주기억장치 DBMS를 위한 고성능 인덱스 관리자의 설계 및 구현

김 상 욱*, 이 경 태**, 최 완**

Design and Implementation of a High-Performance Index Manager in a Main Memory DBMS

Sang-Wook Kim*, Kyung-Tae Lee**, Wan Choi**

요 약

주기억장치 DBMS(MMDBMS)는 디스크가 아닌 주기억장치를 주요 저장 매체로서 사용하므로 고속의 처리를 요구하는 다양한 데이터베이스 응용을 효과적으로 지원한다. 본 논문에서는 차세대 MMDBMS Tachyon의 인덱스 관리자 개발에 관하여 논의한다. 최근 하드웨어의 급격한 발전으로 인하여 주기억장치 액세스 속도와 CPU의 처리 속도의 차는 점점 커지고 있다. 따라서 CPU 내에 있는 캐쉬(cache)의 존재를 충분히 활용하는 자료 구조 및 알고리즘을 고안함으로써 MMDBMS의 성능을 크게 개선시킬 수 있다. 본 논문에서는 Tachyon를 위한 캐쉬-인지 인덱스 관리자의 개발 중에 경험한 실질적인 구현 이슈들을 언급하고, 이들에 대한 해결 방안을 제시한다. 본 논문에서 다루는 주요 이슈들은 (1) 캐쉬(cache)의 효과적인 사용, (2) 인덱스 엔트리 및 인덱스 노드의 집약적 표현(compact representation), (3) 가변 길이 키(variable-length key)의 지원, (4) 다중 애트리뷰트 키(multiple-attribute key)의 지원, (5) 중복 키(duplicated key)의 지원, (6) 인덱스를 위한 시스템 카탈로그의 정의, (7) 외부 API(application programming interface)의 정의, (8) 효과적인 동시성 제어 방안, (9) 효율적인 백업 및 회복 방안 등이다. 또한, 다양한 실험을 통한 성능 분석을 통하여 제안된 인덱스 관리자의 우수성을 규명한다.

ABSTRACT

The main memory DBMS(MMDBMS) efficiently supports various database applications that require high performance since it employs main memory rather than disk as a primary storage. In this paper, we discuss the index manager of the Tachyon, a next-generation MMDBMS. Recently, the gap between the CPU processing and main memory access times is becoming much wider due to rapid advance of CPU technology. By devising data structures and algorithms that utilize the behavior of the cache in CPU, we are able to enhance the overall performance of MMDBMSs considerably. In this paper, we address the practical implementation issues and our solutions for them obtained in developing the cache-conscious index manager of the Tachyon. The main issues touched are (1) consideration of the cache behavior, (2) compact representation of the index entry and the index node, (3) support of variable-length keys, (4) support of multiple-attribute keys, (5) support of duplicated keys, (6) definition of the system catalog for indexes, (7) definition of external APIs, (8) concurrency control, and (9) backup and recovery. We also show the effectiveness of our approach through extensive experiments.

* 한양대학교 정보통신대학(wook@ihanyang.ac.kr), ** 강원대학교 컴퓨터정보통신공학과(mlogue@nate.com)
*** 한국전자통신연구원 컴퓨터시스템 연구부(wchoi@etri.re.kr)
논문번호: 020453-1017, 접수일자:2002년 10월 7일
※ 본 연구는 한국과학재단 목적기초 연구사업(과제번호: R05-2002-000-01085-0)과 2003년도 한양대학교 교내 연구비의 지원으로 수행되었습니다.

I. 서 론

최근, 컴퓨터의 계산 능력이 크게 향상됨에 따라 실시간 응용 분야가 급격히 확대되고 있다. 이러한 실시간 응용 분야의 데이터를 효과적으로 관리하기 위한 대표적인 방법은 DBMS의 저장 매체인 디스크를 액세스 속도가 빠른 주기억장치로 대체하는 것이다[22]. 주기억장치 DBMS(main memory DBMS: MMDBMS)는 저장 매체로서 주기억장치를 사용하며, 이 결과 데이터를 검색 및 갱신할 때 디스크 액세스로 인하여 응답 시간이 지연되는 디스크 기반 DBMS의 문제를 근본적으로 해결한다[2][5][8].

ETRI 실시간 DBMS 팀에서는 강원대학교 데이터 및 지식공학 연구실과 공동으로 차세대 주기억장치 DBMS(main memory DBMS: MMDBMS)[2][8] Tachyon을 개발하였다. Tachyon은 실시간 응용을 주요 지원 대상으로 하며, 따라서 마감 시간(deadline) 개념을 지원한다. 이러한 효과적인 실시간 응용의 지원을 위하여 주요 저장 매체로서 디스크가 아닌 주기억장치를 사용한다. 현재, Tachyon은 (주) 코스모에 의해 상용화에 성공하여, 통신 응용을 중심으로 한 다양한 고성능 응용 분야에 적용 중에 있다.

DBMS의 인덱스 관리자(index manager)는 사용자가 원하는 객체(object)를 신속하게 검색하도록 하는 기능을 지원한다. 즉, 객체의 특정 애트리뷰트(attribute)를 키(key)로 선정하여 인덱스를 구성함으로써 특정 키 값을 갖는 객체를 데이터베이스로부터 직접 액세스하도록 한다. 본 논문에서는 MMDBMS Tachyon을 위한 고성능 인덱스 관리자 개발에 관하여 논의한다.

지금까지 인덱스에 대한 다양한 연구가 수행되어 왔으며, 이러한 연구 결과로 이진 탐색 트리(binary search tree)[14], AVL-트리[14], T-트리[15], B-트리[4], CSS-트리[19], CSB^+-트리[20] 등 다양한 인덱스 구조들이 제안된 바 있다.

그러나 이러한 연구들은 주로 인덱스의 구조 및 특징에 연구의 초점을 두었을 뿐, 구현에서 발생하는 실질적인 이슈들에 대해서는 언급하고 있지 않다. 본 논문에서는 Tachyon의 인덱스 관리자의 개발 및 시스템 통합을 통하여 경험한 여러 실질적인 구현 이슈들에 관하여 논의한다. 즉, 본 연구의 주요 공헌은 새로운 인덱스 구조를 제시하는 것이 아니라, 기존에 제안된 인덱스 구조들 중 MMDBMS Tachyon에 적합한 것은 선정하고, 이를 MMDBMS 상에서 개발하는 과정에서 발생하는 문제점과 이에 대한 효과적인 해결 방안을 제시하는 것이다.

본 연구에서 다루는 주요 이슈들은 (1) 캐쉬(cache)의 효과적인 사용, (2) 인덱스 엔트리 및 인덱스 노드의 집약적 표현(compact representation), (3) 가변 길이 키(variable-length key)의 지원, (4) 다중 애트리뷰트 키(multiple-attribute key)의 지원, (5) 중복 키(duplicated key)의 지원, (6) 인덱스를 위한 시스템 카탈로그의 정의 (7) 외부 API(application programming interface)의 정의, (8) 효과적인 동시성 제어 방안, (9) 효율적인 백업 및 회복 방안 등이다.

본 논문의 구성은 다음과 같다. 제 2장에서는 개발의 대상이 되는 차세대 MMDBMS Tachyon에 관하여 간략히 기술한다. 제 3장에서는 관련 연구로서 기존에 제안된 다양한 인덱스 구조에 관하여 설명하고, 각 구조의 특성과 장단점에 관하여 논의한다. 제 4장에서는 캐쉬-인지 인덱스(cache-conscious index)의 개념과 이의 한 종류인 CSB^+-트리[20]에 관하여 설명한다. 제 5장에서는 Tachyon의 인덱스 관리자 개발 과정에서 획득한 다양한 실질적인 구현 이슈와 해결 방안에 관하여 자세히 논의한다. 제 6장에서는 Tachyon 인덱스를 위한 동시성 제어, 백업, 회복 전략을 제시한다. 제 7장에서는 제안된 인덱스 관리자의 성능을 다양한 실험을 통하여 검증한다. 끝으로, 제 8장에서는 본 논문을 요약하고, 결론을 내린다.

II. Tachyon

본 장에서는 현재 ETRI 실시간 DBMS 팀에서 개발 중인 차세대 DBMS Tachyon에 관하여 간략히 소개한다.

Tachyon의 주요 특징은 다음과 같다.

- 실시간 응용을 주요 지원 대상으로 하므로 마감 시간 개념을 지원하는 실시간 DBMS (real-time DBMS)이다.
- 효과적인 실시간 응용의 지원을 위하여 디스크가 아닌 주기억장치를 주요 저장 매체로서 사용하는 주기억장치 DBMS(main memory DBMS)이다.
- 다양한 응용 프로그램을 쉽게 수용하기 위하여 객체 지향 모델과 관계형 모델을 모두 지원하

는 객체-관계 DBMS(object-relational DBMS)이다.

Tachyon의 전체 시스템 아키텍처는 그림 2.1과 같다. 주기억장치 관리자(main-memory manager)는 전체 주기억장치 풀(pool)을 관리함으로써 상위 단계 관리자들이 요구하는 가변 길이의 주기억장치 더미(chunk)를 풀로부터 할당시켜 주고, 사용이 완료된 주기억장치 더미를 풀로 반환시켜 주는 기능을 제공한다.

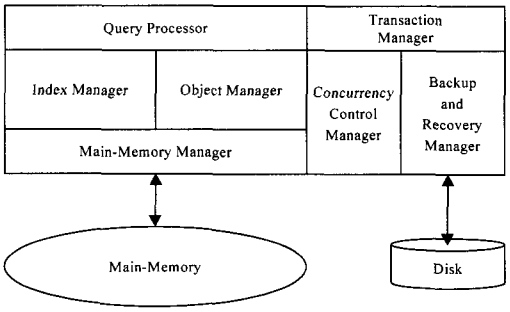

그림 2.1. Tachyon의 전체 시스템 아키텍처.

객체 관리자(object manager)는 사용자 데이터를 객체의 형태로 저장하고 관리하는 기능을 제공한다. 객체의 저장을 위하여 사용되는 고정 길이의 주기억장치 단위를 파티션(partition)이라 하며, 같은 종류의 객체들을 저장하기 위하여 사용되는 파티션들의 집합을 세그먼트(segment)라 한다. 하나의 데이터베이스는 다수의 세그먼트들의 집합으로 구성된다.

인덱스 관리자(index manager)는 사용자가 원하는 객체를 신속하게 검색하도록 하는 기능을 제공한다. 즉, 객체의 특정 애트리뷰트를 키로 선정하여 인덱스를 구성함으로써 특정 키 값을 갖는 객체를 데이터베이스로부터 직접 액세스하도록 한다. 현재, Tachyon에서는 완전 일치 질의(exact-match query)와 범위 질의(range query)를 모두 지원한다.

동시성 제어 관리자(concurrency control manager)는 여러 트랜잭션(transaction)들에 의하여 동시에 액세스되는 데이터베이스의 일관성(consistency)을 보장하는 기능을 제공한다. 즉, 동시에 수행되는 트랜잭션들의 수행 순서를 제어함으로써 데이터베이스가 항상 올바른 상태로 유지되도록 한다.

백업 및 회복 관리자(backup and recovery manager)는 시스템에서 발생되는 각종 오류로부터 데이터베이스를 보호하기 위한 기능을 제공한다. 백업 및 회복 관리자는 시스템에서 발생할 수 있는 다양한 오류에 대비하여 정상 동작 시 변경 연산에 대한 로깅(logging)[3]을 수행하고, 주기적으로 주기억장치 내의 데이터베이스를 디스크로 백업시킴으로써 오류 발생시 백업된 데이터베이스와 로깅 정보를 이용하여 전체 데이터베이스를 일관성 있는 상태로 회복시켜 준다.

트랜잭션 관리자(transaction manager)는 Tachyon내에서 수행되는 다수의 트랜잭션들의 수행 과정을 총괄적으로 제어한다. 특히, 마감 시간을 기반으로 하는 트랜잭션 스케줄링(scheduling)을 수행함으로써 실시간 응용을 효과적으로 지원하도록 한다.

질의 처리 관리자(query processor)는 Tachyon에서 정의하는 SQL(Structured Query Language)[6] 형태의 선언적인 언어(declarative language)를 최적화(optimize)하고, 하위 단계 관리자에 대한 호출로 변환해 주는 기능을 제공한다. DBMS 사용자는 이 선언적인 언어를 통하여 데이터베이스를 액세스하므로 응용 시스템을 보다 손쉽게 개발할 수 있다.

Ⅲ. MMDBMS를 위한 인덱스 구조

본 장에서는 관련 연구로서 MMDBMS를 위하여 기존에 제안된 인덱스 구조들에 대하여 간략히 소개하고 장단점을 지적한다.

3.1. 트리 인덱스

트리 인덱스는 트리 탐색(tree search)을 통하여 객체의 주소를 파악하므로 완전 일치 질의의 처리 성능은 해쉬 인덱스에 비하여 떨어지지만, 인덱스 엔트리들이 키 값의 순서대로 유지되므로 범위 질의의 처리 성능이 뛰어나다.

이진 탐색 트리(binary search tree)[12]는 가장 기본적인 트리 인덱스이며, 검색, 삽입, 삭제 연산이 매우 단순하다는 것이 장점이다. 그러나 트리의 균형성이 보장되지 않으므로 저장되는 키 값의 분포와 삽입 및 삭제되는 순서에 따라 검색 성능이 큰 영향을 받는다.

AVL-트리[13]는 이진 탐색 트리의 비 균형성 문제를 해결한다. 기본 구조는 이진 탐색 트리와 동일하지만, 회전 연산(rotation)을 통하여 모든 노드의 왼쪽 서브 트리와 오른쪽 서브 트리의 깊이(depth)의 차가 항상 1 이하가 되도록 제어한다. AVL-트리 내의 각 노드는 〈키 값, 이 키 값을 갖는 객체의 주

소)로 구성되는 하나의 엔트리, 두 하위 단계 서브 트리를 위한 두 개의 주소, 그리고 기타 제어 정보로서 구성된다. 따라서 하나의 키 값을 유지하기 위한 부가 정보의 양이 지나치게 많다.

T-트리[15]는 이러한 AVL-트리의 저장 공간 오버헤드 문제를 해결한다. 기본 구조와 균형을 유지하는 방법은 AVL-트리와 동일하나, 한 노드내에 다수의 엔트리를 정렬된 형태로 저장한다는 것이 AVL-트리와의 근본적인 차이점이다. 이 결과, 각 키 값당 요구되는 부가 정보의 양이 작아지며, 재균형을 위한 회전 연산의 수도 줄어든다.

B^+-트리[4]는 디스크 기반 DBMS에서 널리 사용되는 완전 균형 트리이다. 각 노드의 팬-아웃(fan-out)을 극대화함으로써 트리의 높이를 극소화하고, 이 결과 디스크 액세스의 최소화를 실현한다.

CSB^+-트리[20]는 최근의 캐쉬(cache)의 특성을 고려하여 개발된 B^+-트리의 새로운 변형이다. 각 노드 내에 (키 값의 수+1) 개의 자식 노드 주소를 저장하는 B^+-트리와는 달리, CSB^+-트리에서는 각 노드에 첫 번째 키와 대응되는 자식 노드 주소만을 저장한다. 나머지 자식 노드들은 첫 번째 자식 노드와 인접한 주기억장치 영역에 할당하도록 함으로써 나머지 자식 노드들의 액세스는 계산을 통하여 가능하다. 이 결과, 동일한 노드 내에 더 많은 수의 연관된 키 값들을 저장되므로 검색 성능이 향상된다.

3.2. 해쉬 인덱스

해쉬 인덱스는 계산을 통하여 객체의 주소를 파악하므로 완전 일치 질의의 처리 성능이 트리 인덱스에 비하여 뛰어난 반면, 범위 질의의 처리 성능은 떨어진다.

체인 버켓 해싱(chained bucket hashing)[14]은 해쉬 테이블(hash table)의 크기 변화가 전혀 허용되지 않는 정적인 구조이다. 해쉬 테이블의 크기가 데이터베이스의 크기에 적절하게 설정된 경우에는 매우 좋은 성능을 발휘할 수 있다. 그러나 해쉬 테이블의 크기가 데이터베이스 크기에 비하여 작은 경우에는 오버플로우 체인(overflow chain)으로 인한 검색 성능의 저하가 발생한다. 동적 환경에서는 데이터베이스 크기 예측이 사실상 불가능하므로 체인 버켓 해싱의 사용은 적절하지 못하다.

확장 해싱(extendible hashing)[7]은 객체 저장을 위한 데이터 페이지와 디렉토리로 구성되며, 디렉토리가 데이터베이스의 크기에 적합한 정도로 변화할 수 있는 동적인 구조이다. 디렉토리는 항상 2^k(k=0, 1, 2, ...) 개의 데이터 페이지 주소를 갖는다. 주어진 디렉토리를 이용하여 데이터 페이지의 오버플로우를 해결할 수 없는 경우에는 디렉토리의 크기를 두 배로 확장시킴으로써 변화되는 동적인 환경에 적응한다. 객체의 키 값들이 해쉬 공간의 특정 위치로 집중되는 경우에는 디렉토리의 비정상적인 증가로 인하여 심각한 저장 공간의 오버헤드를 초래할 수 있다.

선형 해싱(linear hashing)[16]은 데이터 페이지들을 물리적으로 연속된 주기억장치 공간상에 할당함으로써 디렉토리 없이 계산을 통하여 해당 데이터 페이지를 찾아낼 수 있다. 선형 해싱은 데이터 페이지의 오버플로우를 허용하며, 미리 정해진 순서에 의하여 데이터 페이지를 분할시킨다. 오버플로우 체인의 허용은 검색 성능 저하의 단점을 초래하게 되지만, 데이터 페이지의 분할 시점을 조절함으로써 저장 공간 이용률을 높일 수 있는 장점도 제공한다.

참고 문헌[15]에서는 MMDBMS에 적합하도록 선형 해싱을 변형함으로써 디렉토리를 가지는 변형된 선형 해싱(modified linear hashing)을 제안하였다. 변형된 선형 해싱은 디렉토리를 가지므로 데이터 페이지들이 물리적으로 연속할 필요가 없다. 또한, 객체를 전혀 갖지 않는 불필요한 데이터 페이지의 할당을 요구하지 않으므로 원래의 선형 해싱 보다 높은 저장 공간 이용률을 제공한다.

3.3. Tachyon 인덱스 구조의 선정 배경

Tachyon의 인덱스 구조 선정을 위하여 사용한 기준은 (1) 완전 일치 질의 및 범위 질의의 효과적인 처리, (2) 저장 공간 오버헤드, (3) 동적 환경으로의 적응 등이다.

먼저, 범위 질의 처리를 위한 트리 인덱스로는 트리의 균형성, 저장 공간 오버헤드, 캐쉬 활용 능력 등을 고려하여 CSB^+-트리를 선정하였다. 트리의 균형을 보장함으로써 키 값의 분포나 객체 삽입 및 삭제 순서에 영향을 받지 않고 일정한 질의 처리 성능을 보장할 수 있으며, 하나의 노드 내에 다수의 엔트리들을 저장함으로써 저장 공간 오버헤드가 상대적으로 작다. 또한, 일정한 크기를 가지는 노드들을 동적으로 할당하고, 반환하므로 동적 환경에서도 쉽게 적응할 수 있다. CSB^+-트리는 범위 질의를 효과적으로 처리할 수 있으며, 완전 일치 질의도 트리 탐색을 통하여 처리할 수 있다.

트리 인덱스와 해쉬 인덱스는 각각의 처리 대상에 차이가 있으므로 처음에는 트리 인덱스와는 별도로

완전 일치 질의의 효과적인 처리를 위하여 해쉬 인덱스를 Tachyon에서 추가로 지원하고자 하였다. 체인 버켓 해쉬과 같은 정적 구조를 가지는 해쉬 인덱스는 동적인 특성을 가지는 데이터베이스 환경에 적합하지 않으므로 선정 대상에서 우선적으로 제외하였다.

확장 해쉬이나 선형 해쉬과 같은 동적 구조를 가지는 해쉬 인덱스는 계산을 통하여 객체가 속하는 데이터 페이지의 위치를 파악하므로, 디렉토리(확장 해쉬과 변형된 선형 해쉬의 경우) 혹은 데이터 페이지들의 집합(선형 해쉬의 경우)들이 연속된 주기억장치 내에서 할당되어야 한다.

가장 단순한 해결 방안은 확장 가능한 최대 크기의 연속된 전용 공간을 미리 확보한 후, 이를 이후의 필요에 따라 사용하는 것이다. 그러나 이러한 전용 공간의 크기는 초기 시스템 설정 시 결정되어야 하는데, 객체의 삽입과 삭제가 빈번히 발생하는 동적인 환경에서는 이러한 크기 예측이 사실상 불가능하다. 만일, 확보된 전용 공간의 크기가 데이터베이스 크기에 비하여 작은 경우에는 이 공간의 일부만을 사용하게 되므로 주기억장치 공간의 낭비가 발생하며, 반대의 경우에는 오버플로우 체인의 발생으로 인한 성능 저하의 문제가 발생한다.

또 다른 해결 방안은 연속된 공간의 확장이 요구될 때마다 원래의 공간을 반환하고 이보다 큰 새로운 연속된 공간을 주기억장치 관리자에게 요구하는 것이다. 그러나 동적인 환경에서 DBMS가 장시간 동작한 후에는 주기억장치 풀 내에 연속된 공간이 점차 줄어들게 되므로 이러한 공간의 확보가 용이하지 않다는 문제점이 있다. 연속된 공간이 확보되지 않는 경우에는 디렉토리의 확장이 불가능하므로 오버플로우 체인으로 인한 성능 저하가 불가피하다.

이와 같이, 동적인 환경에서는 해쉬 구조를 위한 적절한 디렉토리(혹은 해쉬 공간)의 크기를 미리 예측할 수 없으므로 오버플로우 체인의 발생이 사실상 불가피하다. 검색 대상인 객체가 오버플로우 체인 상에 존재하는 경우에는 이 객체의 검색을 위하여 이 체인 상의 모든 객체들과의 직접적인 비교 연산이 필요하다. 따라서 검색 성능은 오버플로우 체인의 길이에 반비례하며, 이 오버플로우 체인의 길이는 사용 중인 디렉토리(혹은 현재의 해쉬 공간) 크기 및 데이터베이스 크기의 차이와 밀접한 관계가 있다.

해쉬 인덱스에서 오버플로우 체인이 발생하게 되는 경우, 객체 검색을 위한 오버플로우 체인의 탐색은 -트리에서 객체 검색을 위한 트리 탐색과 차이가 없게 된다. 특히, CSB^+-트리의 깊이는 객체 수에 대한 로그 함수로 증가하는 반면, 오버플로우 체인의 길이는 일차 함수로 증가하게 된다. 따라서 데이터베이스 크기에 대한 예측이 부정확한 경우에는 해쉬 인덱스가 트리 인덱스 보다 완전 일치 질의의 처리 성능이 더 떨어지는 경우가 발생한다. 특히, 이 오버플로우 체인의 길이는 CSB^+-트리의 깊이와는 달리 데이터베이스 크기 및 키 값의 분포에 크게 영향을 받게 되므로 최악의 경우의 길이가 보장되지 않는다. 이와 같은 이유로 인하여 본 Tachyon 개발에서는 범위 질의뿐만 아니라 완전 일치 질의의 처리에서도 CSB^+-트리를 사용하도록 결정하였다. 이러한 단일 인덱스 선정으로 인하여 동시성 제어 관리자와 백업 및 회복 관리자 등 다른 Tachyon 서브시스템들을 위한 개발의 노력이 절반으로 줄어드는 부가적인 장점도 얻을 수 있다.

Ⅳ. 캐쉬-인지 인덱스

본 장에서는 캐쉬의 특성을 고려하여 고안된 캐쉬-인지 인덱스(cache-conscious index)에 관하여 소개한다.

4.1. 캐쉬

캐쉬(cache)는 작고, 빠른 동적 RAM이며, 최근에 참조된 데이터를 저장함으로써 프로세스의 수행 속도를 높이는 역할을 한다[21]. 캐쉬 블록이란 캐쉬와 주기억장치간의 전송 단위이며, 그 크기로서 현재 32 바이트에서 128 바이트가 널리 사용된다.

참조하고자 하는 데이터가 캐쉬에 존재하는 경우, 이를 캐쉬 히트(hit)라 한다. 히트되는 경우, 프로세스의 수행은 CPU의 속도로 진행된다. 참조하고자 하는 데이터가 캐쉬에 존재하지 않는 경우, 이를 캐쉬 미스(miss)라 한다. 미스되는 경우, 해당 데이터는 주기억장치로부터 액세스되어야 하므로 프로세스의 수행은 주기억장치 액세스의 속도로 진행된다.

주기억장치에 대한 전통적인 가정은 "액세스되는 위치에 관계없이 주기억장치 액세스 비용은 일정하다"라는 것이었다. 그러나 이러한 전통적인 가정은 이와 같은 캐쉬와 주기억장치의 액세스 지연의 차이로 인하여 더 이상 유효하지 않게 되었다. 참고 문헌[1]에 따르면, 상용 DBMS를 이용한 응용 시스템의 수행 시간 중 많은 부분이 캐쉬 미스로 인하여 발생하는 것으로 나타났다. 따라서 캐쉬의 존재를 고려함으로써 캐쉬 히트율을 높인다면, 주기억장치의 액

세스 지연으로 인한 성능 저하를 크게 완화시킬 수 있다.

4.2. CSB+-트리

CSB$^+$-트리(cache-sensitive B$^+$-tree)[20]는 캐쉬의 특성을 활용하는 B$^+$-트리의 변형이다. 차수 d인 CSB$^+$-트리의 각 노드는 m(d < = m < = 2d)개의 키 값을 갖는다. CSB$^+$-트리는 하나의 노드의 자식 노드(child node)들을 하나의 노드 그룹(node group) 내에 함께 저장한다. 같은 노드 그룹 내의 노드들은 주기억장치 내에서 물리적으로 연속된 곳에 존재하며, 따라서 해당 노드 그룹 내의 첫 번째 노드의 주소를 이용한 계산을 통하여 액세스된다.

CSB$^+$-트리의 내부 노드에는 단 하나의 자식 노드 주소만이 포함되므로, B$^+$-트리에 비하여 훨씬 많은 수의 키 값을 저장할 수 있다. 이러한 사실은 두 가지 측면에서의 장점을 갖는다. 첫째, 이진 탐색(binary search)를 가정할 때, CSB$^+$-트리의 캐쉬 블록 내에서 비교 횟수가 B$^+$-트리의 캐쉬 블록 내에서 비교 횟수가 한번 더 발생하게 되므로, 결과적으로 검색을 위하여 필요한 캐쉬 블록의 수가 줄어든다. 둘째, 각 노드의 팬-아웃이 커지므로 저장 공간을 적게 요구한다.

그림 4.1은 차수가 1인 CSB$^+$-트리의 예를 나타낸 것이다[20]. 각 실선 사각형은 노드를 나타내며, 좀더 큰 각 점선 사각형은 노드 그룹을 나타낸다. 내부 노드로부터 아래쪽으로 향하는 화살표는 첫 번째 자식 노드에 대한 주소를 의미한다. 노드 그룹 내의 모든 노드들은 물리적으로 인접해 저장되어 있다.

V. 인덱스 관리자의 설계 및 구현

Tachyon 인덱스 관리자의 개발 목표는 캐쉬 미스의 수를 최소화함으로써 검색 성능을 극대화하는 것이다. 본 장에서는 Tachyon을 위한 CSB$^+$-트리 인덱스 관리자의 설계 및 구현에 관하여 상세히 논의한다. 먼저, 제 4.1절에서는 본 개발에서 채택하는 노드 엔트리(node entry)의 구조를 제시하고, 가변 길이 키(variable-length key), 다중 키(multi-attribute key), 중복 키(duplicate key)를 지원하는 방식에 관하여 설명한다. 제 4.2절에서는 캐쉬의 특성을 고려하여 본 개발에서 채택하는 인덱스 노드의 구조를 제시한다. 제 4.3절에서는 각 CSB$^+$-트리 인덱스의 관리를 위하여 시스템 카탈로그 내에서 관리하는 정보에 관하여 논의한다.

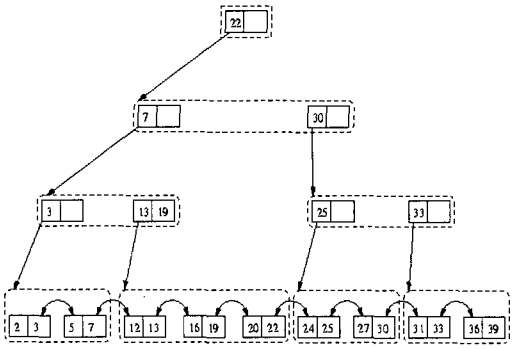

그림 4.1. 차수가 1인 CSB+-트리의 구조.

5.1. 엔트리 구조

참고 문헌[13]에서는 인덱스 엔트리에서 키 값을 직접 유지하지 않고 단순히 〈객체의 주소〉만을 유지하는 방법을 사용한 바 있다. 이것은 이 객체의 주소를 이용하여 키에 참여하는 애트리뷰트 값을 객체로부터 참조할 수 있으므로 키 값을 인덱스 내부에서 관리하지 않더라도 키 값의 비교가 가능하다는 점을 이용한 방식이다. 그러나 이러한 방식은 트리 노드 내에 비교 대상인 키 값이 존재하지 않으므로 이러한 비교를 위하여 매번 주기억장치 내에 존재하는 객체를 액세스해야 한다. 이러한 객체 액세스는 매번 캐쉬 미스를 유발시키므로 검색 및 삽입 삭제 성능이 저하된다.

본 개발에서는 이러한 문제를 해결하기 위하여 노드 내에 키 값을 직접 저장하는 방식을 채택한다. 각 CSB$^+$-트리 엔트리는 〈키 값, 이 키 값을 갖는 객체의 주소〉 형태의 구조를 갖는다. 이러한 방식은 참고 문헌[13] 방식과 비교하여 저장 공간의 오버헤드 측면과 알고리즘 복잡도 측면에서 좋지 못하지만, 노드 내에서의 키 값 비교 시 추가적인 캐쉬 미스를 방지할 수 있으므로 좋은 검색/삽입/삭제 성능을 얻을 수 있다.

가변 길이 키(variable-length key)[10]의 경우에는 키 값의 길이가 객체마다 다르므로, CSB$^+$-트리에서 엔트리 내의 키 값은 〈길이, 애트리뷰트 값〉의 형태로 표현된다. 반면, 고정 길이 키의 경우에는 모든 객체의 키 값이 동일한 길이를 가지므로, 길이 필드 없이 〈애트리뷰트 값〉만이 엔트리 내에 저장된다.

다중 애트리뷰트 키(multi-attribute key)[10]인 경우에는 엔트리 내의 키는 애트리뷰트들의 리스트로

구성된다. 각 애트리뷰트가 가변 길이인 경우에는 〈길이, 애트리뷰트 값〉으로 표현되며, 각 애트리뷰트가 고정 길이인 경우에는 〈애트리뷰트 값〉만으로 표현된다.

중복 키(duplicate key)[10]인 경우를 위하여 엔트리는 〈키 값, 이 키 값을 갖는 객체의 주소들의 리스트〉의 형태를 갖는다. 또한, 많은 키 값들의 중복으로 인하여 한 엔트리가 하나의 노드 내에 관리될 수 없는 경우에는 제 4.2절에서 설명하는 바와 같이 별도의 오버플로우 노드(overflow node)들을 연결하는 방식을 사용한다.

5.2. 노드 구조

CSB$^+$-트리는 내부 노드, 리프 노드, 오버플로우 노드의 세 가지 타입의 노드들로 구성된다.

내부 노드는 〈data[], p0, nEntries, freeOffset, type〉의 구조를 갖는다. 먼저, data[]는 실제 엔트리들이 저장될 수 있는 공간이며, 나머지 네 개의 필드들의 저장 장소를 제외한 모든 부분을 차지한다. nEntries는 현재 이 내부 노드상에 몇 개의 엔트리들이 저장되어 있는가를 나타낸다. freeOffset은 data[] 부분에서 빈 공간이 시작되는 위치를 가리키는 오프셋이다. 끝으로, type은 이 노드의 타입을 나타내는 부분이다. 캐쉬 블록의 크기가 일반적으로 256 바이트 이하이므로, 본 개발에서는 노드 내의 nEntries, freeOffset, type 필드를 모두 1바이트 unsigned char 타입으로 선언하였다.

기존의 B$^+$-트리 노드와의 큰 차이점은 각 엔트리와 대응되는 slot[i] 필드들이 존재하지 않는다는 것이다. 이러한 slot[i] 필드는 현재 노드의 존재하는 엔트리와 일대일 대응되며, i 번째 엔트리가 노드 내에서 시작되는 위치를 가리키는 오프셋이다. 이러한 slot[i]를 사용하는 이유는 가변 길이 키를 사용하는 경우에도 노드 내에서의 이진 탐색을 허용하기 위해서이다[10].

그러나 본 개발에서는 slot[i]의 개념을 사용하지 않는다. 각 엔트리마다 하나의 slot[i]를 할당해야 하는데, 이로 인하여 각 노드 내에 저장할 수 있는 엔트리의 수가 줄어든다. 이것은 캐쉬 미스의 횟수를 증가시키는 중요한 이유가 된다. 즉, 이로 인하여 노드의 팬-아웃을 줄이게 되며, 최악의 경우에는 트리의 깊이가 증가되는 결과를 초래할 수 있다.

slot[i]를 사용하지 않는 경우, 고정 길이 엔트리에 대해서는 여전히 이진 탐색이 가능하지만, 가변 길이 엔트리에 대해서는 선형 탐색(linear search)을 수행해야 한다. 그러나 가변 길이 엔트리의 경우 32바이트에서 128바이트 크기의 인덱스 노드 내에는 엔트리의 수가 많지 않다는 사실과 이 노드가 이미 캐쉬 내에 존재한다는 사실을 고려하면, 이러한 선형 탐색의 비용은 캐쉬 미스와 비교하여 큰 오버헤드가 아니다.

리프 노드는 〈data[], nEntries, freeOffset, (prefixLen), prevNode, nextNode, type〉의 구조를 갖는다. data[], nEntries, freeOffset, type은 내부 노드와 그 용도가 동일하다. prevNode와 nextNode는 각각 이 노드의 좌측과 우측에 존재하는 리프 노드를 연결 리스트로 묶기 위한 필드들이다. (prefixLen)은 가변 길이 키에 한하여 사용되며, 이 리프 노드 내에 존재하는 키 값들의 공통 접두어(common prefix)의 길이를 나타낸다. 즉, 이 노드 내의 엔트리들의 공통 접두어는 단 한번만 저장되며, 각 엔트리는 이 공통 접두어 부분을 제외한 나머지 부분의 키 값만을 포함한다. 이것은 저장 공간의 효율을 높임으로써 캐쉬 미스의 수를 줄이기 위한 것이다[1]. 특히, 데이터베이스가 대형화할수록 이 효과는 더욱 커진다. 내부 노드에서와 마찬가지로 노드 내의 nEntries, freeOffset, type, 〈prefixLen〉 필드를 모두 1바이트 unsigned char 타입으로 선언하였다.

오버플로우 노드는 〈data[], prevNode, nextNode, freeOffset, type〉의 구조를 갖는다. data[], freeOffset, type 등의 용도는 리프 노드와 동일하다. prevNode와 nextNode는 각각 이 노드의 앞쪽과 뒤쪽에 존재하는 오버플로우 노드를 연결 리스트로 묶기 위한 용도로서 사용된다. 내부 노드에서와 마찬가지로 노드 내의 freeOffset, type 필드를 모두 1바이트 unsigned char 타입으로 선언하였다.

5.3. CSB$^+$-트리를 위한 시스템 카탈로그 정보

시스템 카탈로그 내에는 CSB$^+$-트리 인덱스를 관리하기 위하여 필요한 정보들이 저장되어야 한다. Tachyon에서는 CSB$^+$-트리 인덱스에 관한 정보로서 그림 4.1과 같은 TreeInfo를 유지한다.

UorD는 현재 사용 중인 CSB$^+$-트리가 중복된 키를 허용하는가의 여부를 나타낸다. root는 해당 CSB$^+$-트리의 루트 노드의 주소를 나타내며,

1) 이러한 방법을 리프 노드에만 사용하는 이유는 내부 노드에는 모든 키 값들의 공통 접두어가 나타날 가능성이 매우 적기 때문이다.

numAttributes는 키를 구성하는 애트리뷰트의 수를 나타낸다. 즉, MAX개 까지의 attrDesc들이 존재하는 것을 허용함으로써 하나의 키가 다수의 애트리뷰트들로 구성되는 다중 애트리뷰트 키 기능을 제공한다. attrDesc[]는 키를 구성하는 각 애트리뷰트에 관한 정보이다. 키 값의 비교는 키를 구성하는 각각의 애트리뷰트 값의 비교를 반복함으로써 수행된다.

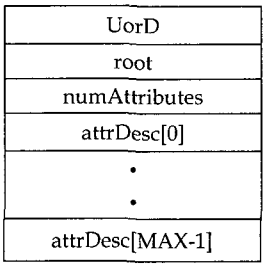

그림 5.1. CSB+-트리를 위한 시스템 카탈로그내의 정보.

5.4. 외부 APIs

Tachyon은 객체지향 프로그래밍 언어인 C++를 이용하여 개발되었다. CSB^+-트리 관리자는 CSBtreeIndex라는 이름의 클래스를 통하여 외부 API를 제공하며, 이를 이용하여 응용 프로그램을 쉽게 개발할 수 있다. 본 절에서는 CSB^+-트리 관리자가 제공하는 구체적인 외부 API에 관하여 논의한다.

CSBtreeIndex 클래스는 MemAllocPtr, CurrentPosition, CSBtreeInfo의 네 가지 멤버 변수를 갖는다. MemAllocPtr은 CSB^+-트리 노드와 같은 주기억장치 더미를 동적으로 할당해 주는 Tachyon의 주기억장치 관리자 클래스 인스턴스에 대한 포인터이며, CurrentPosition는 CSB^+-트리를 이용하여 범위 질의를 처리할 때, 현재 처리 중인 엔트리의 주소를 갖는다. 끝으로, CSBteeInfo는 제 4.2절에서 언급한 바와 같이 루트 노드의 주소, 키를 구성하는 애트리뷰트들의 특성 등 해당 CSB^+-트리에 관한 정보가 저장된 시스템 카탈로그 내의 엔트리 주소를 갖는다.

CSBtreeIndex 클래스는 CSBtreeIndex(), insert(), delete(), deleteCurrent(), search(), getNext(), getPrev(), getFirst(), getLast() 등의 아홉 가지 함수를 외부 API로 제공한다.

CSBtreeIndex(MemAllocPtr, CSBtreeInfo)는 CSBtreeIndex 클래스의 구성자(constructor)인 동시에, 해당 CSB^+-트리를 오픈하는 역할을 한다. 즉, 해당 CSB^+-트리에 관한 정보를 가지는 엔트리를 시스템 카탈로그에서 찾아 멤버 변수인 CSBtreeInfo가 이를 가리키도록 한다. 또한, 이후의 수행에서 주기억장치 할당을 위하여 호출이 필요한 주기억장치 관리자의 클래스 인스턴스를 멤버 변수인 MemAllocPtr이 가리키도록 한다.

insert(OID)는 OID가 가리키는 객체와 대응되는 엔트리를 CSB^+-트리 내에 삽입하는 함수이며, delete(OID)는 OID가 가리키는 객체와 대응되는 엔트리를 CSB^+-트리로부터 삭제하는 함수이다. 객체 삽입 시에는 실제 객체를 먼저 삽입한 후에 이에 대한 엔트리를 CSB^+-트리에 삽입하고, 반대로 삭제 시에는 CSB^+-트리로부터 엔트리를 삭제한 후에 실제 대응되는 객체를 삭제한다. 특히, 삭제의 경우, 같은 키 값을 갖는 다른 객체의 삭제가 발생하면 안되므로 같은 키 값을 갖는 엔트리에 대해서도 OID 값이 동일한가의 여부를 다시 점검한다.

인덱스 관리자에서 처리되는 질의는 범위 질의를 가정하며, 완전 일치 질의는 범위 질의의 특별한 형태로 간주한다. search(Key1, Op1, Key2, Op2, OID)는 범위 질의를 만족하는 첫 번째 객체를 찾아내는 함수이다. Key1과 Key2는 범위 질의의 시작과 끝 조건 값을 주기 위하여 사용되며, Op1과 Op2는 <, <=, >, >=, =, NULL 중의 한 값을 갖는 비교 연산자이다. OID는 이러한 조건을 만족하는 첫 번째 객체의 주소를 반환하기 위하여 사용된다. 예를 들어, search(5, <, 15, <, OID)가 호출되면, CSB^+-트리 검색을 통하여 5 보다 크고, 15 보다 작은 첫 번째 객체가 OID에 반환된다. 또한, CSBtreeIndex 클래스의 멤버 변수 CurrentPosition에 이 객체와 대응되는 CSB^+-트리 엔트리의 주소가 〈노드 주소, 노드 내에서의 엔트리 주소〉의 형태로 설정된다.

getNext(Key2, Op2, OID)는 search()를 통하여 범위 질의를 만족하는 첫 객체를 찾은 후, 이후의 객체들을 차례로 찾아내기 위하여 사용된다. 즉, CSB^+-트리의 중위 순회를 통하여 CurrentPosition 이후에 나타나는 엔트리가 가리키는 객체의 키 값이 Key2, Op2의 조건을 만족하는가를 점검한 후, 만족하는 경우에는 그 객체의 주소 값을 OID를 통하여 반환한다. 즉, getNext()의 호출 시에는 새로운 트리 탐색이 아니라 중위 순회를 이용하게 되므로 다음 객체의 주소를 효과적으로 찾을 수 있다. 찾은 후에는 이 객체와 대응되는 엔트리를 가리키도록 CurrentPosition를 새롭게 설정한다. 요약하면, 상위 단계의 사용자는 search() 함수의 일회 호출과 getNext() 함수의 반복 호출을 통하여 범위 질의를 처리할 수 있다.

본 개발에서는 이 외에도 getPrev(Key2, Op2, OID) 함수를 제공한다. getPrev() 함수는 getNext() 함수와 그 기능이 동일하며, 단지 키 값 크기의 역순으로 엔트리들을 액세스 한다는 것이 차이점이다. 따라서 키 값 크기의 역순으로 범위 질의를 처리하기 위해서는 일회의 search() 함수 호출과 getPrev() 함수의 반복된 호출이 필요하다. 완전 일치 질의는 Op1과 Op2를 모드 =로 설정함으로써 동일한 방식으로 처리할 수 있다.

deleteCurrent() 함수는 CSBtreeIndex 클래스의 멤버 변수인 CurrentPosition이 나타내는 엔트리를 CSB^+-트리로부터 삭제하는 함수이다. 이 함수는 search() 함수 및 getNext()(혹은 getPrev()) 함수와 결합함으로써 주어진 키 값의 범위를 만족하는 엔트리들을 삭제하기 위하여 사용된다. 즉, 사용자가 어떤 키 값의 범위를 만족하는 객체들을 모두 삭제하고자 하는 경우에는 먼저 search() 함수를 호출하여 조건을 만족하는 첫 번째 객체와 대응되는 엔트리를 CSB^+-트리로부터 찾는다. 이때, 해당 엔트리의 물리적인 위치는 CurrentPosition에서 관리된다. 실제 객체는 엔트리 내의 OID를 이용하여 객체 관리자를 호출함으로써 삭제시키고, 엔트리는 deleteCurrent() 함수를 이용하여 CSB^+-트리로부터 삭제시킨다.

이후에는 getNext()(혹은 getPrev()) 함수를 이용하여 조건을 만족하는 다음 객체와 대응되는 엔트리를 CSB^+-트리로부터 찾는다. 이 결과, 해당 엔트리의 물리적인 위치는 CurrentPosition에서 관리된다. 위와 마찬가지 방식으로 실제 객체는 엔트리 내의 OID를 이용하여 객체 관리자를 호출함으로써 삭제시키고, 엔트리는 deleteCurrent() 함수를 이용하여 CSB^+-트리로부터 삭제시킨다. 이러한 작업을 getNext(혹은 getPrev()) 함수에 의하여 만족하는 엔트리가 존재하지 않을 때까지 반복한다.

구현 시 유의해야 할 것은 CurrentPosition 위치의 재 설정이다. 즉, CurrentPosition이 가리키는 엔트리가 제거되면서 다음 엔트리가 앞으로 당겨지므로 getNext()(혹은 getPrev())를 호출하면, CurrentPosition이 가리키는 이 엔트리를 무시하고 지나가게 된다. 따라서 deleteCurrent() 함수에서는 해당 엔트리의 삭제 후, CurrentPosition이 직전 엔트리를 가리키도록 조정한다. 삭제로 인하여 CSB^+-트리의 구조적인 변화가 발생할 수 있으므로 재 설정 작업은 deleteCurrent() 함수의 최종 작업 단계에서 수행하도록 한다.

끝으로, getFirst()는 각각 해당 CSB^+-트리 상의 가장 작은 키 값을 가지는 객체와 대응되는 엔트리를 찾아 CurrentPosition을 설정해 주는 함수이며, getLast()는 CSB^+-트리 상의 가장 큰 키 값을 가지는 객체와 대응되는 엔트리를 찾아 CurrentPosition을 설정해 주는 함수이다. 이 두 함수는 죠인 연산의 처리와 같이 특별한 검색 조건 없이 키 값의 순서대로 전체 객체들을 액세스해야 하는 경우에 사용된다.

VI. 동시성 제어, 백업, 회복 기능의 지원

본 장에서는 Tachyon의 인덱스에 대하여 동시성 제어와 백업 및 회복 기능을 지원하는 효과적인 방안을 소개한다.

6.1. 동시성 제어

인덱스는 대부분의 트랜잭션들에 의하여 공통적으로 액세스되는 데이터이므로 일반 객체의 동시성 제어를 위하여 사용되는 이단계 로킹 규약(two-phase locking protocol)[9]을 동일하게 적용한다면 매우 심각한 동시성의 저하가 발생한다. 예를 들어, CSB^+-트리의 루트 노드에 대하여 각 트랜잭션의 종료 시까지 락을 걸도록 한다면, 트랜잭션들을 순차적으로 수행하는 것과 동일한 결과를 초래하게 된다.

본 연구에서는 이를 해결하기 위하여 각 CSB^+-트리마다 고유의 래치(latch)[17][18]를 둠으로써 동시성을 제어하는 방식을 제시한다. 각 트랜잭션은 CSB^+-트리를 액세스하기 직전 해당되는 래치를 걸고[2], 이에 대한 액세스가 끝나면 이 래치를 즉시 풀어준다. 이러한 래치 만으로 동시성 제어가 가능한 이유는 인덱스가 단지 사용자 객체의 효과적인 검색을 위하여 필요한 메타 데이터(meta data)이기 때문이다. 메타 데이터는 이단계 로킹 규약을 적용하여 논리적 일관성을 보장할 필요가 없으며, 래치를 이용하여 물리적 일관성만을 보장하면 된다[17][18]. 제안된 방식을 이용함으로써 CSB^+-트리의 물리적 일관성을 보장할 수 있으며, CSB^+-트리를 액세스하는 동안만 래치를 걸게 되므로 동시성을 크게 향상시킬 수 있다.

2) 동시성의 극대화를 위하여 단순한 참조 연산을 위한 공유 래치(shared latch)와 변경 연산을 위한 배제 래치(exclusive latch)를 별도로 제공한다.

참고 문헌[17]에서는 이와 유사한 개념을 기반으로 래치를 이용한 ARIES/IM을 제시하였다. 이 방식은 디스크 기반 DBMS의 B-트리를 대상으로 하며, B-트리 노드마다 래치를 할당한다. 트랜잭션은 각 노드를 액세스하기 직전에 해당 래치를 걸고, 액세스가 끝나면 이 래치를 풀어준다. 만일, 트랜잭션이 수행 도중 B-트리의 일관성에 문제가 있음을 감지하면, 트리 전체에 할당된 래치(tree latch)를 획득하도록 함으로써 일관성이 깨어진 인덱스내의 정보를 참조하지 못하도록 한다.

ARIES/IM은 전체 트리가 아닌 액세스하고자 하는 노드에만 래치를 걸게 되므로 본 연구에서 제시하는 방식 보다 동시성을 좀더 높일 수 있다는 것이 장점이다. 그러나 이러한 동시성 향상의 효과는 디스크 기반 DBMS에서는 매우 크게 부각되지만, MMDBMS에서는 그 효과가 상대적으로 미미하다. 디스크 기반 DBMS의 B-트리는 디스크 내에 존재하므로 트리 탐색은 디스크 액세스를 유발한다. 따라서 한 트랜잭션이 트리 탐색을 위하여 다수의 디스크 액세스를 수행하는 동안 다른 트랜잭션들은 이를 대기해야 한다. 그러나 MMDBMS의 CSB^+-트리는 주기억장치 내에 상주하므로 트리 탐색이 매우 빠르게 진행된다. 따라서 한 트랜잭션이 다른 트랜잭션의 트리 탐색을 대기하는 시간은 무시할 만큼 짧다.

ARIES/IM은 그 수행 메카니즘이 매우 복잡하며, 각 트랜잭션이 걸거나 풀어줘야 하는 래치의 수가 많다. 디스크 기반 DBMS에서는 데이터 검색 비용 중 래치 처리 비용이 크게 부각되지 않으므로 동시성의 향상을 위하여 ARIES/IM이 매우 유용하다. 그러나 MMDBMS에서는 데이터베이스 전체가 주기억장치 내에 상주하므로 데이터 검색 비용 중 래치 처리 비용이 차지하는 비중이 매우 크게 나타난다. 따라서 본 연구에서는 ARIES/IM 대신 전체 트리에 래치를 거는 방식을 채택하였다.

6.2. 백업 및 회복 기능

백업 및 회복 관리자는 시스템에서 발생할 수 있는 다양한 오류에 대비하여 정상 동작 시 변경 연산에 대한 로그 레코드를 기록하는 로깅(logging)[3] 작업을 수행하고, 주기적으로 주기억장치 내의 데이터베이스를 디스크로 백업시킴으로써 오류 발생시 백업된 데이터베이스와 로깅 정보를 이용하여 데이터베이스를 일관성 있는 상태로 회복시켜 준다[11][23].

디스크 기반 DBMS에서는 인덱스의 변경 연산에 대해서도 대응되는 로그 레코드를 기록하는 방식을 사용한다[17][18]. 즉, 인덱스의 각 페이지에서 발생되는 변경 연산과 대응되는 로그 레코드를 기록하며, 트랜잭션 오류(transaction failure)나 시스템 오류(system failure)가 발생하였을 때 이를 이용하여 인덱스를 일관된 상태로 회복한다. 이 방식은 하나의 엔트리를 인덱스에 삽입하는 경우, 이 삽입과 연관된 다수의 로그 레코드를 발생시킬 수 있다.

MMDBMS에서는 디스크 액세스를 유발하는 로깅 작업이 전체 MMDBMS 성능 저하의 가장 큰 원인이다. 따라서 기존의 디스크 기반 DBMS의 인덱스 로깅 방식을 MMDBMS에 그대로 적용하는 경우, 다수의 로그 레코드 생성으로 인하여 전체 시스템 성능이 크게 저하된다. 본 연구에서는 인덱스가 객체들을 기반으로 생성된 메타 데이터라는 점에서 착안하여 시스템 오류의 발생 시 먼저 디스크 기반 DBMS와 동일한 방식으로 객체들을 회복한 후, 이를 이용하여 인덱스를 재 생성하는 방식을 제안한다.

이 방식은 다음과 같은 장점을 갖는다. 첫째, 정상 동작 시, 시스템 오류로부터의 회복을 위하여 인덱스에 대한 로그 레코드를 생성할 필요가 없게 된다. 따라서 로깅 작업을 위한 디스크 액세스 수가 크게 줄어들게 되므로 트랜잭션 처리 성능을 개선할 수 있다. 둘째, 주기억장치 내의 데이터베이스를 디스크로 백업하는 경우, 인덱스를 백업 대상에서 제외할 수 있으므로 백업 시간을 크게 단축시킬 수 있다. 셋째, 시스템 오류의 발생 시, 디스크 기반 DBMS의 방식을 이용하여 인덱스를 회복하기 위해서는 다수의 로그 레코드들을 디스크로부터 액세스해야 하는 반면, 제안된 방식은 주기억장치 내의 객체들을 기반으로 인덱스를 생성할 수 있으므로 시스템 오류로부터의 회복 처리 성능이 개선된다.

시스템이 정상적으로 동작하는 중에 발생하는 트랜잭션 오류에 대해서는 인덱스를 올바르게 회복할 수 있어야 한다. 본 연구에서는 이를 위하여 인덱스의 변경 연산과 대응되는 논리적인 로그 레코드를 기록하는 방식을 사용한다. 즉, 새로운 인덱스 엔트리가 삽입되거나 삭제될 때마다 이와 대응되는 단 하나의 로그 레코드를 기록하는 것이다. 예를 들어, 인덱스 I1에 새로운 엔트리 e1이 삽입되는 경우에는 〈Insert, I1, e1〉의 형태를 갖는 로그 레코드를 기록한다. 만일, 트랜잭션 오류가 발생되면, 회복 관리자는 이 로그 레코드 연산의 역 작업인 e1을 I1에서 제거하는 연산을 수행하면 된다. 이러한 논리적인 로그 레코드는 트랜잭션의 수행 기간 동안만 주기억

장치 내에서 유지하며, 해당 트랜잭션이 종료됨과 동시에 이를 무시함으로써 디스크 액세스를 방지시킬 수 있다.

Ⅶ. 성능 평가

본 장에서는 다양한 실험을 통하여 본 연구에서 개발된 인덱스 관리자의 우수성을 검증한다.

7.1. 실험 환경

성능 평가를 위하여 사용된 하드웨어 플랫폼은 Sun Ultra 10 워크스테이션이다. Sun Ultra 10은 UltraSPARC IIi 프로세서(440MHz의 속도와 2MB의 캐쉬)를 사용하였고, 1GB의 주기억장치를 장착하고 있다. 운영 체제로는 SunOS 5.7을 사용하였고, 컴파일러로는 GNU의 C++ 컴파일러 g++ 3.0.1 버전을 사용하였다.

현재 개발된 Tachyon CSB$^+$-트리 인덱스 관리자의 비교 대상으로는 Tachyon에서 기존에 사용하였던 T-트리 관리자를 사용하였다. 두 경우 모두 노드 크기로는 캐쉬 블록과 동일한 64 바이트를 사용하였다. 고정 길이, 가변 길이, 다중 키에 대해서 각각 랜덤하게 생성한 100만개의 데이터를 삽입, 삭제, 검색하면서 시간과 저장 공간의 크기를 측정하였다.

7.2. 시간 분석

그림 7.1, 7.2, 7.3은 고정 길이(4 바이트 정수형), 단일, 비중복 특성을 갖는 키에 대해서 각각 삽입, 완전 일치 질의, 범위 질의를 수행한 실험 결과이다.

그림 7.1은 100만 개의 키 값들을 20만 개 단위로 삽입하면서 삽입에 걸린 시간을 측정한 결과이다. 가로축은 삽입한 키 값들의 수이고 세로축은 처리 시간을 의미한다. CSB$^+$-트리의 삽입 성능이 T-트리와 비교하여 약 1.8배 개선되는 것으로 나타났다. 트리 탐색시 T-트리는 객체 주소를 통해서 키 값을 파악하므로 캐쉬 미스가 많이 발생되는 반면, CSB$^+$-트리는 노드 내에 이미 키 값이 저장되어 있기 때문에 캐쉬 미스가 T-트리에 비해서 상대적으로 적게 발생된다. 또한, CSB$^+$-트리 노드의 압축된 저장 방식으로 인하여 트리 탐색 시 캐쉬 미스가 덜 발생된다. 이러한 캐쉬 미스 발생의 수가 삽입 성능에 반영된 것이다.

그림 7.2는 100만 개의 키 값들을 20만개 단위로 삽입하면서 각 경우에서 10만 번의 완전 일치 질의를 수행한 실험 결과를 나타낸다. 가로축은 삽입한 키 값들의 수이고 세로축은 처리 시간을 의미한다. 이 실험에서도 CSB$^+$-트리가 T-트리보다 1.8배 정도 빠른 성능을 보이고 있음을 알 수 있다. 키 값들이 20만개 삽입되었을 때의 트리의 높이와 100만개가 삽입되었을 때의 트리의 높이의 차가 1에 불과하기 때문에 그래프는 크게 변화하지 않고 완만한 경사를 이루고 있다. 전술한 바와 같이, CSB$^+$-트리는 트리 탐색시 캐쉬 미스가 일어날 가능성이 적기 때문에 탐색 성능에서 T-트리 보다 좋은 것으로 나타났다.

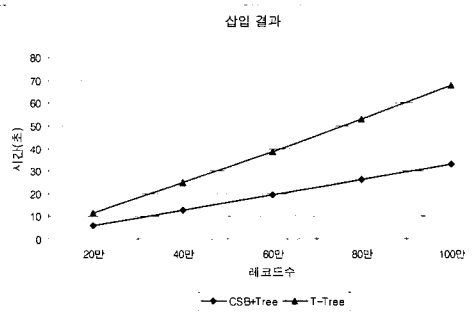

그림 7.1. 삽입 성능.

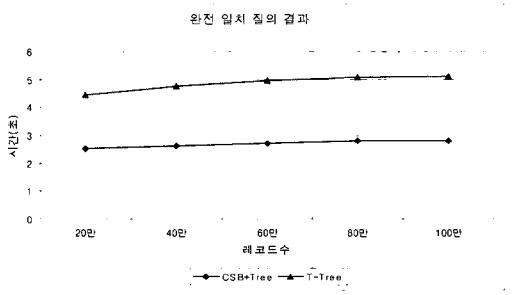

그림 7.2 완전 일치 질의의 성능.

그림 7.3은 100만 개의 키 값들을 삽입 한 후, 5%에서 20%까지 5%단위로 선택률을 변화시키면서 범위 질의를 수행한 실험 결과이다. 가로축은 선택률을 나타내고 세로축은 처리 시간을 의미한다. 이 실험은 CSB$^+$-트리가 T-트리보다 1.6배 정도 빠른 성능을 보이고 있다.

그림 7.4, 7.5, 7.6은 가변 길이(7~10 바이트의 문자형), 단일, 비중복 특성을 갖는 키에 대해서 각각 삽입, 완전 일치 질의, 범위 질의를 수행한 실험 결과이다. 그림 7.4는 100만 개의 키 값들을 삽입하면서 20만 개 단위로 삽입에 걸린 시간을 측정한 결과이다. 가로축은 삽입한 키 값들의 수이고 세로축은

처리 시간을 의미한다. CSB$^+$-트리의 삽입 성능이 T-트리 보다 1.5배정도 향상된 것으로 나타났다. 그림 7.5는 100만 개의 키 값들을 20만개 단위로 삽입하면서 각 경우에서 10만 번의 완전 일치 질의를 수행한 실험 결과를 나타낸다. 가로축은 삽입한 키 값들의 수이고 세로축은 처리 시간을 의미한다. 실험 결과, CSB$^+$-트리가 T-트리 보다 1.7배정도 빠른 성능을 보였다. 그림 7.6은 100만 개의 키 값들을 삽입 한 후, 5%에서 20%까지 5%단위로 선택률을 변화시키면서 범위 질의를 수행한 실험 결과이다. 가로축은 선택률을 나타내고 세로축은 처리 시간을 의미한다. 실험 결과, CSB$^+$-트리가 T-트리 보다 1.5배정도 빠른 성능을 나타냈다.

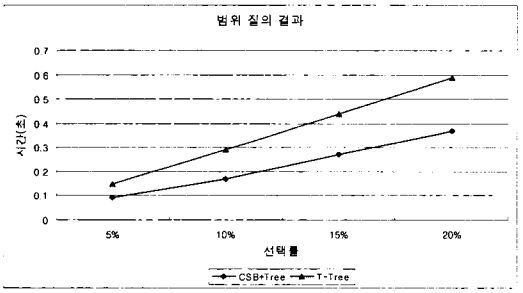

그림 7.3. 범위 질의의 성능.

그림 7.7, 7.8, 7.9는 고정 길이(4 바이트 정수형), 다중, 비중복 특성을 갖는 키에 대해서 각각 삽입, 완전 일치 질의, 범위 질의를 수행한 실험 결과이다.

그림 7.7은 키를 구성하는 애트리뷰트를 1, 2, 3개로 변화시키면서 100만 개의 키 값들을 삽입할 때까지 걸린 시간을 측정한 결과이다. 가로축은 삽입한 키를 구성하는 애트리뷰트의 수이고 세로축은 처리 시간을 의미한다. 실험 결과, 애트리뷰트의 수가 증가함에 따라 T-트리의 성능은 변화가 없는 반면, CSB$^+$-트리의 성능은 점차 저하되는 것을 볼 수 있다. 그 이유는 CSB$^+$-트리에서는 애트리뷰트의 수가 증가함에 따라 키의 길이가 함께 증가하기 때문이다. 따라서 CSB$^+$-트리의 높이가 증가하므로 트리 탐색 시간 및 노드 분할 시간이 함께 커지는 것이다. 반면, T-트리는 키 값의 길이가 애트리뷰트의 수에 영향을 받지 않으므로 거의 일정한 삽입 성능을 보여준다.

그림 7.8은 키를 구성하는 애트리뷰트의 수를 변화시키면서 100만 개의 키 값들을 삽입한 후 10만 번의 완전 일치 질의를 수행한 실험 결과를 나타낸다. 가로축은 키를 구성하는 애트리뷰트의 수이고 세로축은 완전 일치 질의의 처리 시간을 의미한다. 완전 일치 질의 성능도 그림 7.8을 통해서 본 실험과 비슷한 결과를 보여준다. 이는 이전 실험에서와 같이 CSB$^+$-트리의 경우에는 키를 구성하는 애트리뷰트의 수에 영향을 받지만 T-트리는 영향을 받지 않기 때문이다.

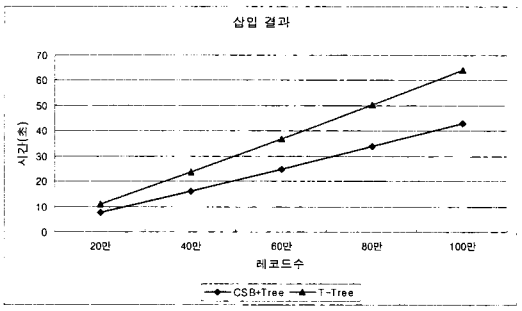

그림 7.4. 삽입 성능

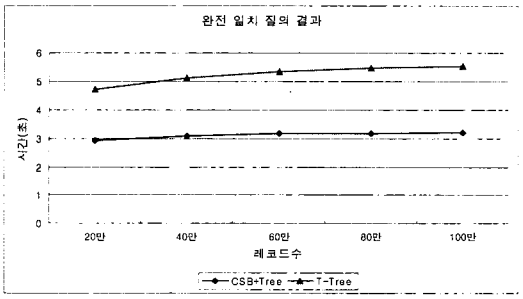

그림 7.5. 완전 일치 질의의 성능.

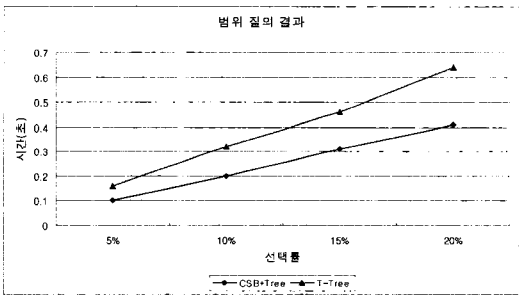

그림 7.6. 범위 질의의 성능.

그림 7.9는 구성하는 애트리뷰트의 수를 변화시키면서 100만 개의 키 값들을 삽입한 후 10만 번의 범위 질의를 수행한 실험 결과를 나타낸다. 가로축은 선택률을 나타내고, 세로축은 범위 질의의 처리 시간을 의미한다. 이 실험 역시 이전 실험과 같은 이유에서 거의

비슷한 경향을 보여 주고 있다.

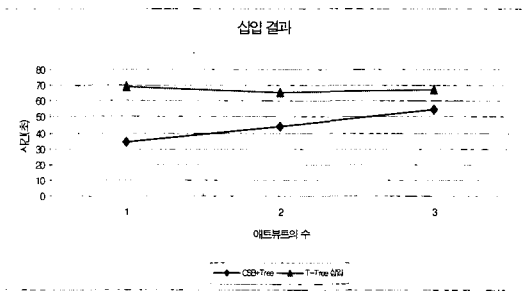

그림 7.7. 삽입 성능.

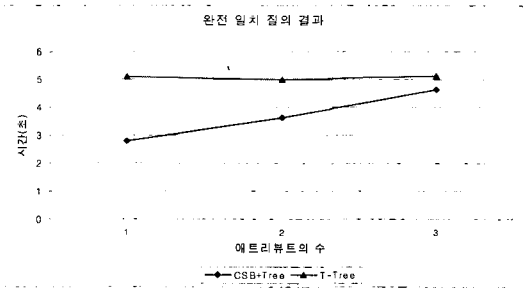

그림 7.8. 완전 일치 질의의 성능.

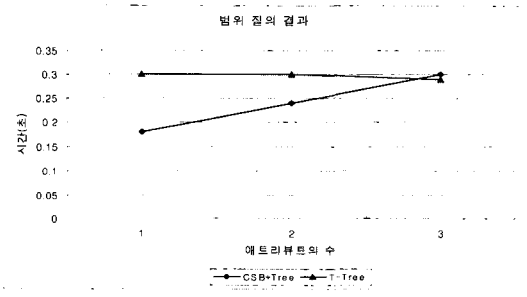

그림 7.9. 범위 질의의 성능

7.3. 공간 분석

그림 7.10은 가변길이(7-10 바이트의 문자형), 단일, 비중복 특성을 갖는 키에 대해서 100만 개까지의 키 값들을 삽입하고 모든 키 값들을 삭제하면서 CSB^+-트리와 T-트리의 저장 공간 사용량을 비교한 실험의 결과이다. 가로축은 키 값들의 수이고 세로축은 사용하는 저장 공간의 크기이다.

CSB^+-트리가 T-트리가 보다 약 두 배 정도의 공간을 사용함을 볼 수 있다. 두 트리의 높이는 비슷하지만 CSB^+-트리는 노드 그룹 단위로 저장 공간을 할당받게 되기 때문에 노드 단위로 저장 공간을 할당 받는 T-트리에 비해서 많은 공간을 차지한다. 또한, CSB^+-트리의 저장 공간 사용량은 키의 길이에 크게 영향을 받게 된다. 키의 길이가 길어지게 되면 노드에 저장할 수 있는 키의 수가 감소하기 때문이다. 따라서 더 많은 저장 공간을 요구하게 된다. 반면, T-트리의 경우는 키의 길이가 아무리 길더라도 노드에 저장되는 엔트리의 수에 영향을 주지 않기 때문에 저장 공간 사용량은 키의 길이에 따라 변화하지 않고 일정하다. 또한, CSB^+-트리의 경우, 삽입 부분의 그래프는 선형적으로 증가하지만 삭제 부분의 그래프는 지연 삭제(lazy-deletion)의 영향으로 점점 큰 폭으로 저장 공간의 사용량이 감소함을 보여 준다.

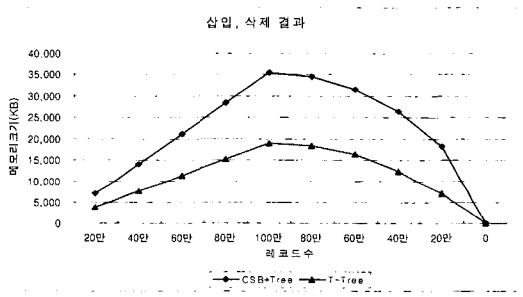

그림 7.10. 저장 공간 사용량.

Ⅷ. 결 론

주기억장치를 데이터베이스의 저장 매체로서 사용하는 MMDBMS는 디스크 액세스로 인하여 응답 시간이 지연되는 디스크 기반 DBMS의 문제를 근본적으로 해결한다. 최근, 주기억장치 기술의 발전으로 인하여 MMDBMS를 실제 응용에 적용하는 사례들이 확대되고 있다.

본 연구에서는 차세대 MMDBMS Tachyon의 인덱스 관리자를 개발하였다. 인덱스 관리자는 객체에 대한 빠른 검색 기능을 지원하는 DBMS의 필수 서브 시스템이다. 본 논문에서는 Tachyon의 인덱스 관리자 개발 중에 경험한 실질적인 구현 이슈들을 언급하고, 이들에 대한 해결 방안을 제시하였다.

본 논문에서 다루었던 주요 이슈들은 (1) 캐쉬(cache)의 효과적인 사용, (2) 인덱스 엔트리 및 인덱스 노드의 집약적 표현(compact representation), (3) 가변 길이 키(variable-length key)의 지원, (4) 다중 애트리뷰트 키(multiple-attribute key)의 지원,

(5) 중복 키(duplicated key)의 지원, (6) 인덱스를 위한 시스템 카탈로그의 정의 (7) 외부 API(application programming interface)의 정의, (8) 효과적인 동시성 제어 방안, (9) 효율적인 백업 및 회복 방안 등이다. 이러한 공헌을 통하여 향후 MMDBMS 개발자들의 시행 착오를 최소화할 수 있으리라 생각된다.

참 고 문 헌

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a Modern Processor: Where Does Time Go?," In Proc. Intl. Conf. on Very Large Data Bases, VLDB, pp. 266-277, 1999.

[2] A. Ammann, M. Hanrahan, and R. Krishnamurthy, "Design of a Memory Resident DBMS," In Proc. Intl. Conf. on COMPCON, Feb. 1985.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987.

[4] D. Comer, "The Ubiquitous B-Trees," ACM Computing Surveys, Vol. 11, No. 2, pp. 121-137, 1979.

[5] D. DeWitt et al., "Implementation Techniques for Main Memory Database Systems," In Proc. Intl. Conf. on Management of Data, pp. 1-8, ACM SIGMOD, 1984.

[6] Elmasri, R. and Navathe, S. B., Fundamentals of Database Systems, Second Edition, Benjamin/Cummings Publishing Company, Inc., 1994.

[7] R. Fagin et al., "Extendible Hashing: A Fast Access Method for Dynamic Files," ACM Trans. on Database Systems, Vol. 4, No. 3, pp. 315-344, 1979.

[8] H. Garcia-Molina and K. Salem, "Main Memory Database Systems: An Overview," IEEE Trans. on Knowledge and Data Engineering, Vol. 4, No. 6, pp. 509-516, 1992.

[9] J. Gray et al., "Granularity of Locks in a Shared Data Base," In Proc. Intl. Conf. on Very Large Data Bases, pp. 428-451, Sept. 1975.

[10] J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufman Publishers, 1993.

[11] T. Haeder and A. Reuter, "Principles of Transaction-Oriented Recovery," ACM Computing Surveys, Vol. 15, No. 4, pp. 287-317, Dec. 1983.

[12] E. Horowitz, S. Sahni, and S. Freed, Fundamentals of Data Structures in C, Computer Science Press, 1993.

[13] S. Kim et al., "Design and Implementation of the Index Manager in the Main Memory DBMS," In Proc. Intl. Symp. on Databases and Applications(DBA 2002), pp. 473-478, 2002.

[14] D. Knuth, The Art of Computer Programming, Addison-Wesley, 1973.

[15] T. Lehman and M. Carey, "A Study of Index Structures for Main Memory Database Management Systems," In Proc. Intl. Conf. on Very Large Data Bases, VLDB, pp. 294-303, Aug. 1986.

[16] W. Litwin, "Linear Hashing: A New Tool For File and Table Addressing," In Proc. Intl. Conf. on Very Large Data Bases, VLDB, pp. 212-223, 1980.

[17] C. Mohan and F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," IBM Research Report RJ 6846, 1989.

[18] C. Mohan et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," ACM Trans. on Database Systems, Vol. 17, No. 1, pp. 94-162, Mar. 1992.

[19] J. Rao and K. A. Ross, "Cache Conscious Indexing for Decision-Support in Main Memory," In Proc. Intl. Conf. on Very Large Data Bases, VLDB, pp. 78-89, 1999.

[20] J. Rao and K. A. Ross, "Making B+-Trees Cache Conscious in Main

Memory," In Proc. Intl. Conf. on Management of Data, ACM SIGMOD, pp. 475-486, 2000.

[21] A. J. Smith, "Cache Memories," ACM Computing Surveys, Vol. 14, No. 3, pp. 473-530 1982.

[22] S. H. Son(Editor), Special Issue on Real-Time Database Systems, ACM SIGMOD Record, Vol. 17, No. 1, Mar. 1988.

[23] J. S. M. Verhofstad, "Recovery Techniques for Database Systems," ACM Computing Surveys, Vol. 10, No. 2, pp. 167-195, Dec. 1978.

김 상 욱 (Sang-Wook Kim)

1989년 2월: 서울대학교 컴퓨터공학과 졸업(학사)

1991년 2월: 한국과학기술원 전산학과 졸업(석사)

1994년 2월: 한국과학기술원 전산학과 졸업(박사)

1991년 7월-8월: 미국 Stanford University, Computer Science Department 방문 연구원

1994년 2월-1995년 2월: KAIST 정보전자연구소 전문 연구원

1999년 8월-2000년 8월: 미국 IBM T.J. Watson Research Center Post-Doc.

1995년 3월-2000년 8월: 강원대학교 컴퓨터정보통신공학부 부교수

2003년 3월-현재: 한양대학교 정보통신대학 정보통신학부 부교수

〈주관심분야〉 데이타베이스 시스템, 실시간 주기억장치 데이타베이스, 데이타 마이닝, 멀티미디어 정보 검색, 공간 데이터베이스/GIS, 트랜잭션 관리

이 경 태 (Kyung-Tae Lee)

2002년 2월 : 강원대학교 정보통신공학과 졸업(학사)

2002년 3월~현재 : 강원대학교 컴퓨터정보통신공학과 석사과정

〈주관심분야〉 데이터베이스, 소프트웨어 공학

최 완 (Wan Choi)

1981년 2월: 경북대학교 전자공학과(전산전공) 졸업(학사)

1983년 2월: 한국과학기술원 전산학과 졸업(석사)

1985년: 한국과학기술원 전산학과 연구 조교

1988년 8월: 정보처리 기술사(전자계산기조직응용) 자격 소지

1985년 3월-현재: 한국전자통신연구원 컴퓨터시스템 연구부 책임연구원

〈주관심분야〉 시스템 소프트웨어(컴파일러, DBMS, OS), 소프트웨어 공학, 바이오 데이터베이스, 웹 서비스