동적 XML 조각 스트림에 대한 메모리 효율적 질의 처리

이 상 욱[†] · 김 진[††] · 강 현 철[†††]

요 약

본 논문은 메모리 용량이 제약되어 있는 이동 단말기에서의 XML 데이터에 대한 질의 처리 기술에 관한 것이다. 대량의 XML 데이터에 대한 질의를 메모리 용량이 크지 않은 단말기에서 처리하는 경우 XML 데이터를 XML 조각(fragment)으로 분할하여 스트림으로 전송하고 처리하는 기술이 필요하다. 이는 전체 XML 문서를 재구성하지 않고 XML 데이터에 대한 질의 처리를 가능하게 한다. XFrag[4], XFPro[5], XFLab[6] 등 기존에 제시된 기법들은 질의 처리를 위해 조각에 대한 정보를 저장하고 사용한 후 더 이상 불필요해진 것들을 식별하여 삭제하지 못하기 때문에 조각 정보가 메모리에 계속 누적되어 대용량의 XML 데이터에 대해 질의 처리를 수행하기에는 문서 크기에 따른 확장성(scalability)이 떨어진다. 특히, XML 조각이 동적으로 생성되어 무한정 스트리밍되는 경우에 한정된 메모리로는 질의 처리를 보장할 수 없다. 본 논문에서는 동적 XML 조각 스트림에 대한 질의 처리에 있어 문서 크기에 따른 확장성 있는 질의 처리를 수행하기 위하여 누적된 조각 정보 삭제 기법들을 제시하고 이들을 바탕으로 기존 기법의 확장을 제시한다. 구현 및 성능 실험 결과 본 논문에서 확장된 기법이 기존의 기법보다 메모리 효율성이 현저히 높고 문서 크기에 따른 확장성이 월등히 우수한 것으로 나타났다.

키워드 : XML 조각 스트림, XML 질의 처리, XML 레이블링, 홀-필러 모델

Memory Efficient Query Processing over Dynamic XML Fragment Stream

Sangwook Lee[†] · Jin Kim[††] · Hyunchul Kang[†††]

ABSTRACT

This paper is on query processing in the mobile devices where memory capacity is limited. In case that a query against a large volume of XML data is processed in such a mobile device, techniques of fragmenting the XML data into chunks and of streaming and processing them are required. Such techniques make it possible to process queries without materializing the XML data in its entirety. The previous schemes such as XFrag[4], XFPro[5], XFLab[6] are not scalable with respect to the increase of the size of the XML data because they lack proper memory management capability. After some information on XML fragments necessary for query processing is stored, it is not deleted even after it becomes of no use. As such, when the XML fragments are dynamically generated and infinitely streamed, there could be no guarantee of normal completion of query processing. In this paper, we address scalability of query processing over dynamic XML fragment stream, proposing techniques of deleting information on XML fragments accumulated during query processing in order to extend the previous schemes. The performance experiments through implementation showed that our extended schemes considerably outperformed the previous ones in memory efficiency and scalability with respect to the size of the XML data.

Key Words : XML Fragment Stream, XML Query Processing, XML Labelling, Hole-Filler Model

1. 서 론

XML이 웹에서 데이터 교환의 표준으로 부각된 이래, 정보 배포, 출판-구독, 센서 네트워크, 모니터링 등의 응용 분야에서 XML 스트림 데이터에 대한 효율적인 질의 처리에 관한 연구가 활발히 진행되고 있다. 이들 기존 연구는 스트림 데이터의 특성으로 인해 많은 메모리의 사용을 전제로 한 기술에 집중되어 있다. 최근 유비쿼터스 컴퓨팅이 새로운 컴퓨팅 패러다임으로 부각되면서 자체 자원과 컴퓨팅 파워를 갖춘 이동 단말기가 널리 보급되고 있고 이들 단말기에서 직접 서버로부터 스트리밍되는 XML 데이터에 대해 질의 처리를 수행하는 기술이 요구되고 있다. 예를 들어, XML로 표현된 주가 데이터가 불특정 다수의 사용자 (즉, 이동 단말기)에게 방송 채널을 통해 스트리밍되면 각 사용자는 이동 단말기에 등록해둔 연속 질의를 처리하여 적절한 거래를 수행할 수 있다[1]. 이러한 응용을 위한 기술이 기존의 XML 스트림 질의 처리 기술에 비해 새로운 점은 이동 단말기의 메모리 제약을 극복해야 하기 때문이다.

본 논문은 가용 메모리 용량이 적은 이동 단말기에서의

※ 본 논문은 한국과학재단 특정기초연구사업(R01-2006-000-10609-0) 지원으로 수행되었음.
† 준 회 원 : 중앙대학교 대학원 컴퓨터공학과 석사과정
†† 정 회 원 : 중앙대학교 대학원 컴퓨터공학과 석사과정
††† 종신회원 : 중앙대학교 컴퓨터공학부 교수
논문접수 : 2007년 8월 31일, 심사완료 : 2007년 10월 29일

DOI: 10.3745/KIPSTD.2008.15-D.1.1

XML 데이터에 대한 질의 처리 기술에 관한 것이다. XML 데이터는 계층적 구조를 가지고 있으며 용량이 클 수 있다. 따라서 이동 단말기의 적은 메모리로 대량의 XML 데이터를 처리하려면, XML 데이터를 적절한 조각(fragment)으로 분할하여 조각 단위로 처리하는 기술[2]이 요구된다.

그러한 기술로 홀-필러(hole-filler) 모델[1, 3]을 이용한 XFrag[4] 및 XFPro[5], 그리고 XFLab[6]이 제시되었다. 그러나 이들 기존의 기법들은 XML 문서 크기에 따른 **확장성(scalability)**에 문제가 있다. 이는 질의 처리 대상이 되는 XML 조각에 대한 정보를 저장하고 사용한 후 더 이상 불필요해진 것들을 식별하여 삭제하지 못하므로 질의 처리가 수행되는 동안에 이들 정보가 누적되어 메모리 사용량이 계속 증가하기 때문이다. 따라서 XML 문서에 엘리먼트가 **동적**으로 추가되어 무한대로 스트리밍되는 경우에는 가용 메모리 초과로 질의 처리를 수행할 수 없는 상황이 일어날 수도 있다. (그림 1)은 XMark 벤치마크[7]의 xmlgen 프로그램으로 생성한 auction.xml 문서의 일부를 개괄적으로 나타낸 것이다. 경매 정보를 표현하는 이 문서에는 open_auction과 bidder 엘리먼트가 계속 반복되어 나타나며 특히 bidder는 정적으로 확보되기 보다는 가격 제시(bidding)가 일어날 때마다 동적으로 생성된다. bidder 엘리먼트는 경매가 진행되는 동안 계속 생성되어 추가될 수 있으며, 이는 문서의 크기가 계속해서 증가한다는 것을 의미한다. 이러한 상황에서 기존 기법들로는 계속해서 누적되는 XML 조각 정보로 인해 한정된 메모리로 질의 처리가 불가능해진다.

본 논문에서는 이와 같이 기존의 기법들이 조각 정보의 누적으로 인해 XML 문서의 크기가 증가할 경우 메모리 사용량이 증가하는 점을 해결하기 위해 누적된 조각 정보를 **삭제**하기 위한 기법들을 제시한다. 그리고 이들 기법을 바탕으로 기존 기법의 확장을 제시한다. 확장된 기법은 XML 문서 크기에 대해 확장성이 있으며, 특히 XML 문서의 크기가 동적으로 무한정 증가하는 상황에서도 안정적으로 질의 처리를 수행할 수 있다.

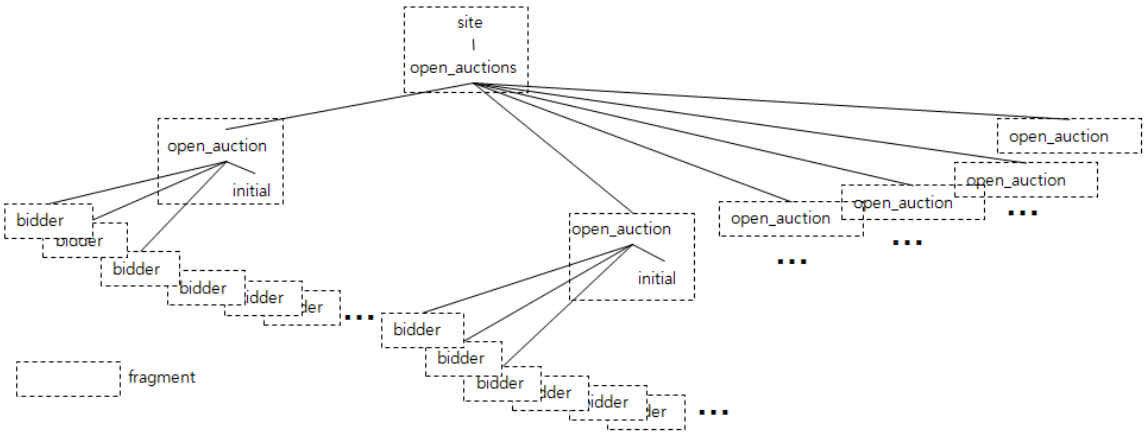
본 논문의 구성은 다음과 같다. 2절에서는 관련 연구를 살펴본다. 3절에서는 본 논문에서 제시하는 기법을 기술한다. 4절에서는 구현 및 성능 평가 결과를 기술한다. 5절에서는 결론을 맺고 향후 연구 내용을 기술한다.

2. 관련연구

다수의 사용자에게 XML 데이터를 조각으로 분할하여 스트리밍하고 이를 대상으로 클라이언트에서 질의 처리를 수행하기 위한 XStreamCast[8]의 기술 요소로 홀-필러 모델[1, 3]이 제안되었다. 홀-필러 모델은 XML 조각 사이의 관계를 표현하는 모델로서 조각 사이의 관계를 홀(hole)에 대응하는 필러(filler)로 기술하고 있다. 전체 문서는 조각으로 분할될 때 조각이 생성되는 위치에 홀 엘리먼트를 추가하고 홀 식별자(hole id)를 부여한다. 생성된 조각은 필러 엘리먼트로 감싸지고 홀 식별자와 동일한 필러 식별자(filler id)가 부여된다. 분할된 조각들 간에는 홀 식별자와 필러 식별자를 비교함으로써 그 관계를 식별할 수 있다.

클라이언트는 조각 스트림을 대상으로 질의 처리를 수행하기 위해 전체 XML 문서의 구조와 이것이 어떻게 분할되었는지에 대한 정보를 필요로 하는데 이를 나타내기 위해 태그 구조(tag structure)라는 메타 데이터를 사용한다. 태그 구조에는 원본 XML 문서 내의 엘리먼트 태그 이름, 엘리먼트의 필러(filler) 정보 및 태그 식별자(tsid) 정보가 기록되어 있다. 태그 이름은 엘리먼트 이름이고, 필러 정보는 해당 엘리먼트가 개별의 조각으로 분할될 것인가를 나타내며, 태그 식별자는 해당 엘리먼트의 구조적 위치에 대한 고유 식별자이다.

XML 문서를 조각으로 분할하여 스트리밍하고 이에 대해 질의 처리를 수행하는 기법으로 제일 먼저 XFrag[4]가 제안되었다. XFrag 연구에서는 홀-필러 모델로 분할된 XML 조각 스트림과 태그 구조를 받아서 XML 질의 처리를 수행할 수 있는 질의 연산자 파이프라인 처리 기법을 제시하였다. 질의 연산자 파이프라인은 XPath 질의를 구성하는 각 스텝(XPath location step)에 대응되는 연산자들로 구성된다. 이들 연산자들은 조각 처리 과정에서 문의(inquiry) 및 트리거(trigger)를 통해 서로 유기적으로 연계된다. 또한 각 연산자는 연관 테이블(association table)이라는 자료 구조에 질의



(그림 1) 반복되며 동적으로 생성되는 open_auction과 bidder 엘리먼트

처리에 필요한 조각 정보를 저장한다.(이하 '질의 연산자 파이프라인'을 간단히 '파이프라인'이라고 지칭.)

XFPro[5]는 홀-필러 모델의 태그 구조를 이용하여, 질의 처리 계획을 생성하고 XFrag에서 제안한 파이프라인의 최적화를 수행한다. 최적화된 파이프라인의 기본 구조나 동작 방식은 XFrag와 유사하지만 XFrag의 파이프라인에 비해 XML 조각을 처리하는 단계가 줄어들게 되어 처리 시간 면에서 개선 효과를 얻을 수 있다.

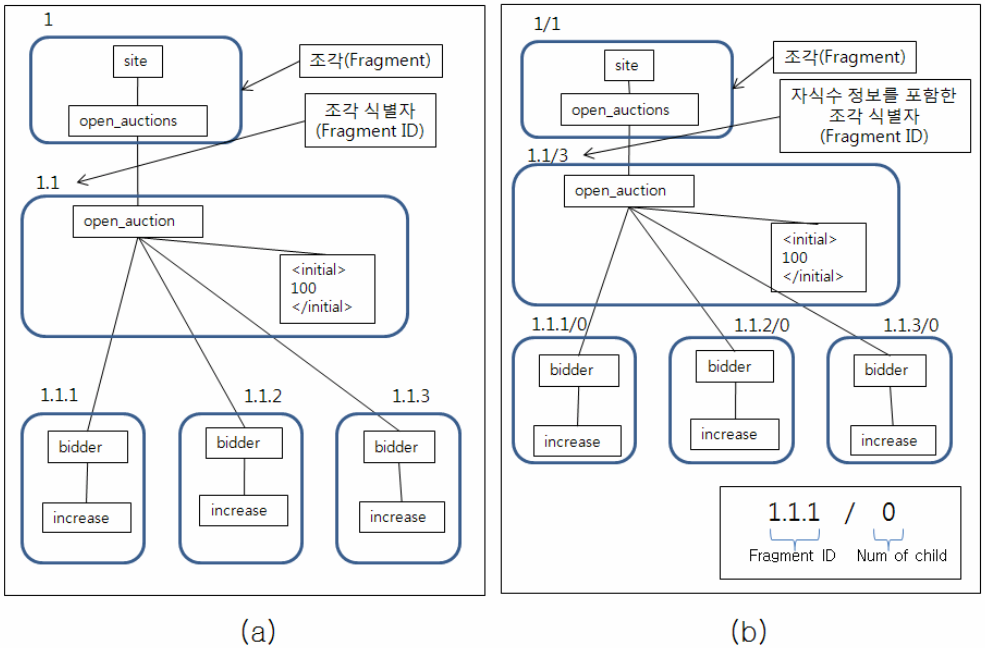
XFrag와 XFPro는 홀-필러 모델의 특징으로 인해 메모리 효율성에 문제점을 안고 있다. [9]에 의하면 웹 상의 거의 모든 대용량 XML 문서는 깊이는 깊지 않고 옆으로 넓게 퍼진 구조로 되어 있는데, 이러한 XML 문서를 홀-필러 모델로 분할하면 매우 많은 홀을 포함하는 조각을 생성하게 된다. 이런 조각은 그 크기가 커지게 되는데 이는 단말기에서의 메모리 사용량을 증가시키는 주요 원인이다. 또한 XML 문서에 새 조각이 동적으로 추가되는 상황에서는 추가된 조각의 전송뿐만 아니라 그 조각에 대응되는 홀을 포함하는 부모 조각도 재전송 해야 한다[3]. 이러한 문서 변경 시나리오는 엘리먼트가 동적으로 추가되는 상황에서 전송 및 처리 효율에 많은 부담을 주게 된다.

XFLab[6]은 홀-필러 모델의 문제점을 해결하기 위해 XML 레이블링 기법을 이용하여 XML 조각 간의 관계를 표현한다. XML 문서를 조각들로 분할하면 각 조각을 노드로 하는 조각 트리가 생성되는데, XFLab은 이들 조각 간의 관계를 나타내기 위해 각 조각에 XML 레이블링 기법을 이용하여 조각 식별자(Fragment ID)를 할당한다. (그림 2)는 XML 조각 트리의 예를 나타낸 것이다. (그림 2(a))의 조각 트리에서 각 조각에 할당된 일련의 레이블은 해당 조각의 식별자로서 Dewey order encoding[10]으로 할당한 것이다. XFrag가 홀 식별자와 필러 식별자를 이용하여 부모 조각과 자식 조각의 비교를 기반으로 질의 처리를 수행하였던 것에 반해 XFLab은 XML 레이블링 기법을 이용하여 XML 조각 간의 부모-자식 뿐만 아니라 조상 후손 관계까지 식별하여 질의 처리를 수행하며 이를 통해 메모리 사용량을 현저히 줄였다.

그러나 이러한 XFrag, XFPro 및 XFLab 기법들은 XML 조각 스트림을 처리하는 데 있어 지속적으로 조각 정보가 누적되어 XML 문서의 크기가 매우 클 경우 한정된 메모리로는 질의 처리를 계속 수행할 수 없는 상황에 이르게 된다. 본 논문에서 제시하는 기법은 이러한 정보 누적에 대한 가용 메모리 감소와 궁극적으로 가용 메모리 초과 문제를 해결하여 문서 크기에 따른 확장성을 제공하기 위한 것이며, 이를 위해 누적된 조각 정보 중 더 이상 질의 처리와 관계없는 것을 식별하여 삭제하는 기법을 제안한다.

3. XML 문서 크기 증가에 따른 확장성 있는 질의 처리 기법

XML 문서 크기 증가에 대해 확장성 있는 질의 처리를 수행하기 위해서는 질의 처리 중에 저장된 조각 정보를 적절히 삭제할 수 있는 기술이 필수적이다. 하지만 질의 처리의 정확성을 보장하면서 질의 처리와 연관된 정보를 삭제하려면 어떤 정보가 더 이상 필요 없는지 정확히 식별해야 한다. 또한 이러한 삭제로 인해 기존 기법들에 비해 상대적으로 부족한 정보를 이용하여 정확한 질의 처리를 수행하기 위해서는 기존의 기법들보다 진보된 조각 관계 표현 및 처리 기법이 필요하다. 본 절에서는 먼저 질의 처리 과정에서 조각 정보가 누적되는 현상에 대해 설명한 후, 이러한 문제를 해결할 수 있는 기법에 대해 설명한다.



(그림 2) XML 조각 트리에 조각 식별자를 할당한 예

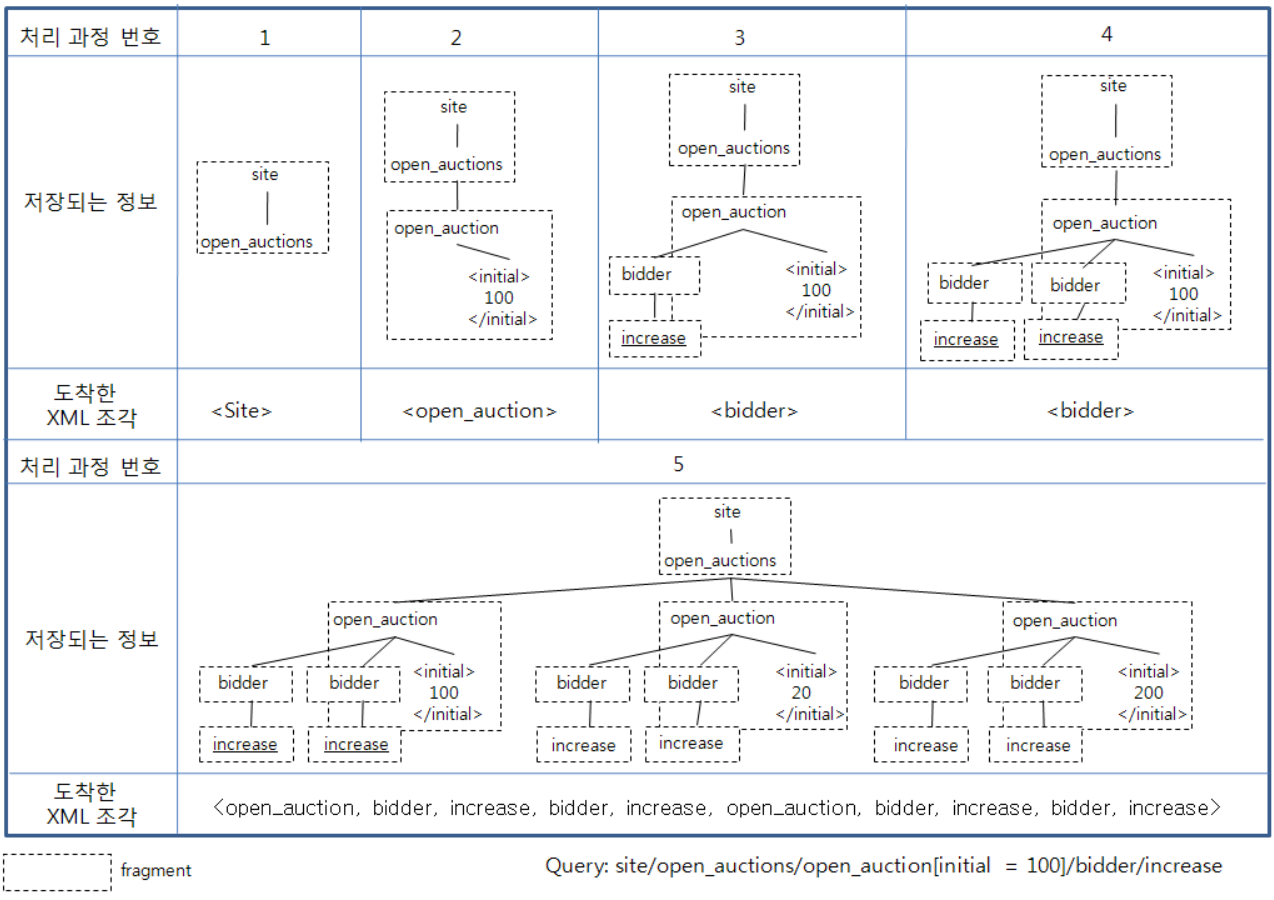
3.1 조각 정보 누적

본 절에서는 2절 관련 연구에서 살펴본 XML 조각 스트림 질의 처리 기법들이 질의 처리를 수행하기 위해 어떤 정보들을 저장하는지 살펴보고, 이러한 처리가 초래하는 조각 정보 누적에 대해 auction.xml 문서에 대한 아래 XPath 질의를 예로 들어 기술한다.

site/open_auctions/open_auction[initial = 100]/bidder/increase

(그림 3)은 기존의 XFrag, XFPro 및 XFLab이 주어진 질의를 처리하기 위해 저장하는 정보를 질의 처리 과정 순서별로 나타낸 것이다. 점선으로 표시한 사각형은 사각형 안의 엘리먼트들이 하나의 조각으로 분할되어 있음을 나타낸다. 도착한 XML 조각과 그 순서를 '도착한 XML 조각' 항목에 <첫 번째 조각의 루트 엘리먼트 이름, …, n번째 도착한 조각의 루트 엘리먼트 이름>의 형식으로 표시하였다. 또한 질의의 결과로 선택되는 엘리먼트는 엘리먼트 이름에 밑줄을 그어 표시하였다. (그림 3)의 과정 2에서는 open_auction 조각(이하 '조각' 생략)이 도착하면 initial의 값(100)을 확인하여 도착한 open_auction이 주어진 조건을 만족하는지 확인한다. 도착한 open_auction은 주어진 조건을 만족하므로 과정 3, 과정 4에서 도착한 bidder와 increase가 저장되고 이중 increase는 결과로 선택된다. 이때 bidder와 increase의 정보는 지워지지 않고 계속 저장된다. 이는 향후 XML 문서 내의 일부 데이터가 수정됨에 따라 해당 조각을 변경된 것으로 재전송하여 XML 조각이 교체되었을 때의 질의 처리를 보장하기 위한 것이다. 과정 5에서 도착한 두 번째 open_auction의 경우 initial 엘리먼트의 값(20)이 조건을 만족하지 않는데, 이 경우 역시 bidder와 increase의 정보는 저장된다. 이는 initial이 수정되어 재전송되었을 때의 질의 처리를 보장하기 위한 것이다.

이상에서 살펴본 바와 같이 기존의 XFrag, XFPro, XFLab은 질의 처리에 필요한 XML 문서의 조각 정보를 저장한 후 삭제하지 않는다. 이러한 방법은 일부 조각이 수정될 때의 질의 처리를 고려한 것이지만 이것으로 인해 이미 처리가 끝난 조각의 정보들을 모두 저장하여야 하는 부담이 있다. 특히 위에서 든 initial 값 변경 예의 경우, open_auction의 의미 상 initial 값 (즉, 경매에서 최초 제시 가격)은 변하지 않으며 따라서 이미 처리가 끝난 bidder 및 increase의 정보는 저장할 필요가 없다. 일반적으로 XML 문서는 응용의 성격 상 변경되지 않는 정적인 부분과 변경이 가능한 동적인 부분으로 나눌 수 있다. 이상의 관찰에 따르면 처리가 완료된 대부분의 조각 정보들은 질의 처리의 정확성을 보장하면서도 삭제하는 것이 가능하다. 3.2절에서는 이러한 관찰에 근거하여 질의 처리의 정확성을 보장하면서 질의 처리 과정에서 누적된 조각 정보들 가운데 더 이상 활용하지 않는 것을 식별하여 삭제할 수 있는 기법들을 설명한다.



(그림 3) XML 조각 도착에 따른 XFrag, XFPro, XFLab의 질의 처리 과정

〈표 1〉 누적된 조각 정보 삭제 기법 요약

조각 정보 삭제 기법 \ 조각 관계 표현 방법	XML 레이블링	홀 필러 모델
자식 수 세기	자식 수 정보 이용을 통한 적용	홀 수 세기를 이용한 적용
중복된 조건 값 제거	중복된 조건 값 제거 적용	중복된 조건 값 제거 적용
홀 공유	해당 없음	홀 공유 적용 가능
무효화된 후손 삭제	무효화된 후손 삭제 적용	적용 불가
레이블링 기법을 이용한 경로 단축	파이프라인 최적화 적용	적용 불가

3.2 누적된 조각 정보 삭제 기법

3.1절에서 살펴본 것처럼 기존의 XFrag, XFPro, XFLab은 문서 크기의 증가에 따른 확장성 문제가 있다. 이와 같은 문제를 해결하기 위해 본 논문에서는 XFrag, XFPro, XFLab에 적용할 수 있는 누적된 조각 정보의 삭제 기법들을 제시한다. <표 1>은 이러한 기법들을 XML 조각 관계 표현 방법 별로 요약한 것이다. 본 절에서는 표 1에 열거된 조각 정보 삭제 기법들을 각각 설명하고, 이러한 기법들을 XFrag, XFPro, XFLab에 어떻게 적용하는지를 기술한다. 조각 정보 삭제 기법의 적용 여부는 조각 관계 표현 방법과 관련이 있다. XFrag와 XFPro는 모두 홀 필러 모델을 사용하기 때문에 두 기법의 적용 여부는 항상 동일하므로 앞으로의 기술에서 XFPro는 언급하지 않는다.

3.2.1 자식 조각 수 세기

어떤 조각의 자식 조각들이 모두 도착했고(이하 삭제 조건-1) 이 조각이 관련된 질의 내 프리디킷(predicate) 조건의 참, 거짓을 판단하는 데 의존하는 XML 조각이 정적이면서 참, 거짓이 결정되었다면(이하 삭제 조건-2) 해당하는 조각을 삭제할 수 있다. 예를 들어 (그림 3)에서 과정 3의 처리 후 initial이 변하지 않는다면 increase가 도착하여 질의 처리를 수행한 후 bidder와 increase 정보를 삭제할 수 있다. 여기서 bidder가 참, 거짓을 판단하는데 의존하는 조각은 initial 엘리먼트가 포함된 조각이다. bidder의 조각 정보를 삭제하는 데 있어 삭제 조건-1을 만족해야 하는 이유는 아래와 같은 질의의 경우 자식 조각의 문의에 대해 응답해야 하기 때문이다.

site/open_auctions/open_auction[initial = 100]/bidder[date = 10/12/1999]/increase

이러한 질의의 경우 bidder의 삭제가 increase의 도착 전에 일어난다면 increase의 문의에 대해 응답할 수 없으므로, 조각 정보의 삭제는 해당 조각의 모든 자식 조각이 도착한 후 가능하다. 살펴본 바와 같이 어떤 조각을 삭제하기 위해서는 해당 삭제 조건-1을 만족하는지 확인할 수 있는 방법이 필요하다. 본 논문에서는 이러한 방법으로 XFrag에서는 조각 내 홀의 수를 세어 도착한 조각 (즉, 필러 조각)의 수와 비교하는 방법을 사용하였고 XFLab에서는 각 조각의 식별자에 자식 조각의 수를 명시하고 도착한 자식 조각의 수를 세는 방법을 사용하였다. 따라서 XFLab 기법에서는 조각의 자식 수를 나타내기 위해 (그림 2(b))와 같이 확장된 조각 식별자를 사용한다.

삭제 조건을 만족하는 조각 정보를 삭제할 때는 조각 단위의 삭제를 수행해야 한다. 그런데 파이프라인을 구성하고 있는 연산자는 엘리먼트 단위로 조각 정보를 저장하므로 다수의 엘리먼트에 대한 정보를 포함하는 조각에 대한 정보는 여러 연산자에 흩어져 저장되게 된다. 이렇게 여러 연산자에 저장되어 있는 조각 정보들을 동시에 삭제하기 위해서는 연산자간의 동의를 거친 삭제가 필요하다. (그림 4)의 알고리즘은 각 연산자에서 삭제와 관련된 이벤트가 발생했을 경우 수행되는 연산자 간 삭제 동의 과정을 나타낸다. 이러한 삭제 동의 과정을 거쳐 모든 연산자가 삭제에 동의하였을 때, FID(Fragment ID 혹은 Filler ID)로 식별되는 조각 정보는 모두 삭제된다.

XFrag와 XFLab 모두 이러한 연산자 간의 삭제 동의를 거쳐 조각 정보를 삭제하지만 동의의 기준에서 조금 다르다. (그림 4)의 알고리즘에서 3행이 해당 연산자가 삭제에 동의할 것인지를 판단하는 부분인데 XFrag는 3행에서 삭제 조건-1과 삭제 조건-2를 모두 확인하지만 XFLab의 경우는 삭제 조건-2만 확인한다. 이것은 XFLab이 삭제 조건-1을 확인하는 데 필요한 정보를 파이프라인에 저장하지 못하기 때문이다. 자식 수 세기에 의한 조각 정보 삭제를 수행하기 위해서는 질의 경로 상의 모든 조각들에 대해 조각 정보를 저장해야 한다. XFrag 기법은 XPath 질의의 조상-후손 축

알고리즘 : negotiateRemovable(조각 정보 삭제 협상)

```
   입력 : op(협상을 시도한 연산자), fragmentID(조각 식별자)
   출력 : TRUE, FALSE(협상 결과)
1: IF (op != NULL && op.isTargetElementInOtherFragment == FALSE ) THEN
   /* 같은 조각을 대상으로하는 연산자인 경우 */
   2: item = op.getAssTableItem(fragmentID);
   /* fragmentID로 검색된 조각 정보가 삭제 가능 한가 */
   3:   IF (item.checkRemoveCondition()) THEN
      / * 하위 연산자로 협상 시도 */
   4:    IF (negotiateRemovable(op.successor, fragmentID)) THEN
   5:      return TRUE;
   6:    ELSE
   7:      return FALSE;
   8:    END IF
   9:  ELSE
   10:   RETURN FALSE;
   11:  END IF
   12:ELSE
    /* 다른 조각을 대상으로 하는 연산자인 경우 */
   13:  RETURN TRUE;
   14:END IF
```

(그림 4) 조각 정보 삭제 협상 알고리즘

알고리즘 : tryToRemove(조각 관리자의 동기화된 삭제)

```
입력 : fragmentID(삭제 대상 조각의 식별자)
   /* 파이프라인의 동의를 거침 */
1: IF (sendNoticeToPipeline(fragmentID) == TRUE )
2:   removeChildCounter(fragmentID);
3: END IF
```

(그림 5) 조각 관리자의 동기화된 삭제 알고리즘

알고리즘 : tryToRemove(파이프라인의 동기화된 삭제)

```
입력 : fragmentID(삭제 대상 조각의 식별자)
       needNegotiation(협상을 수행해야 하는지 여부)
   /* 자신이 속한 조각의 루트 엘리먼트와 관련된 연산자를 선택 */
1: op = getFragmentRootElementOperator();
2: item = op.getAssTableItem(fID);
3: negotiateResult = item.checkRemoveCondition();
   /* 조각의 루트 엘리먼트의 삭제 조건이 만족하였을 때 */
4: IF (negotiateResult==TRUE) THEN
    /* 연산자간의 삭제 동의 */
5:   negotiateResult = negotiateRemovable(op.successor, fragmentID);
     /* 조각 관리자와 삭제 동의 */
6:   IF (sendNoticeToFM(fragmentID)==TRUE) THEN
      /* 조각 관리자가 삭제에 동의 하였을 경우 파이프라인내의 조각 정보 삭제 */
7:     removeFragmentInfo(op, fragmentID);
8:   END IF
9: END IF
```

(그림 6) 파이프라인의 동기화된 삭제 알고리즘

을 자식 축으로 구성된 경로로 변환한 후 파이프라인을 생성하는데 이것은 홀 필러 모델이 부모 자식 간의 관계만을 표현할 수 있기 때문이다. 이로 인해 XFrag는 경로 정보를 저장하기 위한 추가 저장 공간을 필요로 하지만 경로 상의 모든 연산자가 파이프라인에 존재하므로 연산자들의 연관 테이블에 조각의 자식 조각 정보를 저장할 수 있게 된다. 하지만 XFLab은 조상 후손 축에 대해서도 변환 없이 파이프라인을 생성하고 또한 레이블링 기법을 이용한 경로 단축 기법에 의해 경로 상의 엘리먼트에 대한 연산자가 일부 삭제된다. 이로 인해 XFLab의 경우 연산자들 내의 연관 테이블에 경로 상의 모든 조각의 자식 조각 정보를 저장하는 것이 불가능하므로 XFLab은 파이프라인 외부의 조각 관리자에 조각의 자식 조각 정보를 저장한다. XFLab의 조각 관리자는 질의와 연관된 모든 경로 상의 조각에 대해 총 자식 수와 도착한 자식 조각에 대한 정보를 저장한다. 이로 인해 XFLab에서는 조각 정보가 조각 관리자와 파이프라인의 연산자들에 나누어져 저장된다.

상기의 이유로 인해 XFLab은 흩어져서 저장된 조각 정보들을 동시에 지우기 위하여 연산자 간의 삭제 동의 이전에 조각 관리자와 파이프라인 간의 동의가 필요하다. (그림 5)-(그림 6)의 알고리즘은 삭제와 관련된 사건이 일어났을 때, 조각 관리자와 파이프라인이 서로의 삭제 동의를 얻어 동기화된 삭제를 수행하는 과정을 나타낸다. 조각 정보 삭제와 관련된 사건은 조각 관리자 및 파이프라인 모두에서 발생하므로 조각 관리자와 파이프라인은 서로에게 삭제 동의를 요청하고 그 결과가 삭제 성공일 때 삭제를 수행한다. (그림 5)의 알고리즘은 조각 관리자가 파이프라인의 삭제 동의를 얻어 조각 정보를 삭제하는 것을 나타낸 것이고, (그림 6)의 알고리즘은 파이프라인이 조각 관리자와의 삭제 동의를 얻어 조각 정보를 삭제하는 것을 나타낸 것이다. 만약 FID로 식별되는 조각 정보 삭제에 대해 상대방의 동의를 얻지 못하면 삭제를 수행하지 않는다. (그림 5)의 1행에서 조각 관리자는 파이프라인에게 동의를 구하기 위한 통보를 한다. 삭제 통보를 받은 파이프라인은 조각 관리자가 삭제 조건이 만족 되었음을 알고, 파이프라인 내의 연산자 간 삭제 동의를 거쳐 조각 정보의 삭제를 시도한다. 만약 파이프라인의 삭제 시도가 성공했다면 통보 결과로 참을 반환하여 파이프라인이 해당 조각 정보에 대한 삭제에 동의한다는 것을 알리고 동의 결과를 확인한 조각 관리자는 자신의 조각 정보를 삭제한다. (그림 6)의 알고리즘은 이와 유사한 과정이 파이프라인에서부터 시작되어 수행되는 것을 나타낸다.

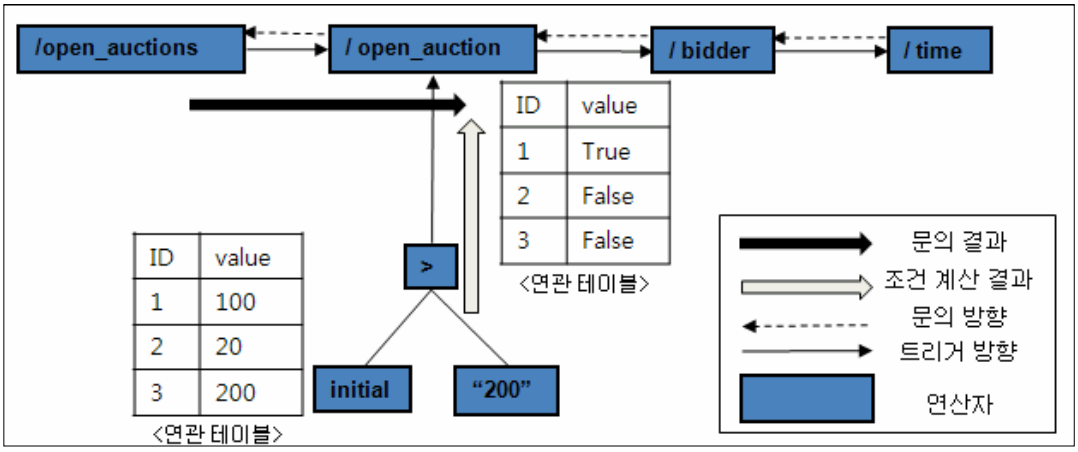
3.2.2 중복된 조건 값 제거

질의 처리 파이프라인에서 각각의 연산자들은 관련된 조각에 대한 정보를 저장한다. 연산자가 조각 정보를 저장하는 목적은 크게 세가지로 나누어지는데, 아래 질의를 예로 들어 분류하면,

```
/site/open_auctions/open_auction[initial>"200"]/bidder/time
```

첫째, open_auction 연산자와 같이 프리디킷을 포함하는 연산자로서 상위 연산자에 대한 문의 결과를 반영한 자신의 계산 결과를 저장하는 경우, 둘째, time 연산자와 같이 질의 처리 파이프라인의 종단 연산자로서 질의의 결과 후보를 저장하는 경우, 셋째 bidder 연산자와 같이 두 경우의 경로 상에 있는 연산자로서 경로 정보 및 상위 연산자에 대한 문의 결과를 저장하는 경우로 나눌 수 있다. 첫째 경우에서 open_auction 연산자는 특정 조각을 처리할 때 이 조각의 조상 조각들의 계산 결과를 자신의 계산 결과에 반영하기 위해 먼저 상위 연산자로 문의를 수행하는데, 문의의 결과가 거짓이거나 결정되지 않음이라면 프리디킷의 계산 결과와 무관하게 이 조각의 값도 거짓이나 결정되지 않음으로 된다. 반면 문의의 결과가 참이라면 이 조각에 대한 자신의 프리디킷 조건을 살펴보고 그 결과를 저장한다. 기존의 연구에서는 문의의 결과와 프리디킷의 조건을 계산하고 문의와 조건의 계산 결과를 반영한 조각 정보를 저장하면서, 프리디킷 조건을 다시 계산할 수 있도록 프리디킷 조건에 대한 조각 정보를 따로 유지한다.

(그림 7)은 문의의 결과가 참인 상황에서 이러한 중복된 조건 값 저장 문제를 나타낸다. (그림 7)의 open_auction 연산자와 조건 연산자는 식별자가 1, 2, 3인 조각에 대한 정보를 각각 저장하고 있는데, 이러한 중복 저장이 반드시 필요한 것은 아니다. 예를 들어 (그림 7)에서처럼 식별자가 1인 조각의 initial 값이 100이었다고 하자. 이 값에 대한 연산을 수행하여 open_auction 연산자에 반영하였다면 조건 연산자에는 더 이상 initial의 값에 대한 정보를 저장할 필요가 없다. 기존 연구에서 이러한 중복 저장을 수행하는 것은 어떠한 조각이 변경 후 재전송되어서 문의 결과가 질의 처리 중간에 변하게 되었을 때의 질의 처리를 고려한 것이나 3.2.1절에서 살펴본 바와 같이 문의의 결과가 변하지 않는다면 이러



(그림 7) 상위 연산자에 반영된 조각 정보의 중복 저장

알고리즘 : removeCondition(중복된 조건값 제거)

입력 : fragmentID(**삭제 대상 조각의 식별자**)

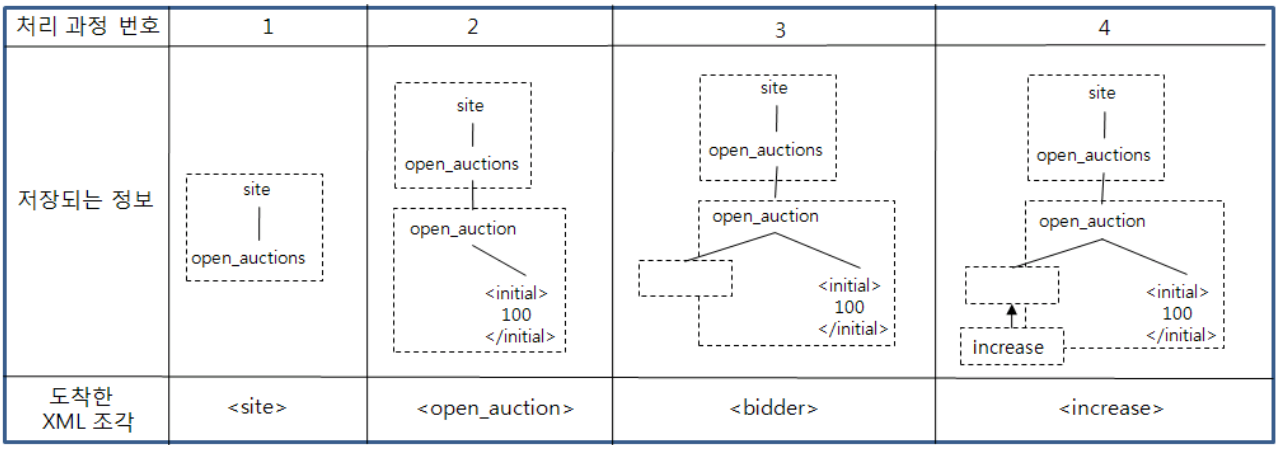
```
   /* 조건 연산자의 상위 연산자의 평가 결과가 변하지 않는 경우 */
1: IF (predecessorOperator.isUpdatable == FALSE) THEN
     /* 조건 연산자의 연관 테이블에서 조각 식별자에 해당하는 정보를 지운다. */
2:   associationTable.remove(fragmentID);
3: END IF
```

(그림 8) 중복된 조건 값 제거 알고리즘

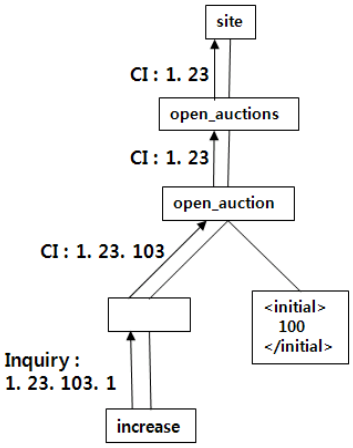
한 중복된 조각 정보를 삭제하는 것이 가능하며, 이것을 **중복된 조건 값 제거**라 부르자. (그림 8)은 중복된 조건 값 제거 알고리즘을 나타낸다. 중복된 조건 값 제거는 XFrag와 XFLab 모두에 적용 가능하다.

3.2.3 무효화된 후손 삭제

(그림 3)의 과정 5에서 도착하는 open_auction들은 initial의 값이 질의의 프리디킷 조건과 일치하지 않는다. 이러한 경우 initial 값이 변하지 않는다면 해당 open_auction의 자식 bidder 조각들은 처리할 필요가 없으며 만약 이미 저장된 bidder 조각 정보가 있다면 삭제가 가능하다. 이렇게 어떤 조각의 조상 조각이 질의의 프리디킷 조건에 의해 거짓으로 결정되었을 때, 해당 조각은 다른 조건을 살펴 볼 필요 없이 질의 결과 대상에서 제외된다. 이와 같이 조상 조각에 의해 질의 결과 대상에서 제외된 조각을 본 논문에서는 **무효화된 후손 조각**이라 하고 이러한 조각에 대한 정보를 삭제하거나, 조각의 처리를 생략하는 것을 **무효화된 후손 삭제**라 부르자. 그러나 위의 예에서 무효화된 후손 삭제에 의해 bidder 조각 정보를 삭제하면 이후 increase가 도착하여 부모 조각에서 필요한 값을 확인하기 위해 상위 연산자로 문의를 시도할 때 부모 조각을 찾을 수 없어 bidder조각이 도착하지 않은 것으로 처리되는 문제가 발생한다. (그림 9)는 이와 같은 문제를 나타낸 것이다. 과정 1-과정 2는 (그림 3)에서 나타낸 것과 동일하다. 과정 3에서 bidder가 도착하지만 부모 조각의 initial 엘리먼트가 조건을 만족하지 않으므로 과정 3의 bidder 조각은 질의 결과와 무관하다. 하지만 bidder 조각을 무효화된 후손 삭제에 의해 저장하고 있을 경우 increase 조각이 도착해서 상위 연산자로 문의를 시도할 때 문제가 발생한다. 파이프라인은 도착한 increase 조각이 질의의 결과로 선택되는지 확인하기 위해, 도착한 increase의 부모 조각까지의 질의 저장 결과를 문의하게 되는데 이때, increase의 부모 조각인 bidder를 처리하지 않았으므로 문의의



(그림 9) 고립 후손 문제



(그림 10) 위임된 문의의 예

알고리즘 : commissionedInquiry(위임된 문의)

입력 : fragmentID(문의를 수행하는 자식 조각의 식별자)
출력 : 참, 거짓, 결정되지 않음(조상 조각의 연산 결과)

```
   /* 자식 조각의 식별자를 부모 조각의 식별자로 변환한다. */
1: fragmentID = convertFragmentIDToParentFragmentID(fragmentID);
   /* 변환된 조각 식별자(부모 조각의 식별자)에 대한 정보를 연관 테이블에서 검색 */
2: tableItem = getAssociationTableItem(fragmentID);
   /* 해당하는 정보가 없는 경우 */
3: IF (tableItem == NULL) THEN
     /* 상위 연산자에게 문의 결과를 위임한다 */
4:   RETURN predecessorOperator.inquiry(fragmentID);
5: ELSE
     /* 계산 결과 정보를 반환한다. */
6:   RETURN tableItem.getReultValue();
7: END IF
```

(그림 11) 위임된 문의 알고리즘

결과는 아직 도착하지 않음으로 판정되어 increase 조각 정보는 이미 버려진 부모 bidder 조각이 도착하기를 기다리게 되는 문제가 발생하는 것이다. 본 논문에서는 이와 같은 문제를 **고립 후손(dangling descendant) 문제**라고 부르며 무효화된 후손 삭제 시 발생하는 이 문제를 해결하기 위해 기존의 문의를 확장한 **위임된 문의(commissioned inquiry)**를 사용한다.

위임된 문의는 문의의 대상으로 부모 조각 뿐 아니라 모든 조상 조각을 대상으로 더 이상 문의할 조상이 없거나, 질의의 프리디킷의 조건에 대한 참, 거짓 값이 결정될 때까지 문의를 수행함으로써 고립 후손 문제를 해결한다. (그림 10)은 위임된 문의의 예를 나타낸 것이다. 위임된 문의는 일반 문의가 실패하였을 때 수행되며 위임된 문의에 필요한 조각 식별자는 상위 연산자에 맞게 적절히 변환된다. 위임된 문의는 레이블링 기법에 의한 조각 간 관계 판별에 의존하므로 XFLab의 경우에만 적용이 가능하고 XFrag에는 적용할 수 없다. (그림 11)은 위임된 문의 알고리즘을 나타낸다.

3.2.4 레이블링 기법을 이용한 질의 경로 단축

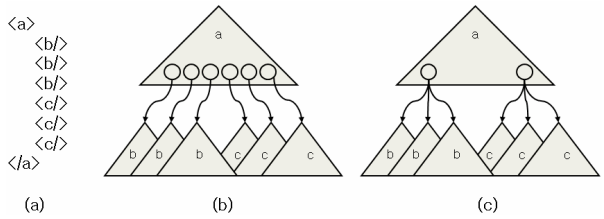
레이블링 기법을 이용하면 조각 간의 부모 자식 관계뿐 아니라 조상 후손 관계까지 파악이 가능하다. 이러한 점을 이용하여 질의 처리에 필요한 조각 정보의 양을 줄일 수 있다. 3.2.2절에서 밝힌 바와 같이 기존 연구에서는 상위 연산자로의 문의 결과와 함께 경로 정보를 유지하기 위해 경로상의 조각 정보를 저장하고 있는데 이는 레이블링 기법을 이용하여 조각 간의 관계를 표현하는 경우 불필요하다. 예를 들어, open_auction, bidder, time이 각각 개별의 조각으로 분할되어 있는 조각 스트림에 대해 3.2.2절에서 예로 든 질의를 처리하기 위해 필요한 조각을 생각해 보자. 먼저 질의 처리의 대상이 되는 open_auction, bidder, time 조각들을 질의가 요구하는 정확한 경로를 만족하는 XML 조각들로 한정시킬 수 있다. 즉, 질의 처리에 대상이 되는 bidder조각은 /site/open_auctions/open_auction/bidder에 위치한 조각이며 마찬가지로 time 조각은 /site/open_auctions/open_auction/bidder/time에 위치하는 조각이다. 만약 open_auction의 후손 중에 /site/open_auctions/open_auction/other/time과 같은 엘리먼트가 있더라도 이 엘리먼트가 포함된 조각은 질의 처리의 대상이 아니며, 이러한 조각은 태그 구조에 명시된 tsid를 이용하여 질의 처리에서 제외시키는 것이 가능하다. 이렇게 선택되어진 open_auction 조각 중 주어진 프리디킷 조건을 만족하는 open_auction 조각을 선택한 후 레이블링 기법을 이용하여 이 open_auction 조각의 후손 조각인 time 조각을 선택하여 질의 처리를 완료할 수 있다.

살펴본 바와 같이 특정한 조건을 만족하는 open_auction의 자식인 bidder의 자식 time 엘리먼트를 포함하는 조각을 bidder 조각 없이 선택하는 것이 가능하다. 이처럼 주어진 질의에 대해 경로의 단축이 가능한지를 미리 살펴보고 질의 처리 파이프라인을 단축하는 것을 **질의 경로 단축**이라 부르자. 질의 경로 단축은 레이블링 기법에 의한 조각 간 관계 판별에 의존하므로 XFLab의 경우에만 적용이 가능하고 XFrag에는 적용할 수 없다.

3.2.5 홀 공유

auction 문서의 bidder 엘리먼트와 같이 계속 반복되어 나타나는 엘리먼트가 다수 포함된 문서를 XFrag의 홀-필러 모델로 분할하게 되면 bidder와 같은 특징을 가지는 엘리먼트의 부모 엘리먼트는 많은 수의 홀 엘리먼트를 포함하게 된다. 특히 bidder와 같은 엘리먼트가 정적으로 확보되지 못한 상태에서 동적으로 계속 추가되는 상황이라면 실제로 추가된 엘리먼트를 포함하는 조각의 전송 뿐만 아니라 그 조각에 대응되는 홀을 포함하는 부모 조각도 재전송해야 한다. 본 논문에서는 홀 필러 모델을 기반으로 하는 XFrag에서 동적으로 생성되는 조각의 효율적 지원을 위해 **홀 공유** 기법을 적용하였다. 반면 XFLab의 경우에는 XML 레이블링을 이용하여 조각 간의 관계를 식별하므로 동적으로 생성되는 조각의 지원에 문제가 없다.

홀 공유 기법은 문서의 특정 위치에 동적으로 추가되는 모든 엘리먼트들에 대한 조각이 정해진 한 홀을 공유하게 하여 한 개의 홀과 다수개의 필러로 조각 간의 관계를 표현하는 방법이다. (그림 12)는 이러한 홀 공유 기법을 나타내고 있다. (그림 12(a))의 문서에 나타난 엘리먼트들을 모두 개별의 조각으로 분할하여 홀-필러 모델로 나타내면 (그림 12(b))와 같은 형태로 된다. 이에 홀 공유 기법을 적용한 결과를 나타낸



(그림 12) 홀 공유 기법의 예

것이 (그림 12(c))이다. 홀 공유 기법을 이용하면 다수의 홀로 인해 조각의 크기가 증가하는 문제를 방지할 수 있고 또 엘리먼트가 추가될 때 부모 조각에 홀 엘리먼트를 추가하여 재전송해야 하는 부담을 줄일 수 있으며, 모든 홀 정보를 저장하기 위해 사용되는 메모리를 절약할 수 있다.

4. 구현 및 성능 평가

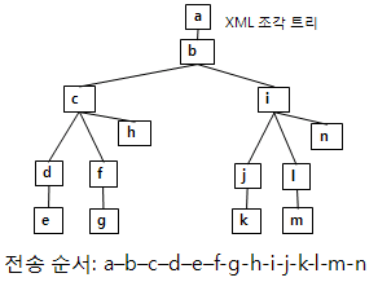
4.1 개요

본 절에서는 본 논문에서 제안한 기법의 구현 및 실험을 통한 기존 기법들과의 비교 성능 평가 결과를 설명한다. 본 논문에서 제안한 삭제 기법으로 확장된 XFrag 및 XFLab을 각각 XFrag-D, XFLab-D라 명명하였다. XFrag-D와 XFLab-D에 적용된 삭제 기법들은 <표 1>에 나타낸 바와 같다. 성능 실험에 사용된 각 기법들은 XFPro에서 제시한 질의 연산자 파이프라인 최적화 알고리즘을 적용하여 구현하였다.

이들 기법들은 J2SE Development kit 5.0 update 11을 사용하는 JAVA 환경에서 구현되었다. 성능 실험은 Windows XP Professional 운영체제에서 Intel의 듀얼 코어 CPU 6600 2.40GHz CPU를 사용하고 메모리는 2GB인 시스템에서 수행하였다. 성능 평가에 사용된 실험 데이터는 XMark 벤치마크[7]의 xmlgen 프로그램으로 생성한 5개의 auction 문서로 크기는 각각 11.3MB, 22.8MB, 34MB, 45.3MB, 56.2MB이다. <표 2>는 홀-필러 모델과 XML 레이블링을 이용하여 이 문서들을 각각 분할한 결과 문서의 크기를 나타낸다. <표 3>은 실험에 사용된 XPath 질의들을 나타낸다. 실제 질의 처리 환경을 시뮬레이션하기 위해 XML 조각을 전송하는 서버를 구현하여 XML 조각을 스트리밍하였으며, 이동 단말기를 시뮬레이션하는 XML 조각 스트림 처리기에서 이를 받아 처리하였다. 스트리밍되는 XML 조각은 분할된 XML 문서를 조각 트리로 구성했을 때 이 트리를 preorder로 탐색하는 순서로 전송하였다. (그림 13)은 이와 같은 전송 순서의 예를 나타낸다. 본 실험에서는 성능 척도로 XML 조각 스트림에 대한 질의 처리에 사용된 메모리 양 및 질의 처리 시간을 고려하였다. 메모리 사용량은 XML 조각 스트림 처리기가 질의 처리 시 사용한 최대 메모리 사용량을 측정하였다. 질의 처리 시간은 서버와의 통신 시간을 제외한 XML 조각 처리 시간만을 측정하였다.

〈표 3〉 실험에 사용된 XPath 질의

질의 번호	질의
Q1	/site/open_auctions/open_auction[initial>"200"]/bidder/time
Q2	/site/open_auctions/open_auction/bidder[increase>"200"]/time
Q3	/site/people/person[name="Claudine Nunn"]/watches/watch
Q4	/site/people/person[name="Claudine Nunn"]//watch
Q5	/site/people/person[name="Torkel Prodromidis"]/profile/interest
Q6	/site/people/person[name="Torkel Prodromidis"]//interest
Q7	/site/open_auctions/open_auction[initial>"200"]/interval/start
Q8	/site/open_auctions/open_auction[initial>"500"]/bidder[increase>"200"]/time
Q9	/site/closed_auctions/closed_auction[price>"100"]/type
Q10	/site/closed_auctions/closed_auction[price>"200"]/annotation/author



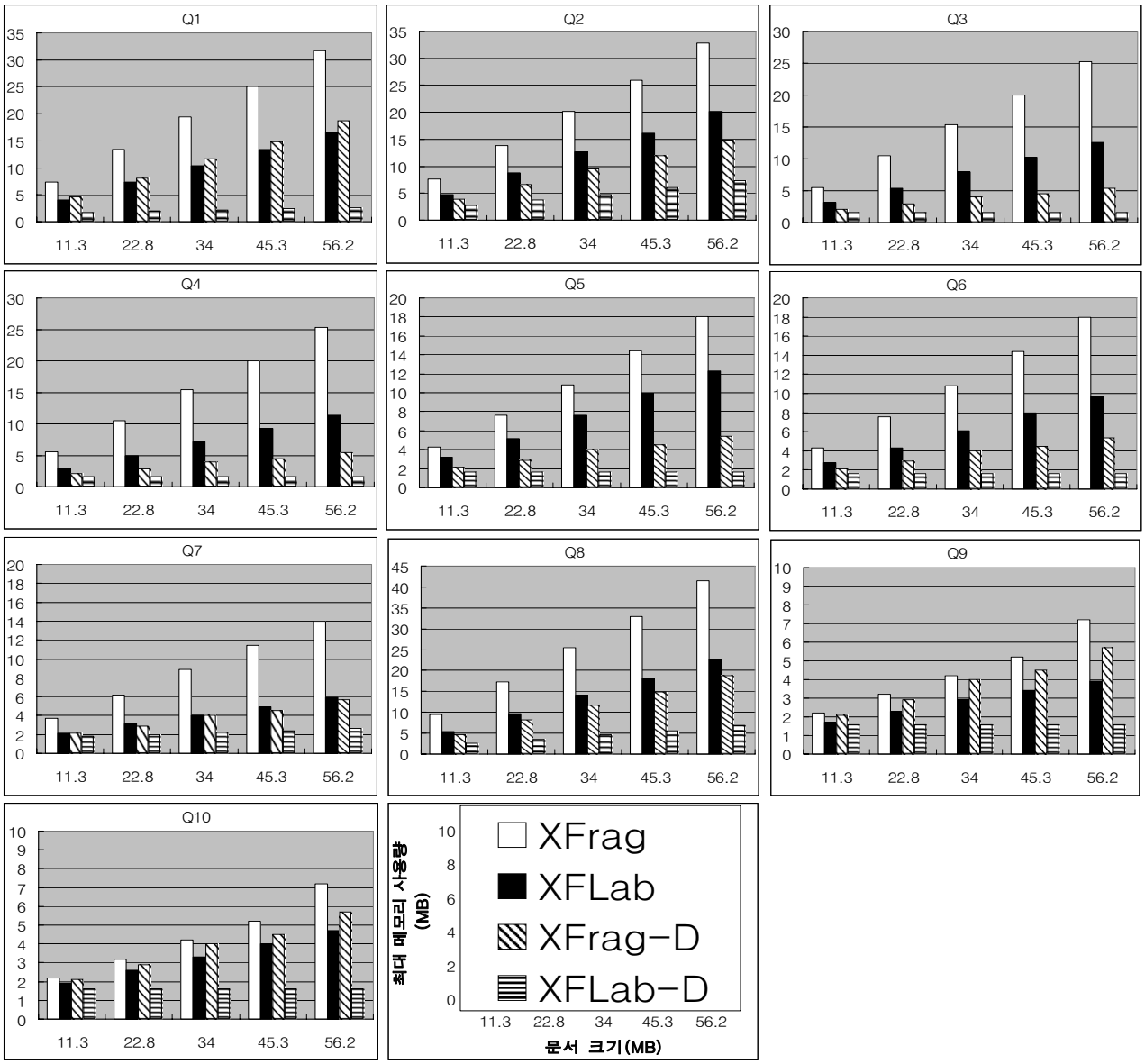
(그림 13) 실험에 사용된 XML 조각 전송 순서

4.2 실험 결과

(그림 14)는 기존의 XFrag와 XFLab 및 본 논문에서 제안한 XFrag-D와 XFLab-D로 다양한 크기의 XML 문서를 분할한 조각 스트림에 대하여 질의 처리를 수행하는 과정에서 측정한 최대 메모리 사용량을 질의 별로 나타낸 것이다. 그래프의 x축은 분할 전 문서 크기를 MB 단위로, y축은 해당 문서를 처리하는 데 사용된 최대 메모리의 양을 MB 단위로 나타낸 것이다. 실험 결과를 보면 XFrag-D와 XFLab-D가 각각 기존의 XFrag, XFLab보다 XML 문서 크기에 대한 확장성 면에서 월등히 우수함을 알 수 있다. 그러나 Q1, Q9, Q10의 질의의 경우에는 제안하는 기법인 XFrag-D보다 기존 XFLab의 메모리 사용량이 더 적었다. 이는 무엇보다도 XFrag-D의 기반

〈표 2〉 분할 방법에 따른 결과 문서의 크기

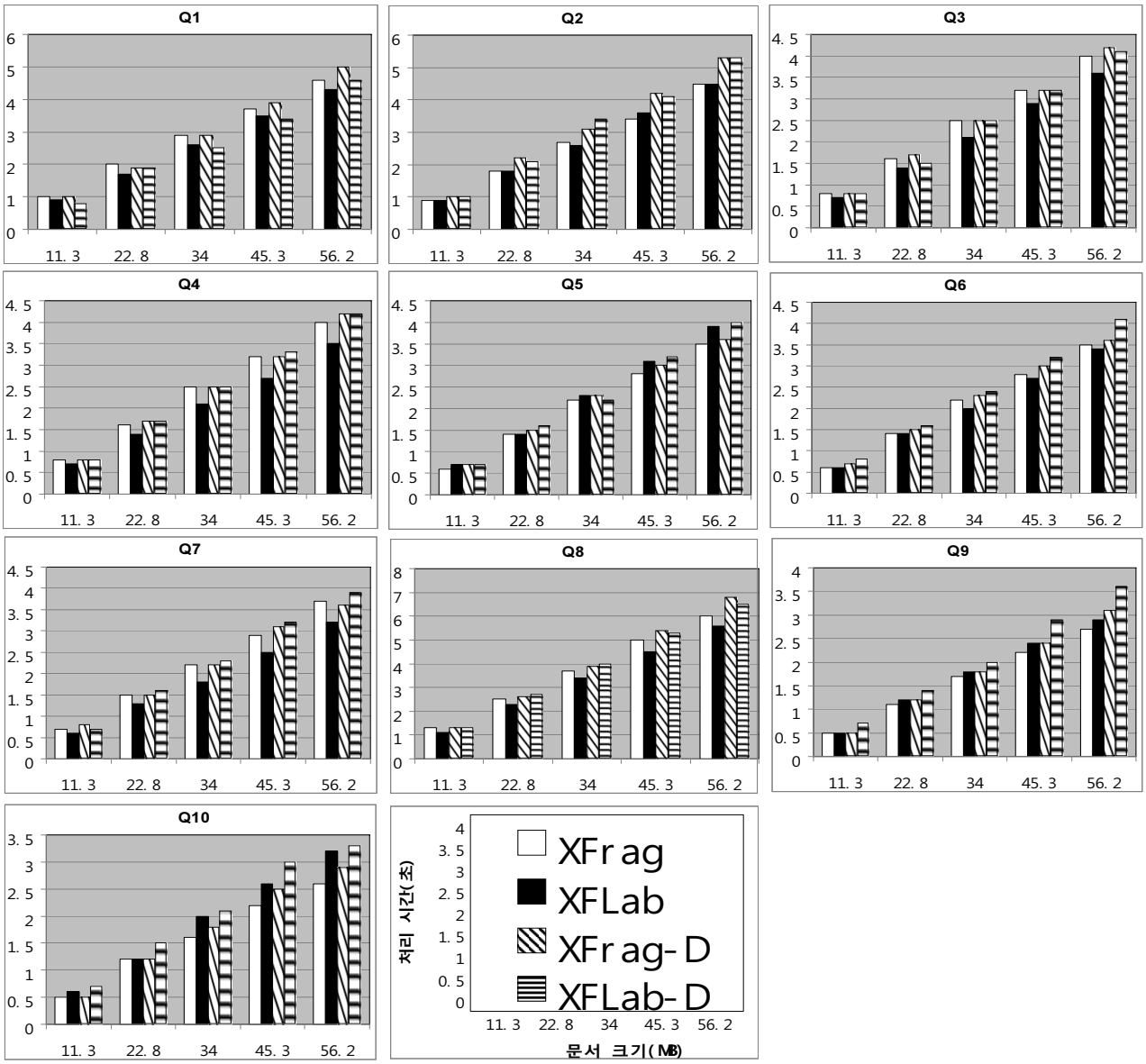
원본문서 분할방법 / 크기	11.3MB	22.8MB	34MB	45.3MB	56.2MB
XML 레이블링을 이용한 분할	13.3MB	26.7MB	39.9MB	53.2MB	66.1MB
홀 필러 모델을 이용한 분할	14.3MB	28.9MB	43.2MB	57.5MB	71.6MB



(그림 14) 기존 기법과 삭제 기능으로 확장된 기법의 메모리 사용량 비교

이 되는 XFrag의 메모리 효율이 XFLab보다 현저히 낮다는 점에서 그 근본적인 원인을 찾을 수 있다[6]. 또한 Q1에 대해서는 홀 공유 기법은 적용 가능했지만 다른 삭제 기법의 적용이 불가능했고, Q9, Q10의 경우에는 홀 공유 기법을 적용할 수 없었기 때문에 조각 내 홀 수가 증가하고 그 결과 조각의 크기가 커져 XFrag 대비 메모리 효율 제고에 한계가 있었다. 본 절의 나머지 부분에서는 실험 결과에 대해 각 삭제 기법 별로 분석하고 설명한다.

XFrag-D의 경우 홀 공유 기법을 통하여 기존의 XFrag보다 우수한 성능을 보이게 되는데 이것은 Q1과 Q8에서 확인할 수 있다. 이 두 질의는 그 경로 상에 bidder 엘리먼트를 포함하고 있다. bidder는 auction 문서에서 반복적으로 나타나며 각 경매 아이템에 대한 가격 제시가 있을 때마다 생긴다. 각 경매를 나타내는 open_auction 엘리먼트마다 가격 제시를 나타내는 복수개의 bidder 엘리먼트가 존재하므로 문서 전체적으로는 open_auction 엘리먼트의 수에 비례하여 그 총수가 매우 크다. (예를 들어, 56.2MB 문서의 경우, open_auction 엘리먼트의 총수가 6,000개이고 bidder 엘리먼트의 총수는 29,629개이다. 즉, open_auction 당 평균 4.94개의 bidder 엘리먼트가 존재한다.) 따라서 질의 경로 상에 bidder를 포함하게 되면 XFrag의 경우 각 open_auction 조각에 대해 자식 조각인 bidder 조각의 수만큼 홀을 생성하고, 질의 처리 과정에서 이러한 홀 정보를 유지해야 하므로 메모리 사용량이 급격히 증가하게 된다. 본 실험에서는 동적으로 추가되는 bidder 엘리먼트의 처리를 위해 XFrag-D에서 bidder 엘리먼트에 대해 하나의 홀을 공유할 수 있도록 하였다. 따라서 XFrag-D는 각 open_auction 조각에 대해 최대 1개의 홀만 가지게 되므로 bidder 엘리먼트의 개수 증가



(그림 15) 기존 기법과 삭제 기능으로 확장된 기법의 질의 처리 시간 비교

에 따른 메모리 사용량의 증가가 높지 않음을 볼 수 있다.

XFrag-D는 홀의 개수를 셈으로써 질의 처리 과정에서 불필요한 정보를 삭제하여 메모리 사용량을 감소시킬 수 있다. Q3, Q4, Q5, Q6, Q9, Q10에 대한 실험 결과에서 이 기법의 효율성을 확인할 수 있다. 위에서 언급한 질의들은 질의 경로 상에 동적인 엘리먼트를 포함하지 않고 있다. 따라서 이들 질의를 처리하는 데 필요한 조각들은 모두 자식 조각의 개수만큼의 홀을 포함하고 있으며, 홀의 개수(자식 조각의 개수)는 질의 처리시 변하지 않는다. XML 조각은 위에서 설명한 preorder 순서로 전송되기 때문에, 질의 처리기는 부모 조각과 자식 조각을 순차적으로 받아 처리하게 되고, 자식 조각을 다 받게 되면 부모 조각의 정보를 질의 연산자 파이프라인의 연관 테이블에서 삭제한다.예를 들어 Q3의 경우, site 조각이 전송되고 이후에 person 조각이 전송된다. 연이어 이 조각의 자식에 해당하는 watch 조각들이 전송된다. person의 자식에 해당하는 watch 조각이 모두 전송되었다면 질의 처리기는 person 조각의 자식 조각이 더 이상 전송되지 않을 것임을 알 수 있기 때문에 person 조각에 대한 정보를 연관 테이블에서 삭제한다. 따라서 질의 처리 중 연관 테이블에 저장되는 person 조각에 대한 정보는 최대 1개이며, 이것은 문서의 크기와는 무관하게 적용된다. 따라서 XPath 질의 경로 상에 동적인 엘리먼트를 포함하지 않는 질의는 문서의 크기 증가에 따른 누적된 조각 정보의 증가가 없다. 문서 크기 증가에 따른 메모리 사용량이 조금씩 증가하는 이유는 문서 크기 증가에 따라 홀 개수의 증가로 인한 것이다. 홀의 개수가 증가하게 되면, 이러한 홀을 포함하는 XML 조각의 크기 역시 커지게 되며, 질의 처리기가 이 조각을 전송 받는 데 사용되는 메모리의 양이 증가하

〈표 4〉 기존 기법과의 성능 비교

(a) 기존 기법에 대한 메모리 사용량 비교

항목 / 문서 크기(MB)	XFrag-D의 개선 비율(%)	XFLab-D의 개선 비율(%)
11.3	44.01	36.74
22.8	48.66	54.84
34	48.62	64.13
45.3	52.71	69.08
56.2	54.37	72.5
평균	49.67	59.46

(b) 기존 기법에 대한 처리 시간 비교

항목 / 문서 크기(MB)	XFrag-D의 시간 증가 비율(%)	XFLab-D의 시간 증가 비율(%)
11.3	6.77	15.98
22.8	9.72	15.48
34	5.83	17.64
45.3	8.19	16.69
56.2	6.90	14.28
평균	7.48	16.01

게 된다. XFrag-D는 위에서 언급한 기법들 이외에도 중복된 조건 값 제거 기법을 이용하여 모든 질의에 대해서 메모리 사용량을 절약하였다.

XFLab-D의 질의 경로 단축 기법의 효과는 Q1과 Q7에서 볼 수 있다. 언급한 두 질의의 경우, XFLab-D는 문서 크기 증가에 따른 최대 메모리 사용량이 거의 증가하지 않는다. 이것은 질의 경로 단축 기법에 의하여 Q1의 경우 bidder, Q7의 경우 interval에 해당하는 연산자가 생략되기 때문이다. 결과적으로 두 질의를 처리할 때, 저장되는 정보는 open_auction 조각에 대한 정보뿐이며, open_auction 조각은 bidder 조각에 비해서 문서 크기 증가에 따른 개수 증가가 크지 않다 (예를 들어, 56.2MB 문서의 경우 6,000개). 결과적으로 문서 크기 증가에 따른 메모리 사용량의 증가가 크지 않음을 볼 수 있다.

XFLab-D의 자식 수 세기 기법의 효과는 XFrag-D와 동일하게 Q3, Q4, Q5, Q6, Q9, Q10에서 나타난다. XFLab-D도 XFrag-D와 마찬가지로 정적인 엘리먼트에 대해서 자식 수를 셀 수 있기 때문에 연관 테이블에 저장되는 정보를 최소화하면서 질의 처리를 수행하게 된다. 하지만 XFrag-D는 문서 크기의 증가에 따라 메모리 사용량이 증가하는데 반해, XFLab-D는 최대 메모리 사용량이 문서의 크기에 무관하게 항상 일정함을 볼 수 있다. 이는 XFLab-D가 XML 레이블링에 의해 분할된 조각에 대해 질의 처리를 수행하기 때문이다. XML 레이블링을 사용하여 분할된 XML 조각은 자식 조각의 개수가 증가하여도 부모 조각의 크기에는 변화가 없다(부모 조각에 홀과 같은 부가 정보를 만들지 않기 때문). 따라서 XFLab-D의 경우 최대 메모리 사용량이 문서의 크기와 무관하게 항상 일정하게 유지되며 XFrag-D에 비해서 문서 크기에 따른 확장성 면에서 우수하다.

(그림 15)는 동일한 실험에서 질의 처리 시간을 질의 별로 나타낸 것이다. 그래프의 x축은 분할 전 문서 크기를 MB 단위로, y축은 해당 문서를 처리하는 데 소요된 시간을 초 단위로 나타내었다. XFrag-D와 XFLab-D의 경우 전반적으로 처리 시간이 기존 기법에 비해 증가하였음을 볼 수 있다. 처리 시간의 증가는 자식 조각의 개수를 세어서 처리하는 부분과 조각 식별자를 변환하는 등의 조각 정보 삭제를 위한 부가 작업들로 인해 발생한 것이다. 하지만 메모리 사용량에서 얻는 효율과 비교하였을 때 처리 시간의 증가는 크지 않다. <표 4(a)>는 XFrag와 XFLab의 메모리 사용량에 비해 XFrag-D와 XFLab-D의 메모리 사용량이 얼마나 감소하였는가를, <표 4(b)>는 XFrag와 XFLab의 질의 처리 시간에 비해 XFrag-D와 XFLab-D의 질의 처리 시간이 얼마나 증가하였는가를 각각 백분율로 나타낸 것이다. <표 4>에 나타낸 값은 모든 질의에 대해 측정한 값을 문서 크기별로 분류하여 평균한 값을 구한 것이다. <표 4(a)>에 의하면 메모리 사용량은 XFrag-D가 평균 50% 가량, XFLab-D가 평균 60% 가량을 개선한 것으로 나타났다. 특히 문서가 커짐에 따라 메모리 사용량 개선의 정도가 커지는데, 이것은 기존의 기법에 비해 문서 크기에 대한 확장성이 개선되었다는 것을 의미한다. 이에 비해 <표 4(b)>에 의하면 질의 처리 시간은 XFrag-D가 평균 7.5% 가량, XFLab-D가 평균 16% 가량 늘어난 것으로 나타났다. 그러나 이는 메모리 사용량이 확연히 줄어든 것과 비교하여 훨씬 적은 부담이라 하겠다. XFLab-D가 56.2MB 크기의 문서를 대상으로 질의 처리한 결과를 예로 들면, XFLab에 비해 메모리 사용량의 감소는 72%, 질의 처리 시간의 증가는 14%이다. 이는 본 논문에서 제시한 삭제 기법을 통한 질의 처리 시간 증가와 메모리 효율 제고의 트레이드오프가 충분히 가치 있다는 것을 의미한다. 또한 <표 4>를 보면 메모리 사용량의 감소 폭은 문서의 크기가 커질수록 증가하지만 질의 처리 시간의 증가 폭은 문서의 크기와 무관하게 일정한 편이라는 점을 알 수 있다.

실험 결과를 요약하면 다음과 같다. 본 논문에서 제시한 누적된 조각 정보 삭제 기법들을 통해 확장된 XFrag-D 및 XFLab-D 기법은 기존의 XFrag 및 XFLab에 비해 문서의 크기 증가에 따른 확장성 있는 메모리 효율적 질의 처리를 수행한다. 특히, XML 레이블링을 활용한 XFLab-D가 홀 필러 모델을 사용한 XFrag-D에 비해서 메모리 효율성이 더 뛰어나며 문서 크기에 따른 확장성 또한 현저히 높은 것으로 나타났다.

5. 결 론

본 논문에서는 XML 조각 스트림에 대한 메모리 효율적인

질의 처리를 수행하는 데 있어 XML 문서 크기에 따른 확장성을 제공하기 위해 누적된 조각 정보 삭제 기법들을 제시하였으며 이러한 조각 정보 삭제 기법들을 이용하여 기존의 XFrag 및 XFLab 기법을 확장한 XFrag-D 및 XFLab-D 기법을 제시하고 기존의 XFrag 및 XFLab 기법과 성능을 비교 평가하였다.

XFrag-D와 XFLab-D에서는 기존의 기법들에서 질의 처리가 끝나거나 질의 결과와 무관한 XML 조각 정보를 버리지 못하는 문제를 누적된 조각 정보 삭제 기법들을 이용하여 해결함으로써 메모리 사용 효율을 제고하였다. 제시된 삭제 기법은 모두 다섯가지이다 (3.2절 <표 1> 참조). 첫째, **자식 수 세기**는 주어진 조각의 자식 조각들이 정적일 때 자식 조각이 모두 도착하였는지 확인하여 모두 도착하였다면 해당 조각에 대한 질의 처리가 종료됨과 동시에 조각 정보를 삭제하는 것이다. 이미 처리가 이루어진 조각에 대한 조각 정보 삭제를 수행함으로써 기존의 기법들이 문서의 크기가 증가함에 따라 메모리 사용량이 크게 증가하는 문제를 해결하였다. 둘째, **중복된 조건 값 제거**는 파이프라인 내의 여러 연산자에 중복된 조각 정보를 삭제하는 기법이다. 셋째, **무효화된 후손 삭제**는 조상 엘리먼트에서 질의 프리디킷 조건을 만족하지 않을 때 이 엘리먼트를 포함하는 조각의 후손 조각들에 대한 정보를 저장하지 않음으로써 메모리 효율을 제고한다. 넷째, **질의 경로 단축**은 XML 레이블링 기법이 가진 조각 간 관계 표현의 장점을 기반으로 경로 상의 조각 처리를 생략함으로써 역시 메모리 효율을 제고한다. 다섯째, **홀 공유**는 홀-필러 모델의 경우 동적으로 생성되는 조각들에 대해 단 하나의 홀만 할당하여 필러 조각들에 대응시킴으로써 메모리 효율을 제고한다.

구현 및 실험을 수행한 결과 본 논문에서 제시한 조각 정보 삭제 기법을 통해 확장된 기법들이 기존의 기법들보다 메모리 효율면에서 우수하고, 따라서 XML 문서 크기의 증가에 따른 확장성 있는 처리를 수행할 수 있음을 확인하였다. 특히 XFrag-D에 비하여 XFLab-D의 경우 XML 레이블링을 이용한 조각 간 관계 표현을 이용함으로써 질의 연산자 파이프라인에 조각에 대한 더 많은 정보를 제공할 수 있었고 이런 부가 정보를 이용하여 더 많은 조각 정보를 삭제할 수 있었다. 이로 인해 XFLab-D가 XFrag-D에 비해 더욱 우수한 메모리 효율성 및 확장성을 보인다는 것 또한 확인하였다.

향후 연구 과제는 다음과 같다. XML 데이터 분할 과정에서 파생되는 정보를 클라이언트 단말기에서의 질의 처리 과정에서 활용하여 메모리 및 처리 효율을 제고하기 위한 기법의 연구가 필요하다. 또한 XML 레이블링 기법을 확장하여 XML 조각 간 구조 관계 표현뿐만 아니라 단말기에서의 질의 처리에 활용할 수 있는 부가 정보도 제공하게 하여 단말기에서의 처리 효율을 제고하기 위한 연구가 필요하다.

참 고 문 헌

[1] Sujoe Bose, Leonidas Fegaras, David Levine and Vamsi Chaluvadi, "A Query Algebra for Fragmented XML Stream Data," DBPL 2003.

[2] "XML Fragment Interchange," W3C Candidate Recommendation 2001.

[3] Leonidas Fegaras, David Levine, Sujoe bose and Vamsi Chaluvadi, "Query Processing of Streamed XML Data," CIKM 2002, pp.126-133.

[4] Sujoe Bose and Leonidas Fegaras, "XFrag: A Query Processing Framework for Fragmented XML Data," Web and Databases 2005.

[5] Huan Huo, Guoren Wang, Xiaoyun Hui, Rui Zhou, Bo Ning, and Chuan Xiao, "Effiecient Query Processing for Streamed XML Fragments," DASFAA 2006.

[6] 이상욱, 김진, 강현철, "XML 레이블링 기법을 이용한 XML 조각 스트림에 대한 질의 처리," 한국정보과학회 추계학술대회 2006, pp.113-117.

[7] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu and Ralph Busse. "XMark: A Benchmark for XML Data Management," VLDB 2002, pp. 974-985.

[8] http://lambda.uta.edu/XStreamCast/

[9] Laurent Mignet, Denilson Barbosa, Pierangelo Veltri, "The XML Web: a First Study," WWW 2003.

[10] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, Chun Zhang, "Storing and Querying Ordered XML Using a Relational Databse System," SIGMOD 2002.

이 상 욱

e-mail : swlee@dblab.cse.cau.ac.kr
2006년 중앙대학교 컴퓨터공학과(학사)
2006년 ~ 현 재 중앙대학교 대학원 컴퓨터공학과 석사과정
관심분야 : XML 데이터베이스, XML 스트림 데이터 처리 등

김 진

e-mail : jkim@dblab.cse.cau.ac.kr
2006년 중앙대학교 컴퓨터공학과(학사)
2006년 ~ 현 재 중앙대학교 대학원 컴퓨터공학과 석사과정
관심분야 : XML 스트림 데이터 처리, 웹 데이터베이스 등

강 현 철

e-mail : hckang@cau.ac.kr

1983년 서울대학교 컴퓨터공학과(공학사)

1985년 U. of Maryland at College Park, Computer Science(M.S.)

1987년 U. of Maryland at College Park, Computer Science(Ph.D.)

1988년 ~ 현 재 중앙대학교 컴퓨터공학부 교수

관심분야 : XML 및 웹 데이터베이스, Stream 데이터 관리, 센서 네트워크 데이터베이스 등