DOI: 10.3745/KIPSTD.2009.16D.6.835

임베디드 데이터베이스 시스템을 위한 블록 단위 스키핑 기법

정 재 혁† · 박 형 민†† · 홍 석 진††† · 심 규 석††††

요 약

일반적으로 데이터베이스 시스템에서의 질의 수행은 대부분의 경우 빠른 응답시간과 더 적은 메모리 사용량을 장점으로 가지는 파이프라이닝 기법으로 이루어진다. 이 때, 질의 수행 계획(QEP)의 각각의 연산 노드들은 Open(), Next(), Close() 함수들을 지원하는 iterator의 인터페이스를 가진다. 그런데, 플래시 메모리 기반의 휴대용 기기들을 위한 임베디드 데이터베이스 시스템에서는 iterator의 Next() 함수뿐만 아니라, 현재 레코드의 이전 레코드를 리턴해주는 Previous()와 같은 함수를 필요로 하는 경우가 많다. 이는 임베디드 환경의 경우 각각의 프로그램이 사용할 수 있는 메모리의 양이 제한적이므로, 사용자가 이전 레코드를 요청하는 경우, 결과 레코드 커서가 현재 레코드를 기준으로 이전 레코드를 다시 가져와야 하기 때문이다. 본 논문에서는 이러한 임베디드 데이터베이스 시스템의 질의 수행 시 각각의 연산 노드들이 Next() 함수뿐만 아니라 Previous() 함수를 블록 단위로 지원할 수 있도록 새롭게 설계 구현하는 과정에서 발생하는 방향 전환 문제를 소개하고 이를 해결하기 위한 블록 단위 스키핑 기법을 제안한다.

키워드 : 파이프라이닝, 임베디드 데이터베이스, 질의 처리기

Block-wise Skipping for Embedded Database System

Jaehyok Chong† · Hyoungmin Park†† · Seokjin Hong††† · Kyuseok Shim††††

ABSTRACT

Today, most of all the query processors in the world generally use the 'Pipelining' method to acquire fast response time (first record latency) and less memory usage. Each of the operator nodes in the Query Execution Plan (QEP) provides Open(), Next(), and Close() functions for their interface to facilitate the iterator mechanism. However, the embedded database systems for the mobile devices, based on the FLASH memory, usually require a function like Previous(), which returns the previous records from current position. It is because that, in the embedded environment, the mobile devices cannot fully provide it main memory to store all the query results. So, whenever needed the previously read records the user (program) should re-fetch the previous records using the Previous() function: the BACKWARD data fetch. In this paper, I introduce the 'Direction Switching Problem' caused by the Previous() function and suggest 'Block-wise Skipping' method to fully utilize the benefits of the block-based data transfer mechanism, which is widely accepted by most of the today's relational database management systems.

Keywords : Pipelining, Embedded Database, Query Processor

1. 서 론

오늘날 PDA나 MP3 플레이어, 휴대폰, 디지털 카메라와 같은 이동형 기기들을 위한 저장매체로서 플래시 메모리가 점점 더 많은 사람들의 관심을 받고 있다. 이는 플래시 메모리의 다음과 같은 장점들 때문이다. 첫째, 하드디스크보다 훨씬 낮은 전력을 필요로 한다. 둘째, 일반적인 메모리와는 달리 전원이 없어도 기록된 데이터를 보존할 수 있다. 셋째, 기계적인 메커니즘을 사용하지 않으므로 우수한 퍼포먼스와 안정성을 보장한다. 넷째, 훨씬 작고 가볍게 만들 수 있어 탁월한 이동성을 보여준다. 특히, 최근에는 다수의 NAND 타입 플래시 메모리를 병렬로 연결하여 SSD (Solid State Disk)라는 대용량 데이터 저장 매체까지 나와 기존의 하드디스크를 대체할 대상으로 주목받고 있다[5, 8].

그러나 플래시 메모리는 다음과 같은 한계도 가지고 있다. 플래시 메모리는 빠른 읽기연산속도에 비해 상대적으로 느린 쓰기연산속도와 훨씬 더 느린 소거연산속도를 가지고 있다. 특히 소거연산의 경우 그 정확한 동작을 보장하는 회수의 제한이라는 것이 있어서, 일반적으로 100,000번이라는

※ 본 연구는 2007년도 삼성전자의 연구비 지원 프로젝트의 일부로 수행하였습니다.
† 정 회 원 : SAP R&D Center Korea Senior Developer
†† 준 회 원 : 서울대학교 전기컴퓨터공학부 박사과정
††† 정 회 원 : 삼성전자 종합기술원 전문연구원
†††† 정 회 원 : 서울대학교 전기컴퓨터공학부 교수
논문접수 : 2009년 5월 12일
수 정 일 : 1차 2009년 6월 18일, 2차 2009년 7월 27일
심사완료 : 2009년 8월 13일

소거연산회수를 지원하며, 그 이후에는 정확한 동작을 보장하지 못한다[3]. 이러한 문제들을 해결하기 위해 플래시 메모리를 대상으로 하는 데이터베이스 색인에 대한 연구들[7-8]이 있었고, 쓰기/소거연산을 최소화하기 위한 파일시스템에 대한 연구[4, 11]가 있었다. 그리고, 트랜잭션처리에 대한 연구들[1-2, 13-14]도 꽤 진행되었지만, 아직까지 질의 최적화나 질의 처리에 관련된 연구는 그다지 많이 진행되지 않고 있다[9, 12].

임베디드 데이터베이스 시스템의 질의 처리기나 질의 최적화기는 플래시 메모리의 특징들 즉, 비대칭적인 읽기/쓰기 연산의 속도 차이와 기계적 메커니즘에 따른 응답시간지연 현상이 없는 점들 때문에 일반적인 디스크기반 데이터베이스 시스템과는 다른 특성을 가진다. 특히 가용할 수 있는 메모리(RAM)가 부족한 휴대용 기기들의 경우에는 그 차이가 더 심해진다. 예를 들면 디스크기반 데이터베이스 시스템의 경우 Hash Join이나 Sort-Merge Join이 Nested-Loop Join보다 훨씬 좋은 성능을 보이지만, 메모리(RAM)가 부족하고, 읽기 연산에 비해 쓰기 연산이 훨씬 느린 플래시 메모리를 데이터 저장의 용도로 사용하는 휴대용기기들의 경우에는 오히려 Nested-Loop Join이 훨씬 좋은 성능을 보인다. 왜냐하면 Hash-Join과 Sort-Merge Join 둘 다 충분하지 않은 메모리 때문에 hashing이나 sorting을 위해 플래시 메모리에 쓰기 연산을 수행해야 하기 때문이다[6].

더하여, 일반적으로 메모리(RAM)가 충분하지 못하므로 질의 결과가 많을 경우 모두 메모리에 올려놓지 못한다. 그래서 그때그때 필요한 데이터를 데이터베이스 시스템에게 요청하여 처리해야 한다. 즉. 디스크기반 데이터베이스 시스템처럼 현재 읽고 있는 레코드를 기준으로 항상 다음 레코드만 요청하는 것이 아니라 이전 레코드가 필요한 경우에는 이전 레코드를 다시 요청하여 받을 수 있어야 한다. 예를 들어 MP3 플레이어에 10000곡의 음악이 저장되어 있고, 사용자가 지정한 임의의 검색 조건을 통해 사용자에게 5000곡의 리스트를 보여준다고 가정해 보자. 일반적으로 MP3 플레이어가 한 화면에 보여줄 수 있는 곡의 개수가 10곡이라고 할 때 전체 리스트를 보여주기 위한 인터페이스로 제공되는 것은 '위'/'아래' 버튼을 이용하여 이전/다음 10곡의 리스트를 보여주는 것이다. 사용자가 리스트를 탐색하다가 이미 지나간 이전의 10곡을 다시 보기 위해 '위' 버튼을 누를 경우, 5000곡의 검색 결과를 모두 메모리에 저장하고 있지 않다면, (1) 처음부터 다시 질의를 수행하여 이전 10곡의 리스트를 가져올 때까지 질의를 계속 수행하는 것보다, (2) 현재 위치에서 이전 10곡의 리스트를 가져오는 방법이 훨씬 효율적일 것이다. 결국 임베디드 데이터베이스 시스템은 현재 커서가 가리키는 레코드의 다음 레코드가 아닌 이전 레코드를 요청할 경우에도 처리해 주어야 하는 것이다.

이렇게, 이전 레코드 요청을 처리해 줘야 하는 플래시 메모리 기반 임베디드 데이터베이스 시스템의 질의 처리기의 경우 기존의 디스크기반 데이터베이스 시스템의 질의 처리기와는 달리 질의 수행 계획(Query Execution Plan)의 각 노드들 간의 데이터 전송이 레코드 단위로 이루어지는 경우가 많다. 이는 사용자가 이전 레코드를 요청할 경우 말단 노드에서 현재 읽고 있는 레코드의 이전 레코드를 읽어서 부모노드로 전해주기만 하면 되므로 질의 처리기 구현이 쉽기 때문이다.

하지만, 레코드 단위의 데이터 전송은 질의 수행 시 질의 수행 계획상에서 불필요하게 많은 함수 호출이 일어날 수밖에 없고, 조인 연산을 수행함에 있어서도 인덱스가 없는 경우 Nested-Loop 조인을 수행해야 하는데, 훨씬 효율적인 블록 Nested-Loop 조인을 이용할 수 없다는 단점이 있다.

기존의 데이터베이스 시스템에서는 이러한 이전 질의 결과를 다시 요청하는 문제에 대한 인식이 없었고 따라서 문제 해결을 위한 연구가 없었다. 이에 대한 본 논문의 기여는 다음과 같다.

- 레코드 단위가 아닌 블록 단위로 이루어지는 질의 수행 계획의 각 노드간의 데이터 전송에 대해, 질의 수행 중에 데이터 전송의 방향이 바뀌는 것을 지원할 수 있도록 고려해야 할 점들을 제시하고, 그 해법을 제시한다.
- 실제로 삼성전자의 임베디드 데이터베이스 시스템인 AceDB[12-14]를 기반으로 실험해 본 결과 상당한 성능 향상을 관찰할 수 있었다. 이는 본 논문이 실제 업계에서 사용되고 있는 임베디드 데이터베이스 시스템의 질의 수행 성능에 많은 기여를 할 수 있음을 보여준다.

앞으로 이어질 내용의 구성은 다음과 같다. 2장에서는 레코드 기반 파이프라이닝 기법과 블록 기반 파이프 라이닝 기법에 대해 간략히 살펴보겠다. 3장에서는 역방향 데이터 전송과 블록 기반 데이터 전송을 동시에 처리하기 위해 풀어야 할 문제에 대해 간략히 살펴보고, 4장에서는 해결 방법으로 블록 단위 스키핑 기법을 소개한다. 5장에서는 여러 시나리오에 따른 레코드 단위의 데이터 전송과 블록 단위의 데이터 전송의 성능을 비교한다. 마지막으로, 6장에서는 결론과 향후 연구 과제를 제시한다.

2. 기본 개념

데이터베이스 시스템에서 질의를 수행하는 방법에는 실체화 기법(materialization)과 파이프라이닝 기법(pipelining)의 두 가지 방법이 있다. 실체화 기법에 비해 파이프라이닝 기법은 임시 테이블을 만들지 않으므로 더 적은 메모리로 질의를 수행할 수 있고, 첫 번째 레코드를 리턴해 주는 응답시간이 매우 짧다는 장점이 있다[10]. 그러므로 휴대용 기기를 대상으로 하는 임베디드 데이터베이스 시스템을 구현할 때 많이 이용한다. 이번 장에서는 이 파이프라이닝에 대해 잠시 알아보겠다.

2.1 파이프라이닝 (Pipelining)

일반적으로 질의 수행 계획이 파이프라이닝 기법을 지원

할 수 있도록 구현하기 위해 각각의 연산들이 Open(), Next(), Close()를 지원하는 Iterator 인터페이스를 가지게 한다. 각각의 연산들은 질의 수행 계획을 구성하는 트리에서 하나의 노드에 대응된다. 이 노드들 사이에 데이터를 전송할 경우 하나의 레코드씩 처리해서 전송하는 방법과 여러 레코드를 하나의 블록으로 처리해서 (버퍼를 이용하여) 전송하는 방법이 있다.

2.1.1 레코드 기반 파이프라이닝

먼저 레코드 기반 파이프라이닝의 경우를 살펴보자.

질의 수행 계획을 실행하여 데이터를 가져오고자 할 때, 사용자는 루트 노드에 대해 질의 수행을 준비하기 위한 Open() 함수를 호출한다. Open() 함수가 정상적으로 수행되고 나면, Next() 함수를 호출하는데, 이 Next() 함수를 통해 사용자는 질의 수행 계획으로부터 해당 질의의 결과 레코드를 얻어낸다. 루트 노드의 Next() 함수가 호출되면, 차례차례 자식 노드들의 Next() 함수가 호출되고, 말단 노드에 이르면 해당 테이블로부터 실제 데이터를 읽어온다. 말단 노드는 테이블로부터 읽어온 데이터에 대해 자신의 연산을 수행하고 그 결과를 부모 노드에게 전송한 후 자신의 Next() 루틴을 종료시킨다. 물론 현재 어느 레코드까지 읽었는지에 대한 기록은 유지한다. 부모 노드는 자식 노드로부터 넘어온 레코드에 대해 자신의 연산을 수행하고, 통과할 경우 해당 레코드를 자신의 부모 노드에게 전송한다. 만약 통과하지 않으면, 다시 자식 노드의 Next() 함수를 호출하여 다음 레코드를 요청한다.

2.1.2 블록 기반 파이프라이닝

블록 기반 파이프라이닝의 경우 기본 동작은 레코드 기반 파이프라이닝과 동일하다. 다만 질의 수행 계획의 노드를 이루고 있는 각각의 연산들이 데이터를 주고받는 단위가 레코드 하나가 아니라 하나의 블록으로 묶여진 여러 개의 레코드라는 것이 다른 점이다.

질의 수행 계획의 각각의 노드들은 자신의 자식 노드로부터 데이터를 받아오기 위한 버퍼를 자식 노드의 개수만큼 가지고 있다. 이 버퍼를 자식 노드의 Next() 함수를 호출할 때, 자식노드에게 인자로 넘겨주어 자식 노드가 해당 버퍼를 채우게 하는 방식으로 레코드들을 전송받는다. 이 때, 자식 노드는 부모 노드로부터 받은 버퍼를 채우는 방식으로 데이터를 전송하므로, 버퍼 크기만큼의 레코드 블록을 부모에게 전송하는 셈이다. 이렇게 버퍼를 이용하여 블록 단위로 레코드를 전송하는 방법은 레코드를 하나씩 전송할 때보다 자식 노드의 Next() 함수를 호출하는 횟수를 훨씬 줄일 수 있다.

2.1.3 첫 레코드 응답 시간

2장을 시작하면서 첫 번째 레코드 응답 시간이 빠르다는 점이 파이프라이닝 기법의 장점이라고 밝힌 바 있다. 실제로 응답 시간이 100ms를 넘는 경우 사람이 이를 인지한다고 밝힌 연구들[15-17]도 있다. 그런데 이 응답 시간의 문제는 레코드 기반 파이프라이닝과 블록 기반 파이프라이닝 사이에도 제기될 수 있다.

첫 레코드 응답 시간은 데이터 값 분포와 필터링 조건에 따라 달라질 수 있다. 테이블 스캔의 경우 만약 필터링 조건에 의해 첫 번째 리턴 하는 레코드가 테이블의 처음에 있다면 레코드 기반 파이프라이닝이 유리하겠지만, 첫 번째 리턴 하는 레코드가 테이블의 마지막에 있다면 블록 기반 파이프라이닝이 훨씬 유리할 것이다. 조인의 경우에는 조인 조건이 첫 레코드 응답 시간에 관련하여 필터링 조건과 같은 역할을 한다.

특히 테이블 스캔의 경우 첫 레코드 리턴은 두 가지 파이프라이닝 기법 모두 매우 짧은 시간 안에 이루어지므로 그 차이가 별 의미가 없다.

3. 방향 전환 문제(Direction Switching Problem)

임베디드 데이터베이스 시스템이 블록 기반 파이프라이닝을 지원하기 위해서는 먼저 사용자가 현재 레코드의 이전 레코드를 요청할 경우에 대해 먼저 살펴보아야 한다. 즉 임베디드 환경의 특수성 때문에 Iterator 인터페이스에 새롭게 추가된 Previous() 함수의 존재와, Next()와 Previous() 함수가 필요에 따라 혼재되어 호출되는 상황을 고려해야 한다.

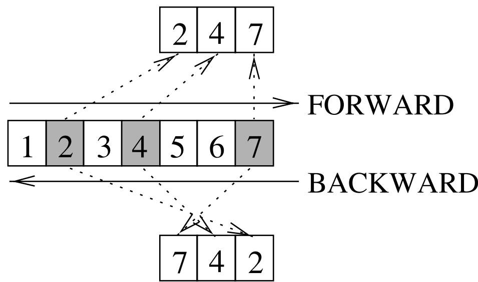
3.1 방향 전환 문제 (Direction Switching Problem)

블록 기반 파이프라이닝 질의 수행 계획에서 Next() 함수와 Previous() 함수를 필요에 따라 무작위로 호출한다고 할 때, 기존의 레코드 기반 파이프라이닝에서는 볼 수 없었던 현상을 볼 수 있다. 간단히 '방향 전환 문제 (Direction Switching Problem)'이라고 명명한 이 문제에 대해 먼저 살펴보자.

[관찰 1] 방향 전환 문제

블록 기반 파이프라이닝에서 질의 수행 계획의 부모 노드가 Next() 함수를 통해 받은 레코드 블록을 가지고 있는 상황에서 Previous()함수를 호출하면 부모 노드는 기존에 가지고 있던 레코드 블록과 동일한 레코드 블록을 다시 받게 된다. ■

(그림 1)은 1에서 7까지의 데이터를 읽고 푸르게 표시된 데이터(2, 4, 7)를 부모 노드의 버퍼에 채우는 그림이다. 먼저 Next() 함수가 호출되어 순방향으로 (FORWARD) 데이터를 읽으면서 부모 노드의 버퍼에 (2, 4, 7)의 데이터를 채웠다고 가정하자. 그러면 현재 노드는 부모 노드의 버퍼를 다 채우고 Next() 함수를 종료하므로 현재 노드의 버퍼 인덱스는 7을 가리키고 있게 된다. 이 상황에서 Previous() 함수가 호출되어 역방향으로 (BACKWARD) 다시 데이터를 읽으면서 부모 노드의 버퍼를 채우면, 현재 인덱스 위치인 7의 데이터부터 읽기 시작해서 부모 노드의 버퍼를 채우게



(그림 1) 방향 전환 문제 - 중복된 데이터 전송 (buffer size=3)

되는데 이 경우 부모 노드의 버퍼에는 (7, 4, 2)의 데이터가 들어가게 된다. 그런데, 이 데이터는 순서는 틀리지만, 이전에 부모 노드가 받았던 레코드 블록과 같은 데이터다. 즉, 중복된 데이터를 부모 노드가 받게 된다.

[관찰 2] 방향 전환 문제의 발생 조건

현재 부모 노드의 버퍼에 저장된 레코드 블록의 방향을 dir_{buff}라고 하고 해당 레코드 블록에 새롭게 데이터를 채울 함수(Next() 혹은 Previous() 함수)의 방향을 dir_{func}이라고 할 때, 관찰 1은 dir_{buff}와 dir_{func}의 값이 다를 경우에만 발생한다. ■

위의 관찰 1과 2를 바탕으로, 현재 부모 노드의 버퍼에 저장된 레코드 블록의 방향과 새롭게 요청하는 레코드 블록의 방향이 다를 경우 중복된 데이터가 전송되는 문제를 전체 질의 수행 계획에서 효율적으로 해결되어야 블록 기반의 파이프라이닝을 임베디드 데이터베이스 시스템에 적용할 수 있다.

3.2 단순 접근 방법 (Naive Method)

방향 전환 문제를 해결하기 위해 가장 먼저 생각해 볼 수 있는 방법은 다음과 같다.

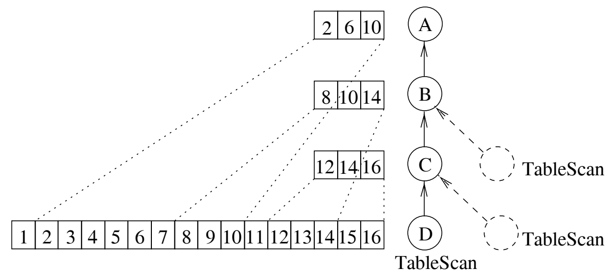
부모 노드가 자식 노드의 Next() 함수를 호출하다가 방향을 바꿔 Previous() 함수를 호출하는 경우 자식 노드는 자신의 레코드 블록을 비우고 새로 자신의 자식 노드의 Previous() 함수를 호출하여 데이터를 받아온 후 부모 노드의 버퍼를 채우면 된다. 이 때 자식 노드는 자신의 버퍼를 지우기 전 마지막으로 접근했던 레코드의 버퍼 인덱스를 유지하고 있어야 한다.

4 블록 단위 스키핑 기법 (Block-wise Skipping)

4.1 블록 단위 스키핑

3.2절의 단순 접근 방법은 방향 전환이 일어나는 경우 이미 버퍼에 저장해 놓은 데이터를 활용할 수 없다는 단점이 있다. 블록 단위 스키핑은 이러한 단순 접근 방법의 비효율적인 부분을 제거하기 위해 고안하였다.

[관찰 3] 각 노드의 레코드 블록의 범위



(그림 2) 각 노드 레코드 블록의 범위 (buffer size = 3)

(그림 2)를 살펴보면 질의 수행 계획의 각각의 노드들이 가지고 있는 버퍼에 저장된 레코드 블록이 실제 테이블 상에서 가지는 범위는 매번 동적으로 바뀌고 규칙성이 없다. 그러나 부모 노드와 자식 노드의 레코드 블록의 범위를 살펴보면, 항상 일정 부분 겹쳐지는 부분이 있음을 알 수 있다. 이 겹쳐지는 부분만큼은 실제 테이블에서 레코드를 가져오지 않아도 자식 노드에서 바로 처리가 가능하다. ■

이러한 관찰 3을 바탕으로 블록 단위 스키핑은 자식 노드에서 부모 노드로 레코드 블록을 넘겨줄 때 미리 중복되는 부분을 자식 노드에서 처리하고 필요한 부분부터 넘겨준다. 개괄적인 알고리즘을 살펴보면 (그림 3)과 같다.

(그림 3)에서 먼저 1번째 라인을 보면, 부모 노드의 레코드 블록의 방향과 현재 수행중인 함수의 방향이 다른 경우, 즉 방향 전환 문제 발생 조건 시에만 수행됨을 알 수 있다. 2번째 라인의 while문에 의해 부모 노드의 버퍼에 들어 있는 레코드 수만큼 부모 노드의 버퍼에 채울 레코드를 스킵한다. 여기서 3번째 라인의 Check() 함수는 해당 레코드에 대해 현재 노드의 조건을 검사하는 부분이다. 현재 레코드를 살펴본 후에는 다시 5번째 라인에서 다음 레코드를 rec_{curr}에 할당한다. 6번째 라인에서는 현재 노드의 버퍼에

```
Procedure BlockWiseSkipping()
// dir_buff: 부모 노드의 버퍼 레코드 방향
// dir_func: 현재 수행중인 함수의 방향
// n_skip: 스키핑 한 레코드 수
// n_buff: 부모 노드의 버퍼에 들어 있는 레코드 수
// rec_curr: 현재 노드의 버퍼 인덱스가 가리키는 레코드
begin
 1. if ( dir_buff ≠ dir_func ) {
 2.     while ( n_skip < n_buff ) {
 3.         if (Check( rec_curr ))
 4.             ++ n_skip;
 5.         Move( rec_curr );
 6.         if ( rec_curr = NULL ) {
 7.             Get data from child;
 8.             First( rec_curr )
 9.         }
10.     }
11. }
12. dir_buff := dir_func;
end
```

(그림 3) 블록 기반 스키핑 알고리즘

있는 레코드들을 모두 읽었는지 검사하고 모두 읽었으면, 7번째 라인에서 자식 노드로부터 데이터를 새로 가져온다. 이 때 자식 노드도 블록 기반 스키핑 기법을 수행한 후 레코드 블록을 돌려주므로 중복된 데이터를 받아오는 경우는 생각해 주지 않아도 된다. 8번째 라인에서는 새로 받아온 데이터의 첫 번째 레코드를 rec_{curr}에 할당한다.

(그림 3)을 보면 알 수 있지만, 블록 기반 스키핑 기법은 먼저 현재 노드의 레코드 블록에 있는 레코드를 재활용함으로써 해당 레코드들을 자식 노드로부터 새로 받아오는 비용을 줄일 수 있다.

이 블록 기반 스키핑 알고리즘은 질의 수행 계획에서 각 노드의 Next() 함수 및 Previous() 함수의 첫 부분에서 수행해주면 된다. 나머지 부분은 일반적인 파이프라이닝 기반의 Next()와 Previous() 함수의 구현과 동일하다.

이제 4.2절과 4.3절의 예를 통해 Next() 함수 수행 중 방향이 바뀌어 Previous() 함수가 호출될 때 블록 기반 스키핑 기법이 어떻게 적용되는지 살펴보자.

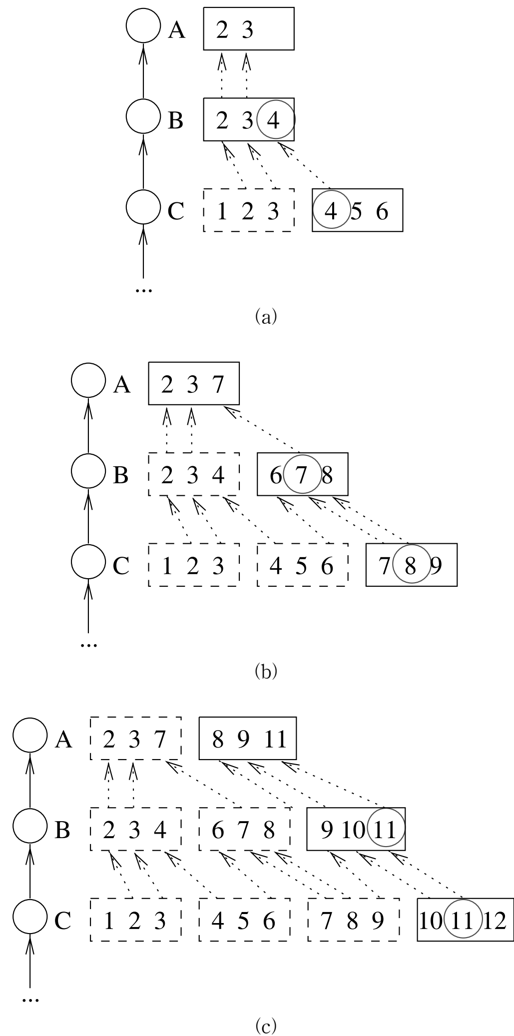
참고로, (그림 4)와 (그림 5)에서 각 노드는 부모 노드로부터 데이터 요청이 들어오면 자신의 버퍼에 있는 레코드들을 검증하여 부모 노드의 버퍼를 채운다. 이 때, 각 노드의 버퍼의 크기는 3이라 가정한다. 즉 3개의 레코드를 저장할 수 있다고 가정한다. 각 노드의 우측에 버퍼의 상태를 나타내는 사각형이 있다. 이 중 점선으로 된 부분은 이전의 상태를 나타내고 우측으로 갈수록 더 최근의 상태를 나타낸다.

4.2 블록 기반 파이프라이닝의 Next() 함수

먼저 (그림 4)의 (a)와 (b)는 A노드가 B노드에게 Next() 함수를 통해 데이터를 요청하는 경우를 보여준다. (그림 4)의 (a)에서 A노드가 B노드에게 데이터를 요청하면 B노드는 C노드에게 다시 필요한 데이터를 요청한다. C노드는 자신의 버퍼에 저장된 데이터 중 자신이 가지고 있는 조건 연산을 통과하는 것들(2, 3)을 B의 버퍼에 채운다. 아직 B노드의 버퍼를 다 채우지 못했으므로 C노드는 자신의 자식 노드에게 다음 데이터를 요청하여 (4, 5, 6)을 받아온다. 이 중 4를 B노드의 버퍼에 넣어 채운 후 자신의 버퍼의 현재 위치(그림에 붉은색 원으로 표시: C노드 버퍼의 첫 번째 위치)를 저장하고 리턴 한다. B노드는 C노드로부터 받은 (2, 3, 4)중 다시 자신의 조건 연산을 통과하는 것들(2, 3)을 A노드의 버퍼에 채운다. B노드는 A노드의 버퍼를 다 채우지 못하였으므로 다시 C노드에게 데이터를 요청하고, C노드는 자신의 버퍼에 남아있는 데이터 (5, 6)을 처리하여 (그림 4)의 (b)로 넘어간다.

(그림 4)의 (b)에서 C노드는 버퍼에 남아있던 데이터로 B노드의 버퍼를 다 채우지 못해 새로 (7, 8, 9)를 자식 노드로부터 받아와 7과 8을 B노드의 버퍼에 채운 후 리턴 한다. B노드는 새로 받은 (6, 7, 8) 중 7로 A노드의 버퍼를 채운 후 현재 위치(B노드 버퍼의 두 번째 위치)를 저장하고 리턴 한다.

(그림 4)의 (c)는 A노드가 다시 B노드의 Next()를 호출한 경우를 나타낸다. B노드와 C노드는 다시 (그림 4)의 (a)와 (b)의 과정을 거친 후, (8, 9, 11)로 A노드의 버퍼를 채운다.



(그림 4) 블록 기반 파이프라이닝의 Next( ) 함수 (buffer size=3)

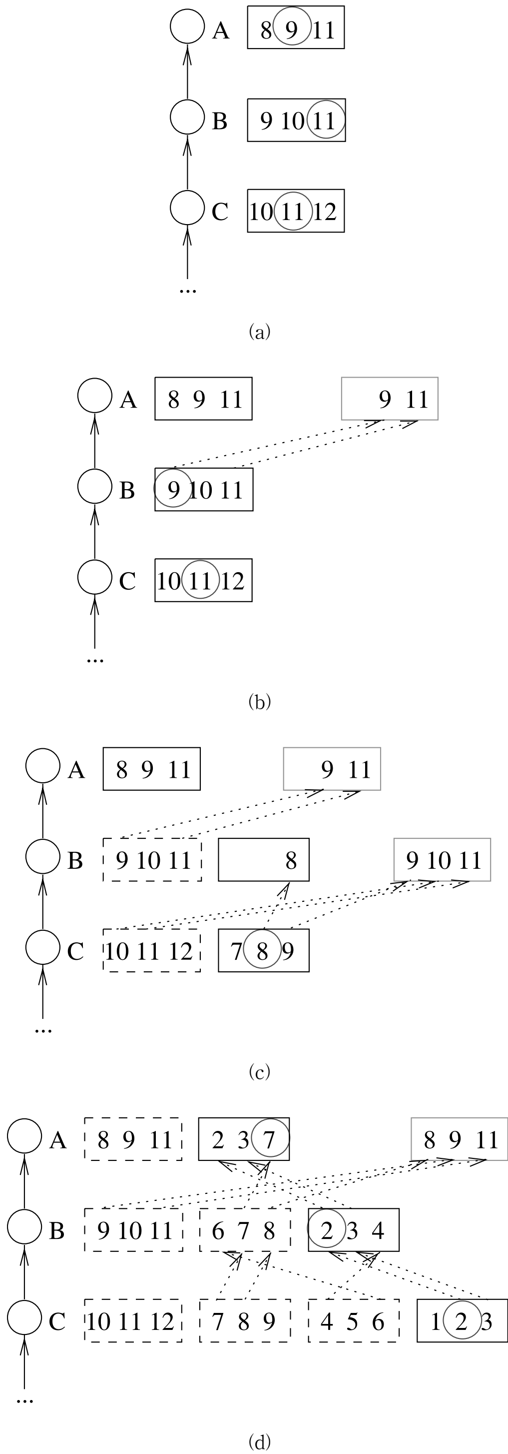
이제 A노드가 B노드의 Previous() 함수를 호출하기 전에, 8과 9의 데이터를 이미 사용자에게 리턴하였다고 가정해 보자. 이 경우 A노드 버퍼의 현재 위치는 두 번째 위치이고 이는 (그림 5)의 (a)와 같다.

4.3 블록 기반 파이프라이닝의 Previous() 함수

이제 사용자가 A노드에게 이전 데이터를 다시 요청하면, A노드는 현재 자신의 버퍼에서 바로 8을 리턴해 주면 된다. 여기서 사용자가 다시 더 이전의 데이터를 요청하면, A노드는 B노드의 Previous() 함수를 호출하여 이전 데이터를 요청한다. B노드도 현재 자신의 버퍼에 있는 데이터를 이용하여 A노드의 Previous() 함수 호출을 처리하게 되는데, B노드 버퍼의 현재 위치는 11이었으므로 그 위치에서부터 역방향으로 데이터를 하나씩 처리한다.

(그림 5)의 (b)에서 A노드가 B노드의 Previous() 함수를 호출할 때, A노드의 버퍼에 있던 데이터는 A노드가 B노드의 Next()를 호출하여 채운 것이다. 그러므로 B노드는 A노

드의 버퍼에 바로 데이터를 채우지 않고 블록 단위 스키핑 기법을 이용하여 중복된 데이터를 처리한 후 A노드의 버퍼에 데이터를 채워준다. (그림 5)의 (b)에서는 중복된 데이터를 처리하기 위해 B노드가 연두색으로 표시된 버퍼에 데이터를 넣고 있다. 이 버퍼는 실제 버퍼가 아니라 블록 단위 스키핑 기법을 그림으로 표현하기 위한 가상의 버퍼로 '스키핑 리스트 (Skipping List)'라고 부른다. B노드가 자신의 버퍼에 있는 데이터를 모두 처리하고 나면, C노드의 Previous() 함수를 호출하여 데이터를 요청한다.



(그림 5) 블록 기반 파이프라이닝의 Previous() 함수 (buffer size=3)

(그림 5)의 (c)에서 C노드도 B노드와 마찬가지로 자신의 버퍼의 현재 위치에서부터 역방향으로 데이터를 읽으면서 처리한다. 이 때 B노드의 버퍼에 있는 데이터도 C노드의 Next() 함수로 채운 것이므로 C노드도 블록 단위 스키핑 알고리즘을 수행한다. C노드는 (9, 10, 11)의 데이터로 스키핑 리스트를 다 채운 후 남은 데이터로 B노드의 버퍼를 채운다. 즉, 8부터는 B노드의 버퍼를 채운다.

(그림 5)의 (d)는 A노드의 B노드에 대한 Previous() 함수 호출이 모두 처리된 상태의 그림이다. C노드부터 살펴보면, C노드가 B노드에게 (6, 7, 8)을 넘겨주면, B노드는 8을 스키핑 리스트에 기록하여 블록 단위 스키핑 알고리즘 수행을 완료하고 A노드의 버퍼를 채우기 시작한다. B노드는 남은 데이터 중 7의 데이터를 A노드의 버퍼에 넣고 다시 C노드의 Previous() 함수를 호출한다. 이번에는 B노드의 버퍼에 있던 (6, 7, 8)의 데이터가 C노드의 Previous() 함수에 의해 채워졌던 데이터이므로, C노드는 블록 단위 스키핑 알고리즘의 수행 없이 5의 데이터부터 역방향로 데이터를 읽으면서 B노드의 버퍼를 채운다. B노드는 (2, 3, 4)를 C노드로부터 받은 후 4의 데이터부터 역방향으로 읽으면서 A노드의 버퍼를 채운다. 즉 3과 2를 A의 버퍼에 넣은 후 리턴하게 되고, 결국 A노드는 (8, 9, 11)의 이전 데이터인 (2, 3, 7)을 받아 7을 사용자에게 리턴 할 수 있게 된다.

5. 실험 결과

본 장에서는 블록 기반 스키핑 알고리즘을 이용하는 임베디드 데이터베이스 시스템을 위한 질의 처리기의 성능을 살펴보겠다. 삼성전자의 임베디드 데이터베이스 시스템인 AceDB [12-14]를 기반으로 구현하였다. 테이블 스캔과 질의 수행 계획의 루트 역할을 하는 커서 클래스를 수정하여 블록 단위 데이터 전송을 지원하게 하였고, 블록 Nested-Loop 조인을 새롭게 추가하였다. 임베디드 시스템 개발 환경인 EDB9315A Development System 에서 실험을 수행하였고 구체적인 사양은 다음과 같다.

- 200 MHz ARM920T Processor
- 16 KB data cache \& 16 KB instruction cache
- 100 MHz system bus
- 64 MB SDRAM
- 2 GB NAND flash memory

File I/O는 플래시 메모리의 페이지 크기인 512 바이트를 단위로 수행하도록 하였다.

블록 Nested-Loop 조인의 경우 조인 조건이 없는 Cartesian Product를 이용하였고, Outer 테이블의 버퍼 크기는 파라미터로 주어지는 최대 메모리 크기에서 질의 수행

계획상의 노드 간 데이터 전송을 위한 버퍼의 크기를 뺀 크기를 할당하였다. 예를 들어, 조인이 쓸 수 있는 최대 메모리가 1024 바이트인 경우 노드 간 데이터 전송을 위해 32 바이트를 버퍼 크기로 이용한다면, Outer 테이블의 버퍼 크기는 1024 - 32 = 992 바이트가 되고, Inner 테이블의 버퍼의 크기는 32 바이트가 된다.

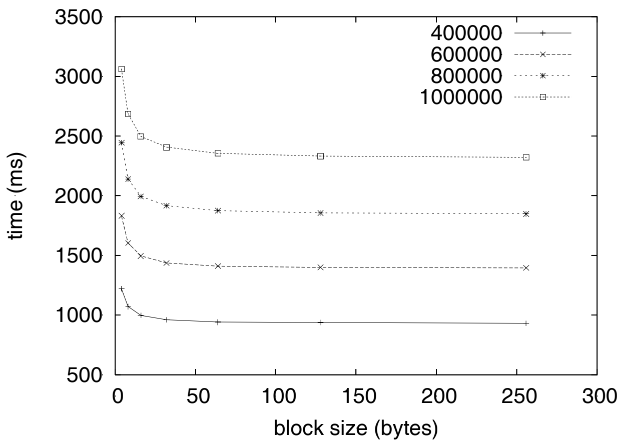
실험에 사용한 데이터는 4 바이트 정수형 데이터 타입의 칼럼을 하나만 가지는 테이블을 가정하였다.

각 실험 그래프에서 가장 왼쪽의 값은 레코드 기반일 때의 수행 속도를 가리킨다.

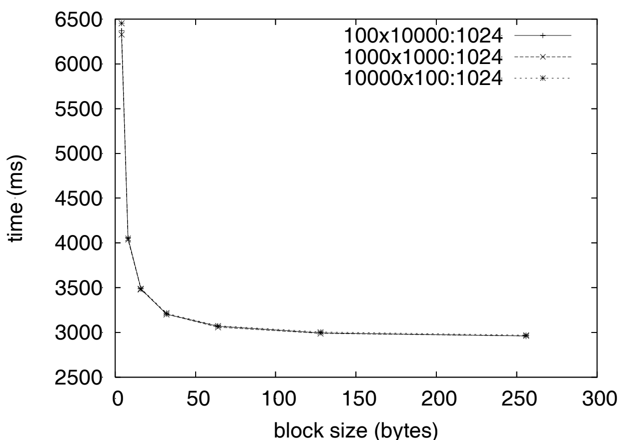
5.1 테이블 스캔

테이블 스캔 실험은 블록 기반 파이프라이닝의 함수 호출 비용의 절감 효과를 보기 위한 실험이다. 실험은 각각 400,000, 600,000, 800,000, 1,000,000건의 레코드를 가지고 있는 테이블을 대상으로 수행하였다.

(그림 6)을 살펴보면, 버퍼의 크기가 증가함에 따라 수행시간이 급속히 수렴하는 것을 알 수 있다. 실제 계산을 해보면, 버퍼의 크기가 두 배로 늘어날 때마다 함수 호출은 1/2씩 줄어든다. 즉, 처음 2개의 레코드를 블록으로 전송할 때 가장 효과가 크고, 그 이후부터는 기하급수적으로 효과가 줄어드는 것이다. 버퍼의 크기가 16에서 32 바이트일 때 메모리 사용량 대비 수행시간의 절감 효과가 가장 좋았다.



(그림 6) 버퍼(블록) 크기에 따른 테이블 스캔 시간 그래프



(그림 7) 버퍼(블록) 크기에 따른 조인 시간 그래프

5.2 블록 NESTED-LOOP 조인

이 실험은 블록 Nested-Loop 조인을 이용할 수 있는 블록 기반 파이프라이닝의 효과를 알아보기 위한 실험이다. 조인이 최대로 사용할 수 있는 메모리의 크기는 1024 바이트로 하였고, 100건, 1,000건, 10,000건의 레코드를 가진 테이블을 대상으로 100×10000, 1000×1000, 10000×100의 세 가지 조인의 경우에 대해 실험했다.

(그림 7)에서 그래프를 살펴보면, 세 가지 조인이 모두 결과 레코드의 수가 같으므로 거의 동일한 수행 속도를 보여준다. 세 가지 조인 모두 버퍼 크기가 32 바이트일 때 레코드 기반 Nested-Loop 조인보다 수행시간이 약 50% 정도로 향상되었다.

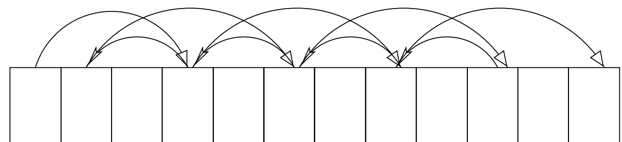
5.3 지그재그 스캔

지그재그 스캔이란 파이프라인 커서에 대해 (그림 8)과 같이 일정 비율로 Next() 함수 호출과 Previous() 함수 호출을 반복하는 것이다.

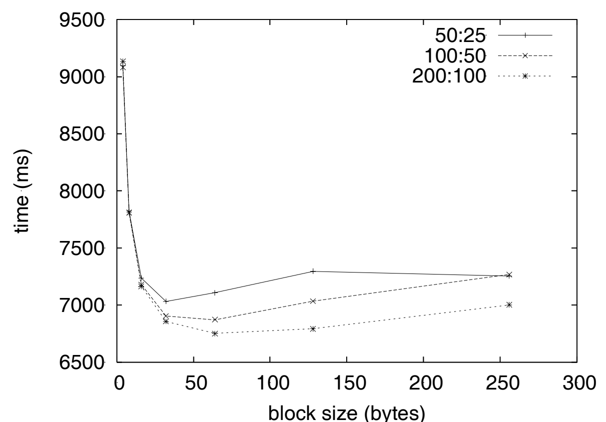
(그림 8)은 4:2의 비율로 Next() 함수와 Previous() 함수를 호출하는 경우를 보여준다.

지그재그 스캔은 실제 Next() 함수 호출과 Previous() 함수 호출이 혼재되어 있는 상황을 시뮬레이션하기 위한 실험으로 블록 기반 파이프라이닝의 약점을 알아보기 위한 실험이다.

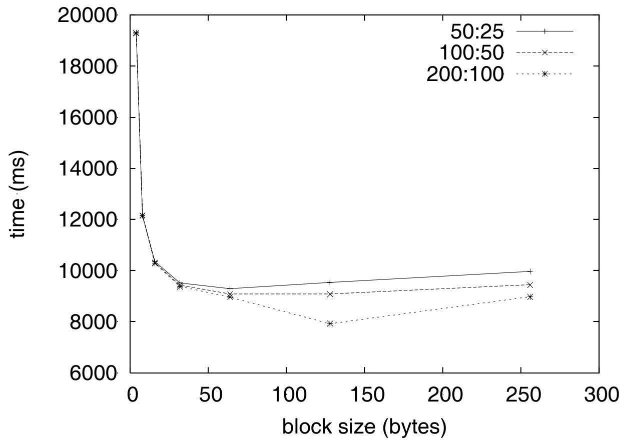
(그림 9)와 (그림 10)은 각각 Next() 함수 호출과 Previous() 함수 호출의 비가 50:25, 100:50, 200:100인 경우를 보여준다.



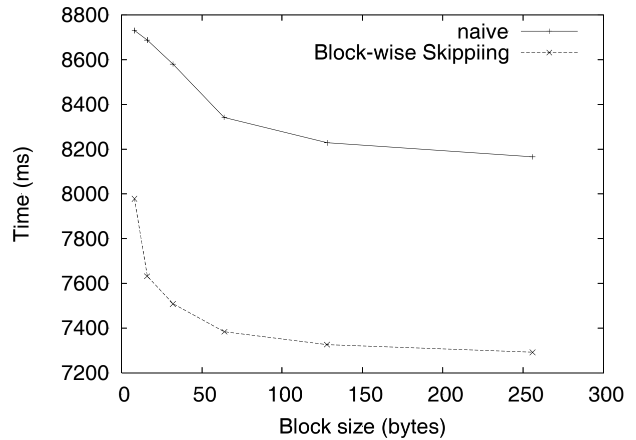
(그림 8) 지그재그 스캔



(그림 9) 지그재그 테이블 스캔 시간 그래프



(그림 10) 조인 결과에 대한 지그재그 스캔 그래프



(그림 11) 블록 단위 스키핑 효과 그래프(10:5 지그재그 스캔)

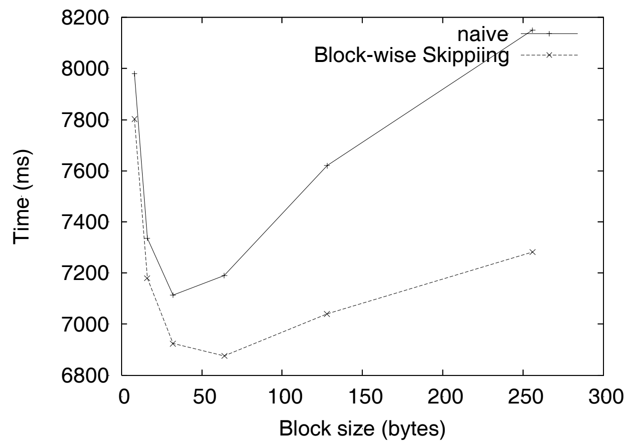
그래프를 살펴보면 공통적으로 버퍼의 크기가 증가함에 따라 수행 시간이 감소하다가 증가하는 모습을 보여준다. 이는 지그재그의 범위가 큰 경우에 더 두드러지는데, 지그재그의 범위가 작은 50:25의 경우 오히려 다시 좋아지는 모습도 보여주고 있다. 이는 지그재그의 특성상 다음과 같은 상황이 발생하기 때문이다.

- 버퍼의 크기가 증가함에 따라 수행시간이 늘어나는 이유는 블록 기반 파이프라이닝에서 각각의 노드가 부모 노드의 버퍼를 모두 채운 후에야 자신의 함수를 끝내기 때문이다. 예를 들면, 100번째 Next() 함수를 호출하였을 때 새로운 레코드 블록을 자식 노드로부터 받아온 경우 새로운 레코드 블록에서 첫 번째 레코드를 리턴한 후 다음 함수 호출은 Previous() 이기 때문에, 나머지 레코드를 사용해보지도 못하고 다시 이전 데이터를 새롭게 받아와야 하는 문제가 발생하는 것이다. 이런 현상은 버퍼의 크기가 클수록 더 심해질 수 있다.
- 지그재그의 범위가 작을 경우에는 버퍼의 크기가 증가하더라도 수행시간이 많이 늘어나지 않는다. 이는 지그재그의 범위가 버퍼의 크기 안에 있을 경우 자식 노드로부터 새로운 레코드 블록을 받아오지 않고 현재 레코드 블록 내에서 지그재그가 이루어져 현재 버퍼의 레코드들을 최대 3번씩 재사용하기 때문이다.

5.4 블록 단위 스키핑 효과

이 실험은 3.2절의 단순 접근 방법에 비해 블록 단위 스키핑 기법이 버퍼에 저장된 레코드 블록을 재사용하는 효과가 얼마나 좋은지 알아보기 위한 실험이다. 실험은 지그재그 범위가 10:5인 경우와 100:50인 경우 두 가지에 대해 수행하였다.

(그림 11)과 (그림 12)를 살펴보면, 이전의 지그재그 실험과 유사한 양상을 보여주는데, 지그재그 범위가 10:5와 같이 극단적으로 작은 경우에는 버퍼의 크기가 늘어나도 전혀 수행시간이 나빠지지 않음을 알 수 있다. 단순 접근 방법의 경우 스키핑 효과가 전혀 없으므로 지그재그 범위가 큰 경



(그림 12) 블록 단위 스키핑 효과 그래프(100:50 지그재그 스캔)

우에 버퍼의 크기가 증가함에 따라 수행시간이 늘어나는 정도가 훨씬 심하다. 단순 접근 방법의 경우 자식 노드로부터 버퍼를 채울 만큼의 레코드를 가져오는 비용이 블록 기반 스키핑 기법보다 훨씬 비싸기 때문이다.

6. 결론 및 향후 연구

본 논문에서는 플래시 메모리상에서의 임베디드 데이터베이스 시스템을 위한 블록 기반 파이프라이닝 기법을 검토해 보았다. 그 결과 방향 전환 문제가 존재한다는 것을 알게 되었고, 그에 대한 해법으로 블록 기반 스키핑 알고리즘을 제시하였다.

본 논문에서 제안한 블록 기반 스키핑 알고리즘이 임베디드 데이터베이스 시스템에 기여한 점을 다시 한 번 살펴보면, 먼저 역방향 데이터 전송과 블록 기반 파이프라이닝 기법을 함께 사용할 수 있게 하였다는 점을 들 수 있다. 그로 인해 블록 기반 파이프라이닝 기법의 장점인 질의 수행 시의 함수의 호출 횟수를 감소를 통한 함수 호출 비용의 절감을 가져올 수 있었으며, 블록 Nested-Loop 조인과 같은 성능이 우수한 블록 기반 알고리즘을 이용할 수 있었다.

또 한 가지는 블록 기반 스키핑 기법을 통해 각 노드의 버퍼에 저장된 데이터를 재활용할 수 있게 하였다는 점이다. 이를 통해 데이터 전송의 방향 전환 시에 단순 접근 방법에 비해 성능 향상을 보여줄 수 있었다.

처음으로 시도된 임베디드 데이터베이스와 블록 기반 파이프라이닝 기법의 조합은 점점 모바일 기기에서 데이터베이스 시스템의 중요성이 증가하고 있는 오늘날 실용적인 적용이 충분히 가능한 의미 있는 연구가 될 것으로 기대된다.

참 고 문 헌

[1] 변시우, 플래시메모리 데이터베이스를 위한 플래시 인지 트랜잭션 관리 기법, 인터넷정보학회논문지 제6권 제1호, 2005-02.

[2] 변시우, 휴대용 데이터베이스를 위한 지연된 소거 리스트를 이용하는 플래시 메모리 쉐도우 페이징 기법, Journal of Information Technology Applications & Management Vol.13, No.2, 2006-06.

[3] Sang-won Lee and Won Kim. On Flash-Based DBMSs: Issues for Architectural Re-Examination, In the Journal of Object Technology, Vol.6, No.8, September-October, 2007.

[4] Sang-Won Lee, Bongki Moon Design of Flash-Based DBMS: An In-Page Logging Approach, SIGMOD. Beijing, China, June, 11-14, 2007.

[5] 이수관, 민상렬, 조유근. 플래시 메모리 관련 최근 기술 동향. 정보과학회지 제24권 제12호, 2006. 12.

[6] Sang-Won Lee, Gap-Joo Na, Jae-Myung Kim, Research Issues in Next Generation DBMS for Mobile Platforms, mobileHCI. Singapore, September, 9-12, 2007.

[7] 노홍찬, 김승우, 김우철, 박상현, 플래시 메모리 상에서의 효율적인 동작을 위한 수정 B-트리 인덱스, 한국정보과학회 가을 학술발표논문집, Vol.33, No.2(C), 2006.

[8] 나갑주, 이상원, 플래시메모리기반의 B+트리 알고리즘, 한국인터넷정보학회 2006 임시총회 및 춘계학술발표대회 제7권 제1호, 2006. 04.

[9] 박상원, 플래시메모리와 데이터베이스, 정보과학회지, 제25권 제6호, 2007. 06.

[10] Abraham Silberschatz, Henry F. Korth, S. Sudarshan. Database System Concepts, Fourth Edition, McGraw-Hill, 2002.

[11] 윤승희, 송하주, 플래시 메모리 기반 임베디드 데이터베이스 시스템을 위한 지연쓰기 기법. 한국정보과학회 가을 학술발표논문집, Vol.33, No.2(C), 2006.

[12] 진희규, 이기용, 우경구, 박희선, "멀티미디어 CE기기를 위한 빠른 질의 복구 기법", 정보과학회, 2008년 6월.

[13] Ki Yong Lee, Hyojun Kim, Kyoung-Gu Woo, Hee-Seon Park, "AceDB Flashlight: Design and Implementation of a Flash-aware DBMS", SAMSUNG Journal of Innovative Technology, Vol.4, No.1, February, 2008.

[14] Hyojun Kim, Ki Yong Lee, Jung Jae Gyu, Kyoungil Bahng, "A New Transactional Flash Translation Layer For Embedded Database Systems Based on MLC NAND Flash Memory," International Conference on Consumer Electronics, 2008.

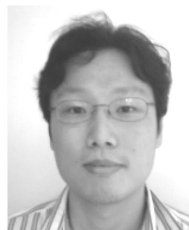
[15] S. K. Card, G. G. Robertson, and J. D. Mackinlay, "The information visualizer: An information workspace," In Proceedings of ACM CHI '91 Conf., 1991, pp.181-188.

[16] R. B. Miller, "Response time in man-computer conversational transactions," Proc. AFIPS Fall Joint Computer Conference Vol.33, 1968, pp.267-277.

[17] B. A. Myers, "The importance of percent-done progress indicators for computer-human interfaces," Proc. ACM CHI '85 Conf., April, 1985, pp.11-17.

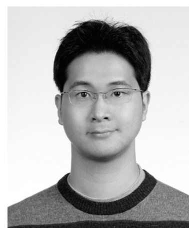
정 재 혁

e-mail : jhchong@kdd.snu.ac.kr
1998년 서울대학교 토목공학과(학사)
2008년 서울대학교 전기공학부(석사)
2008년~현 재 SAP R&D Center Korea Senior Developer
관심분야 : 데이터마이닝, 데이터베이스



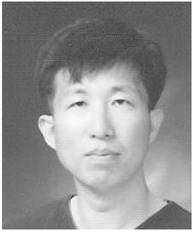
박 형 민

e-mail : hmpark@kdd.snu.ac.kr
2003년 서울대학교 전기컴퓨터공학부(학사)
2004년~현 재 서울대학교 전기컴퓨터공학부 박사과정
관심분야 : 데이터마이닝, 데이터베이스



홍 석 진

e-mail : s.jin.hong@samsung.com
1998년 서울대학교 컴퓨터공학과(학사)
2000년 서울대학교 전기컴퓨터공학부(석사)
2006년 서울대학교 전기컴퓨터공학부(박사)
2006년~현 재 삼성전자 종합기술원 전문연구원
관심분야 : Database, Stream Processing, Data Mining



심 규 석

e-mail : shim@ee.snu.ac.kr

1986년 서울대학교 전기공학과(학사)

1988년 University of Maryland, College Park(석사)

1993년 University of Maryland, College Park(박사)

1994년~1994년 Federal Reserve Board, Research Staff

1994년~1996년 IBM Almaden Research Center, Research Staff

1996년~2000년 Bell Laboratories, Member of Technical Staff

1999년~2002년 KAIST 전산학과 조교수

2002년~현 재 서울대학교 전기컴퓨터공학부 교수

2001년~2007년 VLDB 저널 Editorial Board

2004년~2008년 IEEE TKDE 저널 Editorial Board

2010년 ACM SIGMOD Program Committee Member

2009년 IEEE ICDE Vice-Chair

2008년 IEEE ICDM Program Committee Vice-Chair

관심분야 : 데이터마이닝, 데이터베이스, XML, Stream Data