주기억장치 DBMS를 위한 인덱스 관리자의 설계 및 구현

정회원 김 상 욱*, 염 상 민*, 김 윤 호*, 이 승 선**, 최 완**

Design and Implementation of an Index Manager for a Main Memory DBMS

Sang-Wook Kim*, Sang-Min Yeom*, Yun-Ho Kim*, Seung-Sun Lee**, Wan Choi**
Regular Members

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. 인덱스 관리자는 객체에 대한 빠른 검색 기능을 지원하는 필수적인 DBMS 서브 컴포넌트이다. 기존의 연구 결과로서 다양한 인덱스 구조가 제안된 바 있으나, 실제 DBMS 상에서 인덱스 관리자를 개발하는 경우에 발생하는 실질적인 이슈들에 대해서는 거의 언급하고 있지 않다. 본 논문에서는 Tachyon의 인덱스 관리자의 개발 중에 경험한 실질적인 구현 이슈들을 언급하고, 이들에 대한 해결 방안을 제시한다. 본 논문에서 다루는 주요 이슈들은 (1) 인덱스 엔트리의 집약적 표현, (2) 가변 길이 키의 지원, (3) 다중 애트리뷰트 키의 지원, (4) 중복 키의 지원, (5) 외부 API의 정의, (6) 동시성 제어, (7) 백업 및 회복 등이다. 본 연구 결과를 통하여 향후 MMDBMS 개발자들의 시행 착오를 최소화할 수 있으리라 생각된다.

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 experiences obtained in developing the index manager of the Tachyon, a next-generation MMDBMS. The index manager is an essential sub-component of the DBMS used to speed up the retrieval of objects from a large volume of a database in response to a certain search condition. Previous research efforts on indexing proposed various index structures. However, they hardly dealt with the practical issues occurred in implementing an index manager on a target DBMS. In this paper, we touch these issues and present our experiences in developing the index manager on the Tachyon as solutions. The main issues touched are (1) compact representation of an index entry, (2) support of variable-length keys, (3) support of multiple-attribute keys, (4) support of duplicated keys, (5) definition of external APIs, (6) concurrency control, and (7) backup and recovery. We believe that our contribution would help MMDBMS developers highly reduce their trial-and-errors.

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[14]. 주기억장치 DBMS(main memory DBMS : MMDBMS)는 저장 매체로서 주기억장치를 사용

* 강원대학교 컴퓨터정보통신공학부, ** 한국전자통신연구원 실시간 DBMS 팀
논문번호 : 99269-0703, 접수일자 : 1999년 7월 3일

하며, 이 결과 데이터를 검색 및 갱신할 때 디스크 액세스로 인하여 응답 시간이 지연되는 디스크 기반 DBMS의 문제를 근본적으로 해결한다[10][9][16].

ETRI 실시간 DBMS 팀에서는 강원대학교 정보통신공학과 데이터 및 지식공학 연구실과 공동으로 차세대 MMDBMS Tachyon을 개발 중에 있다. 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. 또한, 다양한 응용 프로그램을 쉽게 수용하기 위하여 객체 지향 모델과 관계형 모델을 모두 지원한다.

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 Tachyon의 인덱스 관리자 개발에 관하여 논의하고자 한다[1].

물론, 지금까지 인덱스에 대한 많은 연구가 수행되어 왔으며, 이러한 연구 결과로 이진 탐색 트리(binary search tree)[19], AVL-트리[1], T-트리[11], B-트리[5], 체인 버켓 해싱(chained bucket hashing)[1], 확장 해싱(extendible hashing)[6], 선형 해싱(linear hashing)[7] 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 중에 경험한 여러 실질적인 구현 이슈들에 관하여 논의하고자 한다. 즉, 본 논문의 공헌은 새로운 인덱스 구조를 제시하는 것이 아니라, 기 개발된 인덱스 구조들 중 MMDBMS Tachyon에 적합한 것은 선정하고, 이를 MMDBMS 상에서 개발하는 과정에서 발생하는 다양한 문제점과 이에 대한 효과적인 해결 방안을 제시하는 것이다. 본 논문에서 다루는 주요 이슈들은 (1) 인덱스 엔트리의 집약적 표현(compact representation), (2) 가변 길이 키(variable-length key)의 지원, (3) 다중 애트리뷰트 키(multiple-attribute key)의 지원, (4) 중복 키(duplicated key)의 지원, (5) 외부 API(application programming interface)의 정의, (6) 효과적인 동시성 제어 방안, (7) 효율적인 백업 및 회복 방안 등이다.

본 논문의 구성은 다음과 같다. 제 2장에서는 개발의 대상이 되는 차세대 MMDBMS Tachyon의 특징, 전체 시스템 아키텍춰, 그리고 각각의 구성요소에 관하여 간략히 소개한다. 제 3장에서는 관련 연구로서 기존에 제안된 다양한 인덱스 구조에 관하여 설명하고, 각 구조의 특성과 장단점에 관하여 논의한다. 제 4장에서는 Tachyon의 인덱스 관리자 개발 과정에서 획득한 다양한 실질적인 구현 이슈와 해결 방안에 관하여 자세히 논의한다. 제 5장에서는 Tachyon의 인덱스를 위한 동시성 제어, 백업, 회복 전략을 제시한다. 끝으로 제 6장에서는 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 지원하는 객체-관계 DBMS(object-relational DBMS)이다.

Tachyon은 현재 Unix를 기반으로 하는 플랫폼을 주요 대상으로 하며, C++ 언어를 사용하여 개발하고 있다. 향후에는 ETRI 실시간 OS 팀에서 개발한 실시간 운영체제인 SROS[21][22]로 플랫폼을 이전시킬 계획을 가지고 있다.

Tachyon의 전체 시스템 아키텍춰는 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 완

1) 현재 전체 MMDBMS의 개발은 진행 중에 있으며, 본 논문에서 기술하는 인덱스 관리자의 개발은 완료된 상태이다.

료된 주기억장치 더미를 풀로 반환시켜 주는 기능을 제공한다.

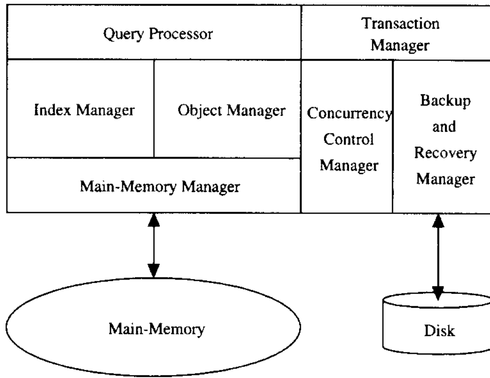


그림 2.1. Tachyon의 전체 시스템 아키텍쳐.

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)[12]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 Tachyon 내에서 수행되는 다수의 트랜잭션들의 수행 과정을 총괄적으로 제어한다. 특히, 마감 시간을 기반으로 하는 트랜잭션 스케쥴링(scheduling)을 수행함으로써 실시간 응용을 효과적으로 지원하도록 한다.

질의 처리 관리자(query processor)는 Tachyon에서 정의하는 SQL(Structured Query Language)[20] 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. 먼저, 제 3.1절에서는 트리 구조를 기반으로 하는 인덱스를 소개하고, 제 3.2절에서는 해쉬 구조를 기반으로 하는 인덱스를 소개한다. 제 3.3절에서는 Tachyon에서 채택한 인덱스 구조의 선정 배경에 대하여 논의한다.

3.1. 트리 인덱스

트리 인덱스는 키 값을 이용한 트리 탐색(tree search)을 통하여 객체의 주소를 파악하므로 완전 일치 질의의 처리 성능은 해쉬 인덱스에 비하여 떨어진다. 그러나 인덱스 내부에서 엔트리들이 키 값의 순서대로 유지되므로 범위 질의의 처리 성능이 매우 뛰어나다.

이진 탐색 트리(binary search tree)[19]는 가장 기본적인 트리 인덱스이며, 검색, 삽입, 삭제 연산이 모두 단순하다는 것이 장점이다. 그러나 트리의 균형성이 보장되지 않으므로 저장되는 키 값의 분포와 삽입 및 삭제되는 순서에 따라 검색 성능이 큰 영향을 받는다는 것이 단점으로 지적된다.

AVL-트리[1]는 이러한 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. 이진 탐색 트리와 더불어 AVL-트리가 갖는 가장 큰 문제점은 저장 공간의 오버헤드가 크다는 것이다.

각 노드는 <키 값, 이 키 값을 갖는 객체의 주소>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 많다는 것이 단점이다.

T-트리[11]는 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 값 당 요구되는 부가 정보의 양이 작아지며, 재균형을 위한 회전 연산의 수도 줄어든다.

B-트리[5]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. B-트리 내의 단말 노드가 가지는 엔트리는 단순히 <키 값, 이 키 값을 갖는 객체의 주소>만을 유지하지만, 비단말 노드가 가지는 각 엔트리는 <키 값, 이 키 값을 갖는 객체의 주소, 이 키 값보다 작은 값들을 가지는 서브 트리의 주소>를 유지한다. 따라서 팬-아웃이 2 이상인 대부분의 환경에서는 T-트리에 비하여 저장 공간의 오버헤드가 크다.

3.2. 해쉬 인덱스

해쉬 인덱스는 키 값을 이용한 계산을 통하여 객체의 주소를 파악하므로 완전 일치 질의의 처리 성능이 트리 인덱스에 비하여 뛰어나다. 그러나 해쉬 함수의 특성 상, 인덱스 내부에서 엔트리들이 키 값의 순서대로 유지될 수 없으므로 범위 질의의 처리 성능은 떨어진다.

체인 버켓 해싱(chained bucket hashing)[1]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 발생되며, 반대의 경우에는 불필요한 저장 공간의 낭비가 발생한다. 객체의 삽입과 삭제가 빈번한 동적 환경에서는 데이터베이스 크기 예측이 사실상 불가능하므로 체인 버켓 해싱의 사용은 적절하지 못하다.

확장 해싱(extendible hashing)[6]은 객체 저장을 위한 데이터 페이지와 디렉토리로 구성되며, 디렉토리가 데이터베이스의 크기에 적합한 정도로 변화할 수 있는 동적인 구조이다. 디렉토리는 항상 2k(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.

확장 해싱과 같이 동적인 구조를 가지는 선형 해싱(linear hashing)[7]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 [11]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 균형성과 저장 공간 오버헤드를 고려하여 T-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[2].

2) 이러한 장점들로 인하여 현재 여러 MMDBMS에서 T-트리를 인덱스 구조로서 채택하고 있다.

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

확장 해싱이나 선형 해싱과 같은 동적 구조를 가지는 해쉬 인덱스는 계산을 통하여 객체가 속하는 데이터 페이지의 위치를 파악하므로, 디렉토리(확장 해싱과 변형된 선형 해싱의 경우) 혹은 데이터 페이지들의 집합(선형 해싱의 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 T-트리에서 객체 검색을 위한 트리 탐색과 차이가 없게 된다. 특히, T-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 T-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 질의 뿐만 아니라 완전 일치 질의의 처리에서도 T-트리를 사용하도록 결정하였다[3].

Ⅳ. 인덱스 관리자의 설계 및 구현

본 장에서는 인덱스 구조로 선정된 T-트리를 소개하고, Tachyon 상에서 T-트리를 구현한 구체적인 전략에 관하여 논의한다. 먼저, 제 4.1절에서는 T-트리의 특성에 관하여 간략히 소개하고, 제 4.2절에서는 가변 길이 키와 다중 애트리뷰트 키를 효과적으로 지원하는 방안에 관하여 논의한다. 제 4.4절에서는 중복 키를 지원하는 방안에 관하여 논의한다. 끝으로, 제 4.5절에서는 인덱스 관리자를 편리하게 사용할 수 있도록 정의된 API를 제시한다.

4.1. T-트리 개요

T-트리[11]는 AVL-트리[1]의 변형으로서 하나의 노드 내에 다수의 엔트리들이 키 값의 순서에 의하여 정렬된 상태로 저장된다. T-트리에 속하는 모든 노드의 왼쪽 서브 트리와 오른쪽 서브 트리의 높이 차는 1 이하이며, 따라서 항상 일정한 정도의 균형을 유지한다. 객체의 삽입으로 인하여 T-트리의 불균형이 발생되면, AVL-트리와 마찬가지로 회전 연산을 이용한 재균형(rebalancing)을 통하여 이를 해

3) 이러한 단일 인덱스 선정으로 인하여 동시성 제어 관리자와 백업 및 회복 관리자 등 다른 Tachyon 서브 시스템들을 위한 개발의 노력이 절반으로 줄어드는 부가적인 장점도 얻을 수 있다.

결한다. 한 노드 내에 다수의 엔트리를 저장하므로 이러한 재균형 연산의 발생 빈도는 AVL-트리에 비하여 작다.

그림 4.1은 T-트리의 구조를 나타낸 것이다. T-트리는 세 가지 종류의 노드들을 사용한다. 내부 노드(internal node)는 두 개의 서브 트리를 가지는 노드를 의미하며, 절반 리프 노드(half-leaf node)는 하나의 서브 트리만을 가지는 노드를 의미한다. 리프 노드(leaf node)는 서브 트리를 가지지 않는 노드를 의미한다. 내부 노드와 절반 리프 노드는 항상 미리 정해진 최소 개수와 최대 개수 사이의 엔트리들의 저장을 보장하며, 리프 노드는 최소 개수의 보장 없이 단지 최대 개수 이하의 엔트리들만을 저장한다.

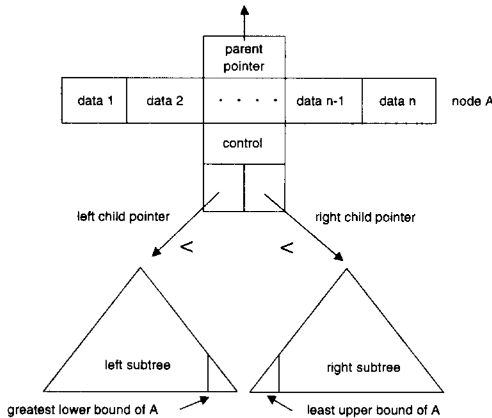


그림 4.1. T-트리 인덱스의 구조

노드 N과 키 값 X에 대하여 X가 N 내의 최대 키 값과 최소 키 값 사이에 존재하는 경우, N은 X를 바운드(bound)한다고 정의한다. 각각의 내부 노드 A에 대하여 A 내의 최소 키 값의 직후 값을 가지는 리프 혹은 절반 리프 노드가 존재하고, 마찬가지로 A 내의 최대 키 값의 직전 값을 가지는 리프 혹은 절반 리프 노드가 존재한다. 이 직전 값을 A의 GLB(greatest lower bound)라 하고, 직후 값을 A의 LUB(least upper bound)라 한다.

T-트리에서의 검색은 이진 탐색 트리에서의 검색과 유사하다. 이진 탐색 트리에서는 노드에서 한 번의 키 값의 비교를 수행하는 반면, T-트리에서는 노드의 최소 키 값 및 최대 키 값과 각각 비교를 수행해야 한다는 것이 차이점이다. 검색은 우선 찾고자 하는 키 값을 바운드하는 노드를 트리 상에서 찾는 트리 탐색 단계와 해당 노드 상에서 키 값을 찾아내는 이진 탐색 단계로 구성된다.

T-트리에서의 삽입은 먼저 삽입될 키 값을 바운드하는 노드를 찾은 후, 해당 키 값을 그 노드내에 삽입한다. 이때, 오버플로우가 발생하면, 그 노드의 최소 값을 이 노드의 GLB를 포함하는 리프 노드(혹은 절반 리프 노드)로 옮김으로써 새로운 GLB가 되도록 한다. 만일, 삽입될 키 값을 바운드하는 노드가 발견되지 않는 경우에는 트리 탐색이 종료된 리프 노드(혹은 절반 리프 노드)내에 키 값을 삽입한다. 이 리프 노드(혹은 절반 리프 노드)에 오버플로우가 발생하면, 이 노드의 하위에 새로운 리프 노드를 생성한다. 이 결과, 트리의 균형이 깨어지면 AVL-트리와 같은 회전 연산을 수행한다.

T-트리에서의 삭제는 먼저 삭제될 키 값을 바운드하는 노드를 찾은 후, 이 노드 내에서 해당 키 값을 삭제한다. 이때, 언더플로우가 발생하면, 이 노드에 대한 GLB를 포함하는 리프 노드(혹은 절반 리프 노드)로부터 GLB를 이 노드로 옮긴다. 이 결과, 절반 리프 노드에 언더플로우가 발생하면, 역시 이 노드의 GLB를 포함하는 리프 노드로부터 GLB를 이 노드로 옮긴다. 이러한 GLB 옮김으로 인하여 리프 노드가 완전히 비게 되면4), 이 리프 노드를 트리에서 삭제한다. 이 결과, 트리의 균형이 깨어지면 AVL-트리와 같은 회전 연산을 수행한다.

4.2. 가변 길이 키 및 다중 애트리뷰트 키의 지원

길이가 고정되지 않는 애트리뷰트가 T-트리의 키로 사용되는 경우, 이를 가변 길이 키라 한다. 디스크 기반 DBMS에서 널리 사용되는 B-트리의 인덱스 엔트리는 <키 값, 이 키 값을 갖는 객체의 주소>로 구성하는 것이 일반적이며, 따라서 이 엔트리 자체가 가변 길이로 될 수 있다. DBMS와 같이 규모가 큰 시스템을 개발하는 경우, 이와 같은 가변 길이의 지원은 관련된 각종 알고리즘의 복잡도를 크게 높이게 되며, 특히 동시성 제어와 회복에 관련된 부분의 문제가 매우 복잡하게 된다[18][17].

본 개발에서는 T-트리 엔트리에서 키 값을 직접 유지하지 않고 단순히 <객체의 주소>만을 유지하는 방법을 사용한다. 이 객체의 주소를 이용하여 키에 참여하는 애트리뷰트 값을 객체로부터 참조할 수 있으므로 키 값을 T-트리 내부에서 관리하지 않더라도 키 값의 비교가 가능하다. 이와 같이, 엔트리 내에 단순히 객체의 주소만을 관리함으로써 다음과

4) T-트리에서는 리프 노드에 한하여 언더플로우를 허용한다.

같은 장점을 얻을 수 있다. 첫째, 전술한 바와 같이 T-트리 관리자는 고정 길이의 <객체의 주소>만을 엔트리로 간주하게 되어 가변 길이 지원을 염두에 두지 않아도 되므로, T-트리 연산 알고리즘은 물론 관련된 동시성 제어 및 회복 알고리즘 등의 복잡도가 크게 줄어든다. 둘째, 키 값을 엔트리 내부에서 관리하지 않아도 되므로 T-트리 유지를 위한 저장 공간의 오버헤드가 줄어든다.

그러나 키 값의 비교를 위해서는 T-트리에서 객체의 어떤 애트리뷰트가 키로 사용되는가를 알아야 한다. 이를 위하여 시스템 카탈로그(system catalog) 에서는 각 T-트리에 관한 정보로서 다음과 같은 TtreeInfo를 유지한다.

UorD
root
numAttributes
attrDesc[0]
• •
attrDesc[MAX-1]

UorD는 현재 사용 중인 T-트리가 중복된 키를 허용하는가의 여부를 나타낸다. root는 해당 T-트리의 루트 노드의 주소를 나타내며, numAttributes는 키를 구성하는 애트리뷰트의 수를 나타낸다. 즉, MAX개 까지의 attrDesc들이 존재하는 것을 허용함으로써 하나의 키가 다수의 애트리뷰트들로 구성되는 다중 애트리뷰트 키 기능을 제공한다. 이 결과, 이러한 기능을 요구하는 다양한 응용 분야를 지원할 수 있다. attrDesc[]는 키를 구성하는 각 애트리뷰트에 관한 정보로서 <offset, size, dataType>으로 구성된다. 이것은 각각 객체 내에서 해당 애트리뷰트의 시작 위치, 애트리뷰트 값의 최대 크기, 그리고 데이터 타입을 의미한다. 키 값의 비교는 키를 구성하는 각각의 애트리뷰트 값의 비교를 반복함으로써 수행된다.

여기서 나타나는 가장 자연스러운 의문의 하나는 제안하는 기법을 B-트리 인덱스에 적용하면 어떨까 하는 것이다. 즉, 인덱스 엔트리에서 키 값을 제거하고, 단순히 객체의 주소만을 유지시키도록 하는 것이다. 그러나 디스크 기반 DBMS에서는 객체들이 모두 디스크 내에 저장되므로 해당 키 값을 같은 B-트리 노드 내에 유지시키지 않으면, 각각의 키 값을 비교할 때마다 새로운 디스크 액세스가 발생하므로 심각한 성능 저하가 발생한다. 반면, MMDBMS에서는 모든 데이터가 주기억장치 내에 상주하므로 이러한 문제가 발생하지 않는다.

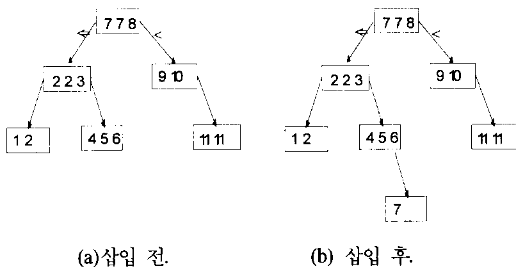
4.3. 중복 키의 지원

중복 키란 같은 값을 가지는 서로 다른 객체들의 존재를 허용하는 키를 의미한다. 디스크를 기반으로 하는 환경에서 널리 사용되는 B-트리 인덱스에서는 이러한 중복 키의 지원을 위하여 <키 값, 이 키 값을 갖는 객체의 주소들의 리스트>의 형태를 갖는 엔트리를 사용한다[18]. 또한, 많은 키 값들의 중복으로 인하여 한 엔트리가 하나의 노드 내에 관리될 수 없는 경우에는 별도의 오버플로우 노드들을 연결하는 방식을 사용한다. 그러나 이러한 방식은 중복 키의 지원을 특별한 경우로 간주하므로 관련된 알고리즘의 복잡도가 높아진다.

본 개발에서는 T-트리 엔트리 내부에 키 값을 유지하지 않으므로 같은 키 값을 갖는 객체들의 주소 리스트를 하나의 엔트리 내에 함께 관리할 필요가 없다. 따라서 같은 키 값을 갖는 객체에 대한 인덱스 엔트리들을 별도로 관리하는 방식을 사용한다. 즉, 서로 다른 두 객체들이 같은 키 값을 갖는 경우에도 이 두 객체들과 대응되는 인덱스 엔트리들은 독립적으로 관리된다. 따라서 B-트리 인덱스에서 중복 키를 관리하는 방식에서와 같은 별도의 관리 메카니즘 없이 원래의 방식대로 관리할 수 있다. 그림 4.1에서 나타난 바와 같이, 중복 키를 지원하지 않는 참고 문헌 [11]의 방식과의 유일한 차이는 각 노드의 왼쪽 서브트리 내의 모든 엔트리들의 키 값은 이 노드의 최소 키 값 보다 <작다>라고 정의했던 것을 <작거나 같다>로 바꾼다는 것이다.

그림 4.2는 중복 키를 허용하는 T-트리 구조에서 중복된 새로운 키 값 7이 삽입되는 과정을 보여준다. 그림 4.2(a)는 삽입 직전의 상태를 나타내고, 그림 4.2(b)는 삽입 직후의 상태를 나타낸다. 여기서 블러킹 인수(blocking factor)는 3임을 가정하였으며, 노드 내의 엔트리는 설명의 편의를 위하여 객체 주소가 아닌 객체의 키 값으로 표현하였다. 루트 노드가 7을 바운드하지만 삽입될 공간이 없으므로 하위 단계 서브 트리로 계속 내려가 새로운 리프 노드를 만들게 된다. 이와 같은 방식을 기반으로 중복 키를 지원함으로써 중위 순회(inorder traversal)를 통하여 트리 내의 모든 엔트리들을 키 값 순서대로 액세스할 수 있도록 T-트리를 유지시킬 수 있다[5].

5) 단, 그림 4.2에서와 같이 동일한 키 값을 갖는 엔트리



(a)삽입 전. (b) 삽입 후.

그림 4.2. 중복된 키 값 7의 삽입 과정.

4.4. 외부 APIs

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

TtreeIndex 클래스는 MemAllocPtr, UorD, CurrentPosition, TtreeInfo의 네 가지 멤버 변수를 갖는다. MemAllocPtr은 T-트리 노드와 같은 주기억장치 더미를 동적으로 할당해 주는 Tachyon의 주기억장치 관리자 클래스 인스턴스에 대한 포인터이며, UorD는 해당 T-트리가 중복된 키를 허용하는가의 여부를 나타낸다. CurrentPosition는 T-트리를 이용하여 범위 질의를 처리할 때, 현재 처리 중인 엔트리의 주소를 갖는다. 끝으로, TtreeInfo는 제 4.2절에서 언급한 바와 같이 루트 노드의 주소, 키를 구성하는 애트리뷰트들의 특성 등 해당 T-트리에 관한 정보가 저장된 시스템 카탈로그 내의 엔트리 주소를 갖는다.

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

TtreeIndex(MemAllocPtr,TtreeInfo)는 TtreeIndex 클래스의 구성자(constructor)인 동시에, 해당 T-트리를 오픈하는 역할을 한다. 즉, 해당 T-트리에 관한 정보를 가지는 엔트리를 시스템 카탈로그에서 찾아 멤버 변수인 TtreeInfo가 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 T-트리 내에 삽입하는 함수이며, delete (OID)는 OID가 가리키는 객체와 대응되는 엔트리를 T-트리로부터 삭제하는 함수이다. 객체 삽입 시에는 실제 객체를 먼저 삽입한 후에 이에 대한 엔트리를 T-트리에 삽입하고, 반대로 삭제 시에는 T-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, T-트리 검색을 통하여 5 보다 크고, 15 보다 작은 첫 번째 객체가 OID에 반환된다. 또한, TtreeIndex 클래스의 멤버 변수 CurrentPosition에 이 객체와 대응되는 T-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, T-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()

들이 다수의 노드 내에 산재될 수 있으므로 질의 처리 시 검색 키 값을 바운드하는 최초의 노드를 발견하더라도 이 노드의 GLB를 확인함으로써 검색 키 값을 갖는 엔트리를 간과하지 않도록 해 준다.

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 TtreeIndex 클래스의 멤버 변수인 CurrentPosition이 나타내는 엔트리를 T-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 T-트리로부터 찾는다. 이때, 해당 엔트리의 물리적인 위치는 CurrentPosition에서 관리된다. 실제 객체는 엔트리 내의 OID를 이용하여 객체 관리자를 호출함으로써 삭제시키고, 엔트리는 deleteCurrent() 함수를 이용하여 T-트리로부터 삭제시킨다.

이후에는 getNext()(혹은 getPrev()) 함수를 이용하여 조건을 만족하는 다음 객체와 대응되는 엔트리를 T-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 T-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 T-트리의 구조적인 변화가 발생할 수 있으므로 재 설정 작업은 deleteCurrent() 함수의 최종 작업 단계에서 수행하도록 한다[6].

6) 삭제에 의한 병합 연산에 의하여 엔트리의 위치가 변화될 수 있으므로 이러한 경우 CurrentPosition의 위치를 재조정한다. 그러나 재 균형 연산이 발생하는 경우에는 전체 T-트리의 구조는 변경되지만, CurrentPosition은 이에 영향을 받지 않으므로 이러한 재조정 작업은 불필요하다. 그 이유는 재 균형 연산이 발생하기 전과 후의 T-트리 구조는 상이하지만, 이 두가지 구조 모두 키 값들을 올바른 순서로 유지한다는 점에서는 차이가 없기 때문이다.

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

V. 동시성 제어, 백업, 회복 기능의 지원

본 장에서는 Tachyon의 인덱스에 대하여 동시성 제어와 백업 및 회복 기능을 지원하는 효과적인 방안을 제시한다. 제 5.1절에서는 인덱스의 동시성 제어 방안에 관하여 논의하고, 제 5.2절에서는 인덱스를 이용하여 유령 현상을 해결하는 방안에 관하여 기술한다. 제 5.3절에서는 인덱스를 위한 백업 및 회복 방안을 제시한다.

5.1. 동시성 제어

데이터베이스 내에서 관리되는 사용자 객체들과 마찬가지로 인덱스도 다수의 트랜잭션들이 동시에 액세스하는 공유 데이터이므로 이에 대한 동시성 제어가 요구된다. 인덱스는 대부분의 트랜잭션들에 의하여 공통적으로 액세스되는 데이터이므로 일반 객체의 동시성 제어를 위하여 사용되는 이단계 라킹 규약(two-phase locking protocol)[2]을 동일하게 적용한다면 매우 심각한 동시성의 저하가 발생한다. 예를 들어, T-트리의 루트 노드에 대하여 각 트랜잭션의 종료 시까지 락을 걸도록 한다면, 트랜잭션들을 순차적으로 수행하는 것과 동일한 결과를 초래하게 된다. 따라서 인덱스에 이단계 라킹 규약을 적용하는 방식은 부적절하다.

본 연구에서는 이를 해결하기 위하여 각 T-트리마다 고유의 래치(latch)[15][17]를 둠으로써 동시성을 제어하는 방식을 제시한다. 각 트랜잭션은 T-트리를 액세스하기 직전 해당되는 래치를 걸고[7], 이에 대

한 액세스가 끝나면 이 래치를 즉시 풀어준다. 이단계 라킹 규약이 아닌 이러한 래치 만으로 동시성 제어가 가능한 이유는 인덱스가 단지 사용자 객체의 효과적인 검색을 위하여 필요한 메타 데이터(meta data)이기 때문이다. 즉, 사용자는 이러한 메타 데이터 자체의 내용에는 관심이 없고, 단지 데이터베이스 내에 저장된 객체들의 내용만을 참조하게 된다. 따라서 메타 데이터는 이단계 라킹 규약을 적용하여 논리적 일관성을 보장할 필요가 없으며, 래치를 이용하여 물리적 일관성만을 보장하면 된다[15][17]. 제안된 방식을 이용함으로써 T-트리의 물리적 일관성을 보장할 수 있으며, T-트리를 액세스하는 동안만 래치를 걸게 되므로 동시성을 크게 향상시킬 수 있다.

참고 문헌 [15]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. ARIES/IM에서는 동시에 최대 두 개까지의 래치만을 걸 수 있도록 허용한다. 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 본 연구에서 제시하는 방식 보다 동시성을 좀더 높일 수 있다는 것이 장점이다. 첫째, 전체 트리가 아닌 액세스하고자 하는 노드에만 래치를 걸게 되므로 트리의 그 외 부분은 다른 트랜잭션들이 동시에 액세스할 수 있다. 둘째, 동시에 획득 가능한 래치의 수를 두 개로 제한하므로 트리상의 같은 경로를 액세스하고자 하는 다른 트랜잭션의 대기 시간이 짧아진다.

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 T-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.

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

5.2. 유령 현상

객체의 삽입과 삭제가 발생하는 동적인 데이터베이스에서는 한 트랜잭션이 같은 세그먼트를 두 번 이상 액세스하였을 때, 검색 결과로 나타나는 객체 집합이 상이하게 나타나는 유령 현상(phantom problem)[3]이 발생할 수 있다. 유령 현상을 해결하기 위한 전통적인 방법으로 술어 라킹 기법(predicate locking)[3]이 제안된 바 있으나, 처리 알고리즘의 복잡도(complexity)가 NP-complete 이므로 실제로 널리 사용되지는 않는다. 참고 문헌 [15]에서는 유령 현상을 해결하는 보다 현실적인 방법으로서 인덱스를 이용한 차순 키 라킹 기법(next-key locking method)을 제안하였다. 본 연구에서는 차순 키 라킹 기법을 Tachyon에 적합하도록 변형하여 사용한다.

일반적으로 트랜잭션이 범위 질의를 수행하는 경우에는 질의 조건을 만족하는 첫 키 값에서부터 마지막 키 값까지 각 키 값을 가지는 모든 객체들에 대하여 공유 락(shared lock)을 유지한다. 또한, 트랜잭션이 객체를 삽입하거나 삭제하는 경우에는 해당 객체에 대하여 배제 락(exclusive lock)을 유지한다.

차순 키 라킹 기법에서는 트랜잭션의 범위 질의 수행 시, 질의 조건을 만족하는 마지막 키 값의 차순 키 값(next-key value)을 가지는 객체에도 추가로 공유 락을 걸도록 한다. 또한, 트랜잭션의 객체 삽입 수행 시, 이 객체의 키 값의 차순 키 값을 가지는 객체에도 추가로 배제 락을 걸도록 한다. 객체를 삽입하고자 하는 트랜잭션은 범위 질의를 수행하는 트랜잭션이 종료되기 전까지는 차순 키 값을

가지는 객체에 대한 락을 획득할 수 없다. 따라서 삽입을 수행하는 트랜잭션은 대기해야 하며, 이 결과 유령 현상의 발생을 방지할 수 있다. 트랜잭션이 객체를 삭제하는 경우, 마찬가지로 이 객체의 키 값의 차순 키 값을 가지는 객체에도 추가로 배제 락을 걸도록 한다. 이것은 삭제를 수행한 트랜잭션의 철회로 인하여 발생되는 유령 현상을 방지하기 위한 것이다.

Tachyon에서는 락의 단위(lock granule)로서 객체들의 저장 단위인 파티션(partition)을 채택하고 있다. 본 연구에서는 참고 문헌 [15]에서 제안한 객체 단위의 차순 키 라킹 기법을 변형하여 파티션 단위의 차순 키 라킹 기법을 적용하고자 한다. 즉, 트랜잭션이 범위 질의를 수행하는 경우, 질의 조건을 만족하는 <첫 키 값에서부터 마지막 키 값까지 각 키 값을 가지는 모든 객체들>과 <마지막 키 값의 차순 키 값을 가지는 객체>를 저장하는 각각의 파티션에 대하여 공유 락을 유지한다. 또한, 트랜잭션이 객체를 삽입하거나 삭제하는 경우에는 <해당 객체>와 <이 객체의 키 값의 차순 키 값을 가지는 객체>를 저장하는 파티션에 대하여 배제 락을 유지한다[8]. 이러한 기법을 통하여 유령 현상의 발생을 방지할 수 있다.

차순 키 라킹 기법의 실제 적용을 위해서는 <범위 질의에서 키로 사용되는 애트리뷰트에 대해서는 항상 인덱스가 존재해야 한다>는 전제가 만족되어야 한다. 인덱스 없이 차순 키 값을 가지는 객체를 찾기 위해서는 전체 세그먼트를 모두 액세스해야 하기 때문이다. 그러나 물리적 데이터베이스 설계 과정(physical database design)을 통한 올바른 인덱스 선정이 데이터베이스 구축의 필수적인 과정임을 고려하면 이 전제에 큰 무리가 없음을 알 수 있다.

5.3. 백업 및 회복 기능

백업 및 회복 기능은 시스템에서 발생되는 각종 오류로부터 데이터베이스를 보호하기 위한 필수적인 기능이다[4][8]. 백업 및 회복 관리자는 시스템에서 발생할 수 있는 다양한 오류에 대비하여 정상 동작 시 변경 연산에 대한 로그 레코드를 기록하는 로깅(logging)[12] 작업을 수행하고, 주기적으로 주기억장치 내의 데이터베이스를 디스크로 백업시킴으로써 오류 발생시 백업된 데이터베이스와 로깅 정보를 이용하여 데이터베이스를 일관성 있는 상태로 회복시켜 준다.

디스크 기반 DBMS에서는 인덱스의 변경 연산에 대해서도 대응되는 로그 레코드를 기록하는 방식을 사용한다[15][17]. 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 모든 데이터베이스 연산이 주기억장치 내에서 수행되므로, 디스크 액세스를 유발하는 로깅 작업은 전체 MMDBMS 성능 저하의 가장 큰 원인이 된다. 따라서 기존의 디스크 기반 DBMS의 인덱스 로깅 방식을 MMDBMS에 그대로 적용하는 경우, 다수의 로그 레코드 생성으로 인하여 전체 시스템 성능이 크게 저하된다. 따라서 회복을 위하여 요구되는 디스크 액세스 수를 최소화할 수 있는 방법이 요구된다.

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, 시스템의 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. 넷째, 로깅 작업을 기반으로 하는 인덱스의 백업 및 회복은 매우

8) 차순 키 값을 가지는 객체는 다음과 같이 정의된다. 범위 질의의 수행 시에는 해당 범위 질의에서 사용되는 인덱스 상에서 검색 키 값의 차순 키 값을 갖는 객체로 정의된다. 삽입 혹은 삭제 시에는 현재 구성된 모든 인덱스 상에서 해당 객체가 가지는 키 값의 차순 키 값을 갖는 객체들로 정의된다. 이 경우에는 차순 키 값을 가지는 객체가 두 개 이상 존재할 수 있으며, 이 객체들을 저장하는 각각의 파티션에 락이 걸리게 된다.

복잡한 과정을 요구한다. 제안하는 방식을 통하여 이러한 복잡도를 낮출 수 있으므로 개발 기간을 단축시키는데도 일조할 수 있다.

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 제시하고자 한다. 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 주기억장치 내에서 유지하며, 해당 트랜잭션이 종료됨과 동시에 이를 무시함으로써 디스크 액세스를 방지시킬 수 있다[9].

요약하면, 제안된 방식을 MMDBMS에 적용함으로써 시스템 정상 동작, 백업 동작, 시스템 오류로부터의 회복 동작, 트랜잭션 오류로부터의 회복 동작 시의 성능을 개선시킬 수 있다. 특히, MMDBMS에서 모든 데이터베이스가 주기억장치 내에 상주한다는 것을 고려하면, 이러한 성능 개선 효과는 매우 클 것이라고 사료된다.

V. 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 논문에서는 차세대 MMDBMS Tachyon의 인덱스 관리자 개발에 관하여 논의하였다. 인덱스 관리자는 객체에 대한 빠른 검색 기능을 지원하는 DBMS의 필수 서브 시스템이다. 기존의 연구 결과로서 다양한 인덱스 구조들이 제안된 바 있으나, 개발 과정에서 발생하는 실질적인 구현 이슈들에 대해서는 언급하고 있지 않다. 본 논문에서는 Tachyon의 인덱스 관리자 개발 중에 경험한 실질적인 구현 이슈들을 언급하고, 이들에 대한 해결 방안을 제시하였다.

본 논문에서 다루었던 주요 이슈들은 (1) 인덱스 엔트리의 집약적 표현, (2) 가변 길이 키의 지원, (3) 다중 애트리뷰트 키의 지원, (4) 중복 키의 지원, (5) 외부 API의 정의, (6) 동시성 제어 방안, (7) 백업 및 회복 방안 등이다. 이러한 공헌을 통하여 향후 MMDBMS 개발자들의 시행 착오를 최소화할 수 있으리라 생각된다.

향후 연구 방향으로는 본 연구에서 세부적으로 제안한 각각의 해결 방안의 성능 개선 효과를 분석적 혹은 실험적으로 검증하는 것을 고려하고 있다.

9) 참고 문헌 [13]에서는 로그 레코드로 인한 디스크 액세스 수를 최소화하기 위한 전략으로서 MMDBMS의 트랜잭션 종료 시 주기억장치 내에서 관리되던 전체 로그 레코드들 중 시스템 오류로부터의 회복을 위하여 필요한 정보만을 선별하여 디스크로 쓰는 방식은 제안한 바 있다. 본 연구에서는 이 방식을 채택하여 논리적인 로그 레코드의 로깅 작업이 디스크 액세스를 유발시키지 않도록 한다.

감사의 글

본 연구는 한국과학재단 99 해외 Post-Doc 연수 프로그램, 한국전자통신연구소 98 위탁과제(주기억장치 데이터베이스 시스템을 위한 트랜잭션 동시성 제어기 개발), 그리고 강원대학교 멀티미디어연구센터를 통한 정보통신부 정보통신 우수대학원 사업 등의 지원을 받았습니다.

참 고 문 헌

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

[2] 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.

[3] K. P. Eswaran et al., "The Notion of Consistency and Predicate Locks in a Database System," *Comm. of the ACM, Vol. 19, No. 11*, pp. 624-633, Nov. 1976.

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

[5] D. Comer, "The Ubiquitous B-Trees," *ACM*

Computing Surveys, Vol. 11, No. 2, pp. 121-137, 1979.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

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

[13] T. Lehman and M. Carey, "A Recovery Algorithm for a High-Performance Memory-Resident Database System," In Proc. Intl. Conf. on Management of Data, ACM SIGMOD, pp. 104-117, 1987.

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

[15] 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.

[16] 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.

[17] 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.

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

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

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

[21] B. G. Jung et al., "Scalable Architecture for Distributed Real-time Operating Systems," *In Proc. Intl. Conf. on Asia-Pacific Conference on Communications*, APCC-97, pp. 292-296, 1997.

[22] S. I. Jun et al., "SROS: A Dynamically-Scalable Distributed Real-time Operating System for ATM Switching Network", *In Proc. IEEE Global Telecommunications Conference*, pp. 2918-2923, 1998.

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 : 정보전자연구소 Post-Doc.
1995년 3월~현재 : 강원대학교 컴퓨터정보통신공학부 조교수
1999년 8월~현재 : 미국 IBM T.J. Watson Research Center 방문 교수
<주관심 분야> DBMS, 실시간 주기억장치 데이타베이스, 트랜잭션 관리, 데이타 마이닝, 멀티미디어 정보 검색, 공간 데이타베이스/GIS

염 상 민(Sang-Min Yeom)

1998년 2월 : 강원대학교 정보통신공학과 졸업(학사)

2000년 2월 : 강원대학교 컴퓨터정보통신공학과 졸업 (석사)

1999년 8월~12월 : 한국전자통신연구원 파견 연구원

2000년 1월~현재 : (주)메디다스 MN연구실 연구원

<주관심 분야> DBMS, 실시간 주기억장치 데이타 베이스, 멀티미디어 데이타베이스

김 윤 호(Yun-Ho Kim)

1999년 2월 : 강원대학교 정보통신공학과 졸업(학사)

1999년 3월~현재 : 강원대학교 정보통신공학과 대학 원 석사과정(석사)

<주관심 분야> 실시간 주기억장치 DBMS, 인덱스 구조, 동시성 제어

이 승 선(Seung-Sun Lee)

1992년 2월 : 한국과학기술대학 전산학과 졸업(학사)

1994년 2월 : 한국과학기술원 전산학과 졸업(석사)

1994년 3월~현재 : 한국전자통신연구원 실시간 DBMS 팀 연구원

<주관심 분야> 데이타베이스 시스템, 객체지향 데이타베이스, 주기억장치 데이타베이스, 실시간 데이터베이스, 분산 시스템

최 완(Wan Choi)

1981년 : 경북대학교 전자공학과(전산전공)졸업(학사)

1983년 : 한국과학기술원 전산학과 졸업(석사)

1985년 : 한국과학기술원 전산학과 연구조교

1988년 8월 : 정보처리 기술사(전자계산기조직응용) 자격 소지

1985년 3월~현재 : 한국전자통신연구원 실시간 DBMS 팀 책임연구원

<주관심 분야> 실시간 시스템 소프트웨어(컴파일러, DBMS, OS), 실시간 DBMS, 소프트웨어 공학