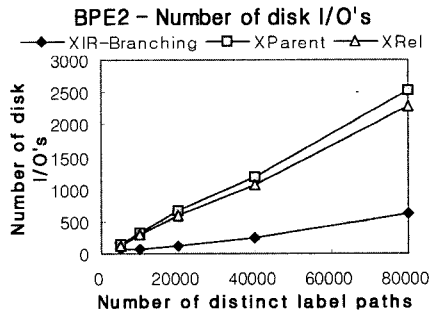


그림 21 BPE2에 대한 XRel, XParent, XIR-Branching 질의 처리 비용(버퍼 200 페이지)

6) 3.2.2장에서 언급한 바와 같이, XParent[7,8]는 부분 매취 질의를 지원하기 위하여 Ancestor 테이블을 사용한다. 그러나, 5.2절에서 논의된 바와 같이, 조인에 참여하는 Ancestor 테이블의 카디널리티가 매우 크기 때문에, XParent는 XRel 보다 나쁜 성능을 보인다. 이러한 결과는 DataPath 테이블을 사용하여 완전 매취 질의들의 성능 만을 보이고 있는 XParent에서 볼 수 있는 결과와 대조를 이루고 있다.

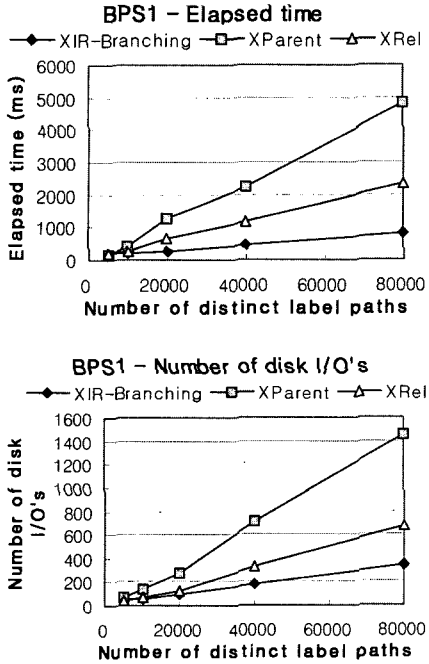


그림 22 BPS1에 대한 XRel, XParent, XIR-Branching 질의 처리 비용(버퍼 200 페이지)

XRel과 XParent의 경우, 거의 선형적(linear)으로 증가하는 반면, XIR-Branching의 경우, 서브-선형적(sub-linear)으로, 거의 상수적(nearly constant)으로 증가한다. 이러한 결과는 매칭하는 레이블 경로를 찾기 위한 방법들인, 스트링 매취와 역 인덱스 검색 간의 차이가 가져오는 영향이 반영된 것이다. BPE의 경우에, 그 비용은 세 방법 모두에서 선형적으로 증가하지만, 그 경사도는 XIR-Branching이 가장 작다. 이것은 5.2절에서 논의된 바와 같이 XIR-Branching의 조인 성능 때문이다. 선택 조건이 **없는** BPE들(그림 21)과 선택 조건이 **있는** BPE들(그림 22)을 비교할 때, 세 방법들의 비용 차이는 선택 조건이 있는 BPE들에 대해 더 작다는 것을 볼 수 있다. 이러한 결과를 보이는 이유는 XRel이나 XParent에서는 **Text**나 **Data**의 *'value'* 컬럼에 생성된 B+-tree 인덱스로 인해 이득을 볼 수 있기 때문이다.

6.2.2 실험 2: 확장성

본 절에서는 실험 데이타베이스의 크기는 고정되어 있으므로, 대신, 버퍼 크기를 줄여가면서, 커지는 데이타베이스 크기를 모의 실험하였다. 그림 23은 데이타베이스 크기(= 114000 페이지)의 0.05%로 부터 0.005% (= 6 페이지)까지 점차 작아지는 버퍼 크기에 대하여, 변화하는 질의 처리 비용을 로그 스케일로 보인다. 그림 23에서는 버퍼 크기가 점차 작아지게 될 때, XRel과 XParent는 그 처리 비용이 상당량 증가하는 것에 비해, XIR-Branching은 약간만 증가하는 것을 볼 수 있다. 이것은 XRel이나 XParent 보다 XIR-Branching이 데이타베이스 크기 관점에서 좋은 확장성을 가졌다는 것을 보여주는 결과이다.

7. 결 론

인터넷 환경에서 전형적인 형태로 존재하는, 이질적이고 방대한 량의 XML 문서들과 이에 주어질 질의를 처리하기 위하여, 본 논문에서는 XIR-Branching이라 부르는 새로운 방법을 제안하였다. XIR-Branching은 선형 경로 표현식 뿐만아니라 분기 경로 표현식까지도 효과적으로 처리하는 XML 질의 처리 방법이다. 이를 위하여, 기 제안된 XIR-Linear 방법으로 선형 경로 표현식를 처리하고, 이를 기반으로 분기 경로 표현식을 효과적으로 처리할 수 있도록 확장하였다. XIR-Linear 방법은 XML 문서에서 발생하는 레이블 경로들을 텍스트들로 간주하고, 그들 위에 역 인덱스를 생성한다. 이 역 인덱스는 선형 경로 표현식 처리 시, XRel이나 XParent의 스트링 매취 보다 훨씬 빠른 부분 매취를 지원하여 선형 경로 표현식을 효과적으로 지원한다. 그러나, XIR-Linear 만으로는 구체적인 조건을 명시할 수 있는 질의 형태인 분기 경로 표현식을 지원하지 못한다. 본 논문에서는 프리픽스 매취 조인이라는 새로운 방법을 사용하여 이러한 점을 개선하였다. 이때, 분기 경로 표현식은 여러 개의 선형 경로 표현식들로 분할되고, 각 선형 경로 표현식의 연산 결과는 프리픽스 매취 조인을 통하여 결합된다. 프리픽스 매취 조인의 사용은 XRel이나 XParent에서 사용된 포함 관계 조인에 비해, 조인 수나 카디널리티를 크게 줄일 수 있다. 본 논문에서는 또한 XML 문서의 모든 레이블 경로와 노드 경로를 저장하는 테이블들과, 역 인덱스를 포함하는 XIR- Branching의 저장 구조를 제안하였다. 이 구조에 기반하여, 선형 경로 표현식과 분기 경로 표현식에 대한 질의 처리 알고리즘들을 제안하였고, 실험을 통하여 XIR-Branching의 성능을 XRel, XParent와 비교 및 분석하였다. 이 비교를 위한 실험 데이타로는 크로울러를 사용하여 인터넷으로 부터 수집된 실제 XML 문서들을 사용하였다. 실험 결과는 XIR-Branching이 비교하는 XRel이나 XParent에 비해, 수배에서 수십배 이상까지 매우 효과적이고 확장 가능하다는 것을 보였다. 본 연구의 중요성은 XML 문서 질의 처리 기술을 인터넷 스케일에 적용 가능한 수준으로 향상시켰다는 점에 있다. 제안된 기술들은 전통적인 XML 문서 검색 엔진에 쉽게 적용될 수 있을 것으로 사료된다.

향후 연구로는 프리픽스 매취 조인을 더욱 효과적으

로 처리하기 위하여 새로운 자료 구조를 활용한 인스턴스-레벨 정보 처리 방안을 연구하고 있다.

참 고 문 헌

[1] eXtensible Markup Language(XML), http://www.w3.org/XML/.

[2] J. Naughton et al., "The Niagara Internet Query System," *IEEE Data Engineering Bulletin*, Vol. 24, No. 2, pp. 27-33, June, 2001.

[3] Xyleme, *http://www.xyleme.com*.

[4] J. Clark and S. DeRose, XML Path Language (XPath), W3C Recommendation, http://www.w3.org/TR/xpath, Nov. 1999.

[5] F. Mandreoli, R. Martoglia, P. Tiberio, "Searching Similar(Sub)Sentences for Example-Based Machine Translation," In *Proc. SEBD'02*, Isola d'Elba, Italy, June 2002.

[6] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura, "XRel: A Path-based Approach to Storage and Retrieval of XML Documents using Relational Databases," *ACM Transactions on Internet Technology(TOIT)*, Vol. 1, No. 1, pp. 110-141, 2001.

[7] H. Jiang, H. Lu, W. Wang, and J. Xu Yu, "Path Materialization Revisited: An Efficient Storage Model for XML Data," In *Proc. the 13th Australasian Database Conference(ADC)*, pp. 85-94, Melbourne, Australia, Jan. 28 - Feb. 1, 2002.

[8] H. Jiang, H. Lu, W. Wang, and J. Xu Yu, "XParent: An Efficient RDBMS-Based XML Database System," In *Proc. the 18th Int'l Conf. on Data Engineering(ICDE)*, pp. 335-336, San Jose, California, Feb. 26 - Mar. 1, 2002.

[10] Q. Li and B. Moon, "Indexing and Querying XML Data for Regular Path Expressions," In *Proc. the 27th Int'l Conf. on Very Large Data Bases(VLDB)*, pp. 361-370, Rome, Italy, Sept. 11-14, 2001.

[11] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohmann, "On Supporting Containment Queries in Relational Database Management Systems," In *Proc. 2001 ACM SIGMOD Int'l Conf. on Management of Data*, pp. 425-436, Santa Barbara, California, May 21-24, 2001.

[12] S. Al-Khalifa, H. V. Jagadish, N. Koudas, and J. M. Patel, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," In *Proc. the 18th Int'l Conf. on Data Engineering(ICDE)*, pp. 141-152, San Jose, California, Feb. 26 - Mar. 1, 2002.

[13] N. Bruno, N. Koudas, and D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching," In *Proc. 2002 ACM SIGMOD Int'l Conf. on Management of Data*, pp. 310-321, Madison, Wisconsin, June 3-6, 2002.

[14] H. Jiang, H, Lu, W, Wang, and B. C. Ooi, "XR-Tree: Indexing XML Data for Efficient Structural Joins," In *Proc. the 19th Int'l Conf. on Data Engineering(ICDE)*, pp. 253-264, Bangalore, India, Mar. 5-8, 2003.

[15] H. Jiang, W. Wang, H. Lu, and J. X. Yu, "Holistic Twig Joins on Indexed XML Documents," In *Proc. the 29th Int'l Conf. on Very Large Data Bases(VLDB)*, pp. 273-284, Berlin, Germany, Sept. 9-12, 2003.

[16] Jan-Marco Bremer and Michael Gertz, "XQuery/IR: Integrating XML Document and Data Retrieval," In *Proc. the Fifth Int'l Workshop on the Web and Databases(WebDB 2002)*, pp. 1-6, Madison, Wisconsin, 2002.

[17] Daniela Florescu, Donald Kossmann, and Ioana Manolescu, "Integrating Keyword Search into XML Query Processing," In *Proc. the 9th WWW Conference/Computer Networks*, pp. 119-135, Amsterdam, NL, May 2000.

[18] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram, "XRANK: Ranked Keyword Search over XML Documents," In *Proc. 2003 ACM SIGMOD Int'l Conf. on Management of Data*, pp. 16-27, San Diego, California, June 9-12, 2003.

[19] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. Viglas, Y. Wang, J. F. Naughton, and D. J. DeWitt, "Mixed Mode XML Query Processing," In *Proc. the 29th Int'l Conf. on Very Large Data Bases(VLDB)*, pp. 225-236, Berlin, Germany, Sept. 9-12, 2003.

[20] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon, "A Fast Index for Semistructured Data," In *Proc. the 27th Int'l Conf. on Very Large Data Bases(VLDB)*, pp. 341-350, Rome, Italy, Sept. 11-14, 2001.

[21] 박영호, 한욱신, 황규영, "정보 검색 기술을 이용한 대규모 이질적인 XML 문서에 대한 효율적인 선형 경로 질의 처리," 정보과학회논문지:데이타베이스, 제31권, 제5호, 2004년 10월.

[22] G. Salton and M. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, New York 1983.

[23] S. Al-Khalifa, H. V. Jagadish, N. Koudas, and J. M. Patel, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," In *Proc. the 18th Int'l Conf. on Data Engineering(ICDE)*, pp. 141-152, San Jose, California, Feb. 26 - Mar. 1, 2002.

[24] A. Aboulnaga, A. R. Alameldeen, and J. Naughton, "Estimating the Selectivity of XML Path Expressions for Internet Scale Applications," In *Proc. the 27th Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 591-600, Rome, Italy, Sept. 11-14, 2001.

[25] N. Polyzotis and M. Garofalakis, "Statistical Synopses for Graph-structured XML Databases,"

In *Proc. 2002 ACM SIGMOD Int'l Conf. on Management of Data*, pp. 358-369, Madison, Wisconsin, June 3-6, 2002.

[26] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth, "Covering Indexes for Branching Path Queries," In *Proc. 2002 ACM SIGMOD Int'l Conf. on Management of Data*, pp. 133-144, Madison, Wisconsin, June 3-6, 2002.

[27] M. Altinel, M. J. Franklin, "Efficient Filtering of XML Documents for Selective Dissemination of Information," In *Proc. the 26th Int'l Conf. on Very Large Data Bases(VLDB)*, pp. 53-64, Cairo, Egypt, Sept. 10-14, 2000.

[28] Z. Ives, A. Levy, and D. Weld, Efficient Evaluation of Regular Path Expressions on Streaming XML Data, Technical Report UW-CSE-2000-05-02, University of Washington, 2000.

[29] J. McHugh, J. Widom, "Query Optimization for XML," In *Proc. the 25th Int'l Conf. on Very Large Data Bases(VLDB)*, pp. 315-326, Edinburgh, Scotland, UK, Sept. 7-10, 1999.

[30] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang, "Storing and Querying Ordered XML Using a Relational Database System," In *Proc. 2002 ACM SIGMOD Int'l Conf. on Management of Data*, pp. 204-215, Madison, Wisconsin, June 3-6, 2002.

[31] R. Goldman and J. Widom, "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases," In *Proc. the 23th Int'l Conf. on Very Large Data Bases(VLDB)*, pp. 436-445, Athens, Greece, Aug. 26-29, 1997.

[32] Kyu-Young Whang, Min-Jae Lee, Jae-Gil Lee, Min-Soo Kim, and Wook-Shin Han, "Odysseus: a High-Performance ORDBMS Tightly-Coupled with IR Features," Technical Report CS-TR-2004-204, Department of Computer Science, *http://cs.kaist.ac.kr/research/technical/Archive/CS-TR-2004-204.pdf*, KAIST, Dec., 2004.

[33] Teleport Pro Version 1.29, *http://www.tenmax.com/teleport/pro/home.htm*.

[34] ReGet Deluxe 3.3 Beta(build 173), *http://deluxe.reget.com/en/*.

[35] Kyu-Young Whang, Min-Jae Lee, Jae-Gil Lee, Min-Soo Kim, and Wook-Shin Han, "Odysseus: a High-Performance ORDBMS Tightly-Coupled with IR Features," In *Proc. the 21st Int'l Conf. on Data Engineering,(ICDE)*, National Center of Sciences, Tokyo, Japen, April 5-8, 2005.

박 영 호

1990년 2월 동국대학교 컴퓨터공학과 학사. 1992년 2월 동국대학교 컴퓨터공학과 석사. 1999년 2월 한국전자통신연구원 선임연구원. 1999년 3월~2005년 8월 한국과학기술원 전산학과 박사. 관심분야는 XML, Web, DBMS, 정보 검색(IR), 운영체제, 임베디드 시스템

한 욱 신

정보과학회논문지 : 데이타베이스 제 32 권 제 3 호 참조

황 규 영

정보과학회논문지 : 데이타베이스 제 32 권 제 3 호 참조